Complex Event Processing with Triceps CEP v1.0

Developer's Guide

Sergey A. Babkin

Complex Event Processing with Triceps CEP v1.0 : Developer's Guide

Sergey A. Babkin

The author can be contacted by e-mail at <babkin@users.sf.net> or <sab123@hotmail.com>.

Many of the designations used by the manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this manual, and the author was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this manual, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

List of Figures

Chapter 1. The field of CEP

1.1. What is the CEP?

CEP stands for the Complex Event Processing. If you look at Wikipedia, it has separate articles for the Event Stream Processing and the Complex Event Processing. In reality it's all the same thing, with the naming driven by the marketing. I would not be surprised if someone invents yet another name, and everyone will start jumping on that bandwagon too.

In general a CEP system can be thought of as a black box, where the input events come in, propagate in some way through that black box, and come out as the processed output events. There is also an idea that the processing should happen fast, though the definitions of “fast” vary widely.

If we open the lid on the box, there are at least three ways to think of its contents:

- a spreadsheet on steroids
- a data flow machine
- a database driven by triggers

Hopefully you've seen a spreadsheet before. The cells in it are tied together by formulas. You change one cell, and the machine goes and recalculates everything that depends on it. So does a CEP system. If we look closer, we can discern the CEP engine (which is like the spreadsheet software), the CEP model (like the formulas in the spreadheet) and the state (like the current values in the spreadsheet). An incoming event is like a change in an input cell, and the outgoing events are the updates of the values in the spreadsheet.

Only a typical CEP system is bigger: it can handle some very complicated formulas and many millions of records. There actually are products that connect the Excel spreadsheets with the behind-the-curtain computations in a CEP system, with the results coming back to the spreadsheet cells. Pretty much every commercial CEP provider has a product that does that through the Excel RT interface. The way these models are written are not exactly pretty, but the results are, combining the nice presentation of spreadsheets and the speed and power of CEP.

A data flow machine, where the processing elements are exchanging messages, is your typical academical look at CEP. The events represented as data rows are the messages, and the CEP model describes the connections between the processing elements and their internal logic. This approach naturally maps to the multiprocessing, with each processing element becoming a separate thread. The hiccup is that the research in the dataflow machines tends to prefer the non-looped topologies. The loops in the connections complicate the things.

And many real-world relational databases already work very similarly to the CEP systems. They have the constraints and triggers propagating these constraints. A trigger propagates an update on one table to an update on another table. It's like a formula in a spreasheet or a logical connection in a dataflow graph. Yet the databases usually miss two things: the propagation of the output events and the notion of being “fast”.

The lack of propagation of the output events is totally baffling to me: the RDBMS engines already write the output event stream as the redo log. Why not send them also in some generalized format, XML or something? Then people realize that yes, they do want to get the output events and start writing some strange add-ons and aftermarket solutions like the log scrubbers. This has been a mystery to me for some 15 years. I mean, how more obvious can it be? But nobody budges. Well, with the CEP systems gaining popularity and the need to connect them to the databases, I think it will eventually grow on the database vendors that a decent event feed is a competitive advantage, and I think it will happen somewhere soon.

The feeling of “fast” or lack thereof has to do with the databases being stored on disks. The growth of CEP has coincided with the growth in RAM sizes, and the data is usually kept completely in memory. People who deploy CEP tend to want the performance not of hundreds or thousands but hundreds of thousands events per second. The second part of “fast” is connected with the transactions. In a traditional RDBMS a single event with all its downstream effects is one transaction.

Which is safe but may cause lots of conflicts. The CEP systems usually allow to break up the logic into multiple loosely-dependent layers, thus cutting on the overhead.

1.2. The uses of CEP

Despite what Wikipedia says (and honestly, the Wikipedia articles on CEP and ESP are not exactly connected with reality), the pattern detection is **not** your typical usage, by a wide, wide margin. The typical usage is for the data aggregation: lots and lots of individual events come in, and you want to aggregate them to keep a concise and consistent picture for the decision-making. The actual decision making can be done by humans or again by the CEP systems. It may involve some pattern recognition but usually even when it does, it doesn't look like patterns, it looks like conditions and joins on the historical chains of events.

The usage in the cases I know of includes the ad-click aggregation, the decisions to make a market trade, the watching whether the bank's end-of-day balance falls within the regulations, the choosing the APR for lending.

A related use would be for the general alert consoles. The data aggregation is what they do too. The last time I worked with it up close (around 2006), the processing in the BMC Patrol and Nagios was just plain inadequate for anything useful, and I had to hand-code the data collection and console logic. I've been touching this issue recently again at Google, and apparently nothing has changed much since then. All the real monitoring is done with the systems developed in-house.

But the CEP would have been just the ticket. I think, the only reason why it has not been widespread yet is that the commercial CEP licenses had cost a lot. But with the all-you-can-eat pricing of Sybase, and with the Open Source systems, this is gradually changing.

Well, and there is also the pattern matching. It has been lagging behind the aggregation but growing too.

1.3. Surveying the CEP langscape

What do we have in the CEP area now? The scene is pretty much dominated by Sybase (combining the former competitors Aleri and Coral8) and StreamBase.

There seem to be two major approaches to the execution model. One was used by Aleri, another by Coral8 and StreamBase. I'm not hugely familiar with StreamBase, but that's how it seems to me. Since I'm much more familiar with Coral8, I'll be calling the second model the Coral8 model. If you find StreamBase substantially different, let me know.

The Aleri idea is to collect and keep all the data. The relational operators get applied on the data, producing the derived data ("materialized views") and eventually the results. So, even though the Aleri models were usually expressed in XML (though an SQL compiler was also available), fundamentally it's a very relational and SQLy approach.

This creates a few nice properties. All the steps of execution can be pipelined and executed in parallel. For persistence, it's fundamentally enough to keep only the input data (what has been called BaseStreams and then SourceStreams), and all the derived computations can be easily reprocessed on restart (it's funny but it turns out that often it's faster to read a small state from the disk and recalculate the rest from scratch in memory than to load a large state from the disk).

It also has issues. It doesn't allow loops, and the procedural calculations aren't always easy to express. And keeping all the state requires more memory. The issues of loops and procedural computations have been addressed in Aleri by FlexStreams: modules that would perform the procedural computations instead of relational operations, written in SPLASH — a vaguely C-ish or Java-ish language. However this tends to break the relational properties: once you add a FlexStream, usually you do it for the reasons that prevent the derived calculations from being re-done, creating issues with saving and restoring the state. Mind you, you can write a FlexStream that doesn't break any of them, but then it would probably be doing something that can be expressed without it in the first place.

Coral8 has grown from the opposite direction: the idea has been to process the incoming data while keeping a minimal state in the variables and short-term *windows* (limited sliding recordings of the incoming data). The language (CCL) is very SQL-like. It relies on the state of variables and windows being pretty much global (module-wide), and allows the statements to be connected in loops. Which means that the execution order matters a lot. Which means that there are some quite extensive rules, determining this order. The logic ends up being very much procedural, but written in the peculiar way of SQL statements and connecting streams.

The good thing is that all this allows to control the execution order very closely and write things that are very difficult to express in the pure un-ordered relational operators. Which allows to aggregate the data early and creatively, keeping less data in memory.

The bad news is that it limits the execution to a single thread. If you want a separate thread, you must explicitly make a separate module, and program the communications between the modules, which is not exactly easy to get right. There are lots of people who do it the easy way and then wonder, why do they get the occasional data corruption. Also, the ordering rules for execution inside a module are quite tricky. Even for some fairly simple logic, it requires writing a lot of code, some of which is just bulky (try enumerating 90 fields in each statement), and some of which is tricky to get right.

The summary is that everything is not what it seems: the Aleri models aren't usually written in SQL but are very declarative in their meaning, while the Coral8/StreamBase models are written in an SQL-like language but in reality are totally procedural.

Sybase is also striking for a middle ground, combining the features inherited from Aleri and Coral8 in its CEP R5 and later: use the CCL language but relax the execution order rules to the Aleri level, except for the explicit single-threaded sections where the order is important. Include the SPLASH fragments for where the outright procedural logic is easy to use. Even though it sounds hodgy-podgy, it actually came together pretty nicely. Forgive me for saying so myself since I've done a fair amount of design and the execution logic implementation for it before I've left Sybase.

Still, not everything is perfect in this merged world. The SQLy syntax still requires you to drag around all your 90 fields into nearly every statement. The single-threaded order of execution is still non-obvious. It's possible to write the procedural code directly in SPLASH but the boundary where the data passes between the SQLy and C-ish code still has a whole lot of its own kinks (less than in Aleri but still a lot). And worst of all, there is still no modular programming. Yeah, there are “modules” but they are not really reusable. They are tied too tightly to the schema of the data. What is needed, is more like C++ templates. Only preferrably something more flexible and less difficult to debug than the C++ templates.

StreamBase does have modules with parametrizable arguments (“ capture fields”), somewhat like the C++ templates. The limitation is that you can say “and carry any additional fields through unchanged” but can't really specify subsets of fields for a particular usage (“and use these fields as a key”). Or at least that's my understanding. I haven't used it in practice and don't understand StreamBase too well.

1.4. We're not in 1950s any more, or are we?

Part of the complexity with CCL programming is that the CCL programs tend to feel very broken-up, with the flow of the logic jumping all over the place.

Consider a simple example: some incoming financial information may identify the securities by either RIC (Reuters identifier) or SEDOL or ISIN, and before processing it further we want to convert them all to ISIN (since the fundamentally same security may be identified in multiple ways when it's traded in multiple countries, ISIN is the common denominator).

This can be expressed in CCL approximately like this (no guarantees about the correctness of this code, since I don't have a compiler to try it out):

```
// the incoming data
create schema s_incoming (
  id_type string, // identifier type: RIC, SEDOL or ISIN
  id_value string, // the value of the identifier
  // add another 90 fields of payload...
);

// the normalized data
create schema s_normalized (
  isin string, // the identity is normalized to ISIN
  // add another 90 fields of payload...
);

// schema for the identifier translation tables
create schema s_translation (
  from string, // external id value (RIC or SEDOL)
  isin string, // the translation to ISIN
);

// the windows defining the translations from RIC and SEDOL to ISIN
create window w_trans_ric schema s_translation
  keep last per from;
create window w_trans_sedol schema s_translation
  keep last per from;

create input stream i_incoming schema s_incoming;
create stream incoming_ric  schema s_incoming;
create stream incoming_sedol  schema s_incoming;
create stream incoming_isin  schema s_incoming;
create output stream o_normalized schema s_normalized;

insert
  when id_type = 'RIC' then incoming_ric
  when id_type = 'SEDOL' then incoming_sedol
  when id_type = 'ISIN' then incoming_isin
select *
from i_incoming;

insert into o_normalized
select
  w.isin,
  i. ... // the other 90 fields
from
  incoming_ric as i join w_tranc_ric as w
    on i.id_value =  w.from;

insert into o_normalized
select
  w.isin,
  i. ... // the other 90 fields
from
  incoming_sedol as i join w_tranc_sedol as w
    on i.id_value =  w.from;

insert into o_normalized
select
  i.id_value,
  i. ... // the other 90 fields
from
  incoming_isin;
```

Not exactly easy, is it, even with the copying of payload data skipped? You may notice that what it does could also be expressed as procedural pseudo-code:

```
// the incoming data
struct s_incoming (
  string id_type, // identifier type: RIC, SEDOL or ISIN
  string id_value, // the value of the identifier
  // add another 90 fields of payload...
);

// schema for the identifier translation tables
```

```
struct s_translation (
  string from, // external id value (RIC or SEDOL)
  string isin, // the translation to ISIN
);

// the windows defining the translations from RIC and SEDOL to ISIN
table s_translation w_trans_ric
  key from;
table s_translation w_trans_sedol
  key from;

s_incoming i_incoming;
string isin;

if (i_incoming.id_type == 'RIC') {
  isin = lookup(w_trans_ric,
    w_trans_ric.from == i_incoming.id_value
  ).isin;
} elsif (i_incoming.id_type == 'SEDOL') {
  isin = lookup(w_trans_sedol,
    w_trans_sedol.from == i_incoming.id_value
  ).isin;
} elsif (i_incoming.id_type == 'ISIN') {
  isin = i_incoming.id_value;
}

if (isin != NULL) {
  output o_ normalized(isin,
    i_incoming.(* except (id_type, id_value))
  );
}
```

Basically, writing in CCL feels like programming in Fortran in the 50s: lots of labels, lots of GOTOs. Each stream is essentially a label, when looking from the procedural standpoint. It's actually worse than Fortran, since all the labels have to be pre-defined (with types!). And there isn't even the normal sequential flow, each statement must be followed by a GOTO, like on those machines with magnetic-drum main memory.

This is very much like the example in my book [Babkin10], in section 6.4. *Queues as the sole synchronization mechanism*. You can alook at the draft text online at http://web.newsguy.com/sab123/tpopp/06odata.txt. This similarity is not accidental: the CCL streams are queues, and they are the only communication mechanism in CCL.

The SQL statement structure also adds to the confusion: each statement has the destination followed by the source of the data, so each statement reads like it flows backwards.

Chapter 2. Enter Triceps

2.1. What led to it

It had happened that I've worked for a while on and with the Complex Event Processing (CEP) systems. I've worked for a few years on the internals of the Aleri CEP engine, then after Aleri acquired Coral8, some on the Coral8 engine, then after Sybase gobbled up them both, I've designed and did the early implementation of a fair bit of the Sybase CEP R5. After that I've moved on to Deutsche Bank and got the experience from the other side: using the CEP systems, primarily the former Coral8, now known as Sybase CEP R4.

This made me feel that writing the CEP models is unnecessarily difficult. Even the essentially simple things take too much effort. I've had this feeling before as well, but one thing is to have it in abstract, and another is to grind against it every day.

Which in turn led me to thinking about making my own Open Source CEP system, where I could try out the ideas I get, and make the streaming models easier to write. I aim to do better than the 1950's style, to bring the advances of the structured programming into the CEP world.

Thus the Triceps project was born. For a while it was called Biceps, until I've learned of the existence of a recearch project called BiCEP. It's spelled differently, and is in a substantially differnt area of CEP work, but it's easier to avoid confusion, so I went one better and renamed mine Triceps.

Since then I've moved on from DB, and I'm currently not using any CEP at work (though you never know what would happen), but Triceps has already gained momentum by itself.

The Triceps development has been largely shaped by two considerations:

- It has to be different from the Sybase products on which I worked. This is helpful from both legal standpoint and from marketing standpoint: Sybase and StreamBase already have similar products that compete head to head. There is no use getting into the same fray without some major resources.
- It has to be small. I can't spend the same amount of effort on Triceps as a large company, or even as a small one. Not only this saves time but also allows the modifications to be easy and fast. The point of Triceps is to experiment with the CEP language to make it easy to use: try out the ideas, make sure that they work well, or replace them with other ideas. The companies with a large established product can't really afford the radical changes: they have invested much effort into the product, and are stuck with supporting it and providing compatibility into the future.

Both of these considerations point into the same direction: an embeddable CEP system. Adapting an integrated system for an embedded usage is not easy, so it's a good open niche. Yeah, this niche is not empty either. There already is Esper. But from a cursory look, it seems to have the same issues as Coral8/StreamBase. It's also Java-centric, and Triceps is aimed for embeddability into different languages.

And an embeddable system saves on a lot of components.

For starters, no IDE. Anyway, I find the IDEs pretty useless for development in general, and especially for the CEP development. Though it comes handy once in a while for the analysis of the code and debugging.

No new language, no need to develop compilers, virtual machines, function libraries, external callout APIs. Well, the major goal of Triceps actually is the development of a new and better language. But it's one of these paradoxes: Aleri does the relational logic looking like procedural, Coral8 and StreamBase do the procedural logic looking like relational, and Triceps is a design of a language without a language. Eventually there probably will be a language, to be mixed with the parent one. But for now a lot can be done by simply using the Triceps library in an existing scripting language. The existing scripting languages are already powerful, fast, and also support the dynamic compilation.

No separate server executable, no need to control it, and no custom network protocols: the users can put the code directly into their executables and devise any protocols they please. Well, it's not a real good answer for the protocols, since it means that everyone who wants to communicate the streaming data for Triceps over the network has to implement these protocols from scratch. So eventually Triceps will provide a default implementation. But it doesn't have to be done right away.

No data persistence for now either. It's a nice feature, and I have some ideas about it too, but it requires a large amount of work, and doesn't really affect the API.

The language used to implement Triceps is C++, and the scripting language is Perl. Nothing really prevents embedding Triceps into other languages but it's not going to happen anywhere soon. The reason being that extra code adds weight and makes the changes more difficult.

The multithreading support has been a major consideration from the start. All the C++ code has been written with the multithreading in mind. However for the first release the multithreading did not propagate into the Perl API yet.

Even though Triceps is a system aimed for quick experimentation, that does not imply that it's of a toy quality. The code is written in production quality to start with, with a full array of unit tests. In fact, the only way you can do the quick experimentation is by setting up the proper testing from the scratch. The idea of “move fast and break things” is complete rubbish.

2.2. Hello, world!

Let's finally get to business: write a simple “Hello, world!” program with Triceps. Since Triceps is an embeddable library, naturally, the smallest “Hello, world!” program would be in the host language without Triceps, but it would not be interesting. So here is the a bit contrived but more interesting Perl program that passes some data through the Triceps machinery:

```
use Triceps;
use Carp;

$hwunit = Triceps::Unit->new("hwunit") or die "$!";
$hw_rt = Triceps::RowType->new(
  greeting => "string",
  address => "string",
) or confess "$!";

my $print_greeting = $hwunit->makeLabel($hw_rt, "print_greeting", undef, sub {
  my ($label, $rowop) = @_;
  printf("%s!\n", join(', ', $rowop->getRow()->toArray()));
} ) or confess "$!";

$hwunit->call($print_greeting->makeRowop(&Triceps::OP_INSERT,
  $hw_rt->makeRowHash(
    greeting => "Hello",
    address => "world",
  )
)) or confess "$!";
```

What happens there? First, we import the Triceps module and the Carp module. The Carp module isn't strictly necessary but it makes the debugging easier by printing the whole stack trace, not just the line number when an error has happened. The function `confess` is provided by Carp, and works very much like `die` only with the stack trace.

Then we create a Triceps execution unit. An execution unit keeps the Triceps context and controls the execution for one thread. Nothing really stops you from having multiple execution units in the same thread. It might come handy if you want to essentially make multiple ilogical user thread, each with its own unit in one kernel thread. But a single execution unit must never ever be used in multiple threads. It's single-threaded by design and has no synchronization in it.

The argument of the constructor is the name of the unit, that can be used in printing messages about it. It doesn't have to be the same as the name of the variable that keeps the reference to the unit, but it's a convenient convention to make the debugging easier. This is a common idiom of Triceps: when you create something, you give it a name. If any errors occur later with this object, the name will be present int the error message, and you'll be able to find easily, which object has the issue and where it was created.

If something goes wrong, the constructor will return an `undef` and set the error message in `$!`. This actually has turned out to be not such a good idea as it seemed, since writing “`or confess`” (or “`or die`”) at every line quickly turns tedious. And there is usually not much point in catching the errors of this type, since they are essentially the compile-time errors and should cause the program to die anyway.

In the upcoming versions this will be changed throughought the code to confess on any serious errors (and if it needs to be caught, it can be caught with `eval`). Some methods have been already changed to this new convention, but most haven't yet. At the moment it's a bit of a mix.

The next statement creates the type for rows. For the simplest example, one row type is enough. It contains two string fields. A row type does not belong to an execution unit. It may be used in parallel by multiple threads. Once a row type is created, it's immutable, and that's the story for pretty much all the Triceps objects that can be shared between multiple threads: they are created, they become immutable, and then they can be shared. (Of course, the containers that facilitate the passing of data between the threads would have to be an exception to this rule).

Then we create a label. If you look at the “SQLy vs procedural” example in Section 1.4: “We're not in 1950s any more, or are we?” (p. 3) , you'll see that the labels are analogs of streams in Coral8. And that's what they are in Triceps. Of course, now, in the days of the structured programming, we don't create labels for GOTOs all over the place. But we still use labels. The function names are essentially labels, the loops in Perl may have labels. So a Triceps label can often be seen kind of like a function definition, but so far only kind of. It takes a data row as a parameter and does something with it. But unlike a proper function it has no way to return the processed data back to the caller. It has to either pass the processed data to other labels or collect it in some hardcoded data structure, from which the caller can later extract it back. This means that until this gets worked out better, a Triceps label is still much more like a GOTO label or Coral8 stream than a proper function. Just like the unit, a label may be used in only one thread.

A basic label takes a row type for the rows it accepts, a name (again, purely for the ease of debugging) and a reference to a Perl function that will be handling the data. Extra arguments for the function can be specified as well, but there is no use for them in this example.

Here it's a simple unnamed Perl function. Though of course a reference to a named function can be used instead, and the same function may be reused for multiple labels. Whenever the label gets a row operation to process, its function gets called with the reference to the label object, the row operation object, and whatever extra arguments were specified at the label creation (none in this example). The example just prints a message combined from the data in the row.

Note that the label's handler function doesn't just get a row as an argument. It gets a row operation (“rowop” as it's called throughout the code). It's an important distinction. A row just stores some data. As the row gets passed around, it gets referenced and unreferenced, but it just stays the same until the last reference to it disappears, and then it gets destroyed. It doesn't know what happens with the data, it just stores them. A row may be shared between multiple threads. On the other hand, a row operation says “take these data and do such and such a thing with them”. A row operation is a combination of a row of data, an operation code, and a label that has to carry out the operation. It is confined to a single thread. Inside this thread a reference to a row operation may be kept and reused again and again, since the row operation object is also immutable.

Triceps has the explicit operation codes, very much like Aleri (only Aleri doesn't differentiate between a row and row operation, every row there has an opcode in it, and the Sybase CEP R5 does the same). It might be just my background, but let me tell you: the CEP systems without the explicit opcodes are a pain. The visible opcodes make life a lot easier. However unlike Aleri, there is no UPDATE opcode. The available opcodes are INSERT, DELETE and NOP (no-operation). If you want to update something, you send two operations: first DELETE for the old value, then INSERT for the new value. All this will be described in more detail later.

For this simple example, the opcode doesn't really matter, so the label handler function quietly ignores it. It gets the row from the row operation and extracts the data from it into the Perl representation, then prints them. The Triceps row data may be represented in Perl in two ways: an array and a hash. In the array format, the array contains the values of the fields in the order they are defined in the row type. The hash format consists of name-value pairs, which may be stored either in an actual hash or in an array. The conversion from a row to a hash actually returns an array of values which becomes a real hash if it gets stored into a hash variable.

As a side note, this also suggests, how the systems without explicit opcodes came to be: they've been initially built on the simple stateless examples. And when the more complex examples have turned up, they've been aready stuck on this path, and could not afford too deep a retrofit.

The final part of the example is the creation of a row operation for our label, with an INSERT opcode and a row created from hash-formatted Perl data, and calling it through the execution unit. The row type provides a method to construct the rows, and the label provides a method to construct the row operations for it. The `call()` method of the execution unit does exactly what its name implies: it evaluates the label function right now, and returns after all its processing its done.

Chapter 3. Building Triceps

3.1. Downloading Triceps

The official Triceps site is located at SourceForge.

http://triceps.sf.net is the high-level page.

http://sf.net/projects/triceps is the SourceForge project page.

The official releases of Triceps can be downloaded from SourceForge.

The release policy of Triceps is aimed towards the easy development. As the new features are added (or sometimes removed), they are checked into the SVN repository and documented in the blog form at http://babkin-cep.blogspot.com/. Periodically the documentation updates are collected from the blog into this manual, and the official releases are produced.

If you want to try out the most bleeding-edge features that have been described on the blog but not officially released yet, you can get the most recent code directly from the SVN repository. The SVN code can be checked out with

```
svn co https://triceps.svn.sourceforge.net/svnroot/triceps/trunk
```

You don't need any login for check-out. You can keep it current with latest changes by periodically running `svn update`. After you've checked out the trunk, you can build it as usual.

3.2. The reference environment

The tested reference build environment is where I do the Triceps development, and currently it is Linux Fedora 11. The build should work automatically on the other Linux systems as well but has not been tested much in practice.

The build should work on the other Unix environments too but would require some manual configuration for the available libraries, and has not been tested either.

Currently you must use the GNU Linux toolchain: GNU make, GNU C++ compiler (version 4.4.1 has been tested), glibc, valgrind. You can build without valgrind by running only the non-valgrind tests.

The tested Perl version is 5.10.0, and should work on any later version as well. With the earlier versions your luck may vary. The Makefile.PL has been configured to require at least 5.8.0 but you may edit it and try building on the older versions.

I am interested in hearing the reports about builds in various environments.

The normal build expectation is for the 64-bit machines. The 32-bit machines should work (and the code even includes the special cases for them) but have been untested at the moment. Some of the tests might fail of the 32-bit and/or big-endian machines due to the different computation of the hash values, and thus producing a different row order in the result.

3.3. The basic build

If everything works, the basic build is simple, go to the Triceps directory and run:

```
make all
make test
```

That would build and test both the C++ and Perl portions of Triceps. The C++ libraries will be created under `cpp/build`. The Perl libraries will be created under `perl/Triceps/blib`.

The tests are normally run with valgrind for the C++ part, without valgrind for the Perl part. The reason is that Perl produces lots of false positives, and the suppressions depend on particular Perl versions and are not exactly reliable.

If your system differs substantially, you may need to adjust the configurable settings manually, since there is no `./configure` script in the Triceps build yet. More information about them is in the Section 3.9: “Build configuration settings” (p. 17) .

In some circumstances the Per unit test print the error messages as a normal part of running the test. Such messages are prepended with a message that they are expected, for example:

```
Expect message(s) like: Error in PerlSortedIndex(bad) comparator: test a death in
 PerlSortedIndex comparator
Error in PerlSortedIndex(bad) comparator: test a death in PerlSortedIndex comparator
Error in PerlSortedIndex(bad) comparator: test a death in PerlSortedIndex comparator
```

These messages come from the tests of the C++ code catching of the fatal errors in the Perl code. In the future all such catching will be converted to the stack unwind and confession back to the surrounding Perl code. However for now there still are the old-style handlers that simply print the error message.

The other interesting `make` targets are:

`clean`
: Remove all the built files.

`clobber`
: Remove the object files, forcing the libraries to be rebuilt next time.

`vtest`
: Run the unit tests with valgrind, checking for leaks and memory corruption.

`qtest`
: Run the unit tests quickly, without valgrind.

`release`
: Export from SVN a clean copy of the code and create a release package. The package name will be triceps-*version*.tgz, where the *version* is taken from the SVN directory name, from where the current directory is checked out. This includes the build of the documentation.

3.4. Building the documentation

If you have downloaded the release package of Triceps, the documentation is already included it in the built form. The PDF and HTML versions are available in `doc/pdf` and `doc/html`. It is also available online from http://triceps.sf.net.

The documentation is formatted in DocBook, that produces the PDF and HTML outputs. If you check out the source from SVN and want to build the documentation, you need to download the DocBook tools needed to build it. I hate the dependency situations, when to build something you need to locate, build and download dozens of other packages firsti, and then the versions turn out to be updated, and don't want to work together, and all kinds of hell break loose. To make things easier, I've collected the set of packages that I've used for the build and that are known to work. They've collected in http://downloads.sourceforge.net/project/triceps/docbook-for-1.0/. The DocBook packages come originally from http://docbook.sf.net, plus a few extra packages that by now I forgot where I've got from. An excellent book on the DocBook tools and their configuration is [Stayton07]. And if you're interested, the text formatting in Docbook is described in [Walsh99].

DocBook is great in the way it takes cary of great many things automatically but configuring it is plainly a bitch. Fortunately, it's all already taken care of. I've reused the infrastructure I've built for my book [Babkin10] for Triceps. Though some elements got dropped and some added.

Downloading and extraction of the DocBook tools gets taken care of by running

```
make -C doc/dbtools
```

These tools are written in Java, and the packages are already the compiled binaries, so they don't need to be built. As long as you have the Java runtime environment, they just run. However like many Java packages, they are sloppy and often don't return the correct return codes on errors. So the results of the build have to be checked visually afterwards.

The build also uses Ghostscript for converting the figues from the EPS format. The luck with Ghostscript versions also varies. The version 8.70 works for me. I've seen some versions crash on this conversion. Fortunately, it was crashing after the conversion actually succeeded, so a workaround was to ignore the exit code from Ghostscript.

After the tools have been extracted, the build is done by

```
make -C doc/src
```

The temporary files are cleaned with

```
make -C doc/src cleanwork
```

The results will be in `doc/pdf` and `doc/html`.

If like me you plan to use the DocBook tools repeatedly to build the docs for different versions of Triceps, you can download and extract them once in some other directory and then set the exported variable `TRICEPS_TOOLS_BASE` to point to it.

3.5. Running the examples and simple programs

Overall, the examples live together with unit tests. The primary target language for Triceps is Perl, so the examples from the manual are the Perl examples located in `perl/Triceps/t`. The files with names starting with “`x`” contain the examples as such, like `xWindow.t`. Usually there are multiple related examples in the same file.

The examples as shown in the manual usually read the inputs from stdin and print their results on stdout. The actual examples in `perl/Triceps/t` are not quite exactly the same because they are plugged into the unit test infrastructure. The difference is limited to the input/output functions: rather than reading and writing on the stdin and stdout, they take the inputs from variables, put the results into variables, and have the results checked for correctness. This way the examples stay working and do not experience the bit rot when something changes.

Speaking of the examples outputs, the common convention in this manual is to show the lines entered from stdin as bold and the lines printed on stdout as regular font. This way they can be easily told apart, and the effects can be connected to their causes. Like this:

```
OP_INSERT,1,AAA,10,10
Contents:
  id="1" symbol="AAA" price="10" size="10"
lbAverage OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
Contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
lbAverage OP_INSERT symbol="AAA" id="3" price="15"
```

The other unit tests in the `.t` files are interesting too, since they contain absolutely all the possible usages of everything, and can be used as a reference. However they tend to be much more messy and hard to read, exactly because they contain in them lots of tiny snippets that do everything.

The easiest way to start trying out your own small programs is to place them into the same directory `perl/Triceps/t` and run them from there. Just name them with the suffix `.pl`, so that they would not be picked up by the Perl unit test infrastructre (or if you do want to run them as a part of unit tests, use the suffix `.t`).

To make your programs find the Triceps modules, start them with

```
use ExtUtils::testlib;
use Triceps;
use Carp;
```

The module `ExtUtils::testlib` takes care of setting the include paths to find Triceps. You can run them from the parent directory, like:

```
perl t/xWindow.t
```

The parent directory is the only choice, since `ExtUtils::testlib` can not set up the include paths properly from the other directories.

3.6. Installation of the Perl library

If you have the root permissions on the machine and want to install Triceps in the central location, just run

```
make -C perl/Triceps install
```

If you don't, there are multiple options. One is to create your private Perl hierarchy in the home directory. If you decide to put it into `$HOME/inst`, the installation there becomes

```
mkdir -p $HOME/inst
cp -Rf perl/Triceps/blib/* $HOME/inst/
```

You can then set the environment variable

```
export PERL5LIB=$HOME/inst/lib:$HOME/inst/arch
```

to have your private hierarchy prepended to the Perl's standard library path. You can then insert “`use Triceps;`” and the Triceps module will be found. If you want to have the man pages from that directory working too, set

```
export MANPATH=$HOME/inst:$MANPATH
```

Not that Triceps has any usable man pages at the moment.

However if you're building a package that uses Triceps and will be shipped to the customer and/or deployed to a production machine, placing the libraries into the home directory is still not the best idea. Not only you don't want to pollute the random home directories, you also want to make sure that your libraries get picked up, and not the ones that might happen to be installed on the machine from some other sources (because they may be of different versions, or completely different libraries that accidentaly have the same name).

The best idea then is to copy Triceps and all the other libraries into your distribution package, and have the binaries (including the scripts) find them by a relative path.

Suppose you build the package prototype in the `$PKGDIR`, with the binaries and scripts located in the subdirectory `bin`, and the Triceps library located in the subdirectory `blib`. When you build your package, you install the Triceps library in that prototype by

```
cp -Rf perl/Triceps/blib $PKGDIR/
```

Then this package gets archived, sent to the destination machine and unarchived. Whatever the package type, `tar`, `cpio` or `rpm`, doesn't matter. The relative paths under it stay the same. For example, if it gets installed under `/opt/my_package`, the directory hierarchy would look like this:

```
/opt/my_package
     +- bin
     |  +- my_program.pl
     +- blib
        +- ... Triceps stuff ...
```

The script my_program.pl can then use the following code at the top to load the Triceps package:

```
#!/usr/bin/perl

use File::Basename;

# This is the magic sequence that adds the relative include paths.
BEGIN {
  my $mypath = dirname($0);
  unshift @INC, "${mypath}/../blib/lib", "${mypath}/../blib/arch";
}

use Triceps;
```

It finds its own path from $0, by taking its directory name. Then it adds the relative directories for the Perl modules and XS shared libraries to the include path. And finally loads Triceps using the modified include path. Of course, more paths for more packages can be added as well. The script can also use that own directory (if saved into a global instead of my variable) to run the other programs later, find the configuration files and so on.

3.7. Installation of the C++ library

There are no special install scripts for the C++ libraries and includes. To build your C++ code with Triceps, simply specify the location of Triceps sources and built libraries with options -I and -L. For example, if you have built Triceps in $HOME/srcs/triceps-1.0.0, you can add the following to your Makefile:

```
TRICEPSBASE=$(HOME)/srcs/triceps-1.0.0
CFLAGS += -I$(TRICEPSBASE)/cpp -DTRICEPS_NSPR4
LDFLAGS += -L$(TRICEPSBASE)/cpp/build -ltriceps -lnspr4 -pthread
```

The Triceps include files expect that the Triceps C++ subdirectory is directly in the include path as shown.

The exact set of -D flags and extra -l libraries may vary with the Triceps configuration. To get the exact ones used in the configuration, run the special configuration make targets:

```
make --quiet -f cpp/Makefile.inc getconf
make --quiet -f cpp/Makefile.inc getxlib
```

The additions to CFLAGS are returned by getconf. The additional external libraries for LDFLAGS are returned by getxlib. It's important to use the same settings in the build of Triceps itself and of the user programs. The differing settings may cause the program to crash.

If you build your code with the dynamic library, the best packaging practice is to copy the libtriceps.so to the same directory where your binary is located and specify its location with the build flags (for GCC, the flags of other compilers may vary):

```
LDFLAGS += "-Wl,-rpath='$$ORIGIN/.'"
```

Or any relative path would do. For example, if your binary package contains the binaries in the subdirectory bin and the libraries in the subdirectory lib, the setting for the path of the libraries relative to the binaries will be:

```
LDFLAGS += "-Wl,-rpath='$$ORIGIN/../lib'"
```

But locating the binaries and the shared libraries won't work if Triceps and your program get ever ported to Windows. Windows searches for the DLLs only in the same directory.

Or it might be easier to build your code with the static library: just instead of -ltriceps, link explicitly with $(TRICEPSBASE)/cpp/build/libtriceps.a and the libraries it requires:

```
LDFLAGS += $(TRICEPSBASE)/cpp/build/libtriceps.a -lpthread -lnspr4
```

3.8. Disambiguation of the C++ library

A problem with the shared libraries is that you never know, which exact library will end up linked at run time. The system library path takes priority over the one specified in `-rpath`. So if somene has installed a Triceps shared library system-wide, it would be found and used instead of your one. And it might be of a completely different version. Or some other package might have messed with `LD_LIBRARY_PATH` in the user's `.profile`, and inserted its path with its own version of Triceps.

Messing with `LD_LIBRARY_PATH` is bad. The good solution is to give your libraries some unique name, so that it would not get confused. Instead of `libtriceps.so`, name it something like `libtriceps_my_corp_my_project_v_123.so`.

Triceps can build the libraries with such names directly. To change the name, edit `cpp/Makefile.inc` and change

```
LIBRARY := triceps
```

to

```
LIBRARY := triceps_my_corp_my_project_v_123
```

and it will produce the custom-named library. The Perl part of the build detects this name change automatically and still works (though for the Perl build it doesn't change much, the static C++ Triceps library gets linked into the XS-produced shared library).

There is also a special `make` target to get back the base name of the Triceps library:

```
make --quiet -f cpp/Makefile.inc getlib
```

The other potential naming conflict could happen with both shared and dynamic libraries. It appears when you want to link two different versions of the library into the same binary. This is needed rarely, but still needed. If nothing special is done, the symbol names in two libraries clash and nothing works. Triceps provides a way around it by having an opportunity to rename the C++ namespaces, instead of the default namespace “`Triceps`”. It can be done again by editing `cpp/Makefile.inc` and modifying the setting `TRICEPS_CONF`:

```
TRICEPS_CONF += -DTRICEPS_NS=TricepsMyVersion
```

Suppose that you have two Triceps versions that you want both to use in the same binary. Suppose that you are building them in `$(HOME)/srcs/triceps-1.0.0` and `$(HOME)/srcs/triceps-2.0.0`.

Then you edit `$(HOME)/srcs/triceps-1.0.0/cpp/Makefile.inc` and put in there

```
TRICEPS_CONF += -DTRICEPS_NS=Triceps1
```

And in `$(HOME)/srcs/triceps-2.0.0/cpp/Makefile.inc` put

```
TRICEPS_CONF += -DTRICEPS_NS=Triceps2
```

If you use the shared libraries, you need to disambiguate their names too, as described above, but for the static libraries you don't have to.

Almost there, but you need to have your code use the different namespaces for different versions too. The good practice is to include in your files

```
#include <common/Conf.h>
```

and then use everywhere the Triceps namespace `TRICEPS_NS` instead of `Triceps`. Then as long as one source file deals with only one version of Triceps, it can be easily manipulated to which version to use by providing that version in the include path. And you get your program to work with two versions of Triceps by linking the object files produces from these source files together into one binary. Then you just build some of your files with `-I$(HOME)/srcs/triceps-1.0.0/cpp` and some with `-I$(HOME)/srcs/triceps-2.0.0/cpp` and avoid any conflicts or code changes.

At the link time, you will need to link with the libraries from both versions.

3.9. Build configuration settings

Since Triceps has no autoconfiguration yet, it may need to be configured manually for the target operating system. The same method is used for the build options.

The configuration options are set in the file `cpp/Makefile.inc`. The extra defines are added in `TRICEPS_CONF`, the extra library dependencies in `TRICEPS_XLIB`.

So far the only such configurable library dependency is the NSPR4 library. It's used for its implementation of the atomic integers and pointers. Without it the code still works but uses a less efficient implementation of an integer or pointer protected by a mutex. It is enabled by

```
TRICEPS_CONF += -DTRICEPS_NSPR4
TRICEPS_XLIB += -lnspr4
```

The other build options require only the `-D` settings.

```
TRICEPS_CONF += -DTRICEPS_NS=TricepsMyVersion
```

Changes the namespace of Triceps.

```
TRICEPS_CONF += -DTRICEPS_BACKTRACE=false
```

Disables the use of the glibc stack backtrace library (it's a standard part of glibc nowadays but if you use a non-GNU libc, you might have to disable it). This library is used to make the messages on fatal errors more readable, and let you find the location of the error easier.

Chapter 4. API Fundamentals

4.1. Languages and layers

As mentioned before, at the moment Triceps provides the APIs in C++ and Perl. They are similar but not quite the same, because the nature of the compiled and scripted languages is different. The C++ API is more direct and expects discipline from the programmer: if some incorrect arguments are passed, everything might crash. The Perl API should never crash. It should detect any incorrect use and report an orderly error. Besides, the idioms of the scripted languages are different from the compiled languages, and different usages become convenient.

So far only the Perl API is documented in this manual. Its is considered the primary one for the end users, and also richer and easier to use. The C++ API will be documented as well, just it didn't make the cut for the version 1.0. If you're interested in the C++ API, read the Perl documentation first, to understand the ideas of Triceps, and then look in the source code. The C++ classes have very extensive comments in the header files.

The Perl API is implemented in XS. Some people, may wonder, why not SWIG? SWIG would automatically export the API into many languages, not just Perl. The problem with SWIG is that it just maps the API one-to-one. And this doesn't work any good, it makes for some very ugly APIs with abilities to crash from the user code. Which then have to be wrapped into more scripting code before they become usable. So then why bother with SWIG, it's easier to just use the scripting language's native extension methods. Another benefit of the native XS support is the access to the correct memory management.

In general, I've tried to avoid the premature optimization. The idea is to get it working at all first, and then bother about working fast. Except for the cases when the need for optimization looked obvious, and the logic intertwined with the general design strongly ehough, that if done one way, would be difficult to change in the future. We'll see, if these “obvious” cases really turn out to be the obvious wins, or will they become a premature-optimization mess.

There is usually more than one way to do something in Triceps. It has been written in layers: There is the C++ API layer on the bottom, then the Perl layer that closely parallels it, then more of the niceties built in Perl. There is more than one way to organize the manual, structuring it by features or by layers. Eventually I went in the order of the major features, discussing each one of them at various layers.

I've also tried to show, how these layers are built on top of each other and connected. Which might be too much detail for the first reading. If you feel that something is going over your head, just skim over it. It could be marked more clearly but I don't like this kind of marking. I hate the side-panels in the magazines. I like the text to flow smoothly and sequentially. I don't like the “simplifications” that distort the real meaning and add all kinds of confusion. I like having all the details I can get, and then I can skip over the ones that look too complicated (and read them again when they start making sense).

Also, a major goal of Triceps is the extendability. And the best way to learn how to extend it, is by looking up close at how it has already been extended.

4.2. Errors, deaths and confessions

In Perl, when a Triceps method detects an error, it has two ways of reporting it:

1. Set the error code in the special variable `$!` and return an `undef`. This is the “traditional” approach that is gradually replaced by the other one.

2. Set the error code in the special variable `$@` (and also `$!`, for compatibility with the other way) and tell the Perl interpreter to die. This is the “new” approach.

Currently most of the methods inplemented in C++ through XS return the errors in the first way but some have been converted to the new second way. Changing all the code at once is a lot of work, so the code is being converted to the new way gradually. The methods implemented in Perl use the second way. The methods using the second way are marked in their documentation as such. I hesitate to give a whole list of them, because the list changes as more methods get converted to the new way of error handling.

When you use the methods of the first group, you need to check their return code. The common idiom is:

```
my $table = $unit->makeTable($tabType, "EM_CALL", $name)
  or confess "$!";
```

It checks the return value and dies with an error message. `confess` is a nicer form of `die` and comes from the module `Carp`. `Carp` is a standard part of modern Perl, so all you need to do is just say

```
use Carp;
```

No need to download and install anything.

The problem with the simple `die` is that it reports an error but prints only the location where it has been called, which may be ten layers deep inside a library, not a full stack trace. The functions in `Carp` fix that. `confess` is the most interesting one of them. It works just like `die` but prints the whole stack trace.

The full description of Carp is available at http://perldoc.perl.org/Carp.html. It has more functions, however I find the full stack trace the most helpful thing in any case.

The nice part abour writing `or confess` is that the error message is fully controlled. If the error happens in a template, the template can produce a smarter message, telling what was wrong with it arguments on the high level, for example:

```
my $table = $unit->makeTable($tabType, "EM_CALL", $name)
  or confess "Query2: bad table type, table creation failed: $!";
```

But the problem is that you need to not forget writing `or confess` after every call. Also, it's inconvenient when the result of one call is passed directly to another one, like:

```
$self->{unit}->call(
  $self->{resLabel}->makeRowop("OP_INSERT", $rh->getRow()));
```

Here `call()` is a method with the second way of the error reporting, and it doesn't need `or confess`. But `makeRowop()` and `getRow()` use the first way. Writing `or confess` for each of them would work but would be quite tedious:

```
$self->{unit}->call(
  $self->{resLabel}->makeRowop(
    "OP_INSERT", $rh->getRow() or confess "$!"
  ) or confess "$!"
);
```

Because of this, the error checking in such nested calls usually gets skipped and the error manifests itself by the enveloping call dying with the XS error of bad argument type. Which doesn't print the stack trace and loses the information about the original error.

The “new” second way of error reporting is much better in this regard. It reports the error where and when it happens. It executes `confess` directly, which includes the stack trace into the error message and then proceeds like classic `die`. This includes the code implemented both in Perl and in C++ XS. With some exceptions though: the errors detected by the code auto-generated in XS, such as bad call arguments, still report through the plain `die`.

There are modules to make all the cases of `die` work like `confess`, `Devel::SimpleTrace` and `Carp::Always`. They work by intercepting the pseudo-signals `__WARN__` and `__DIE__`. The logic of `Carp::Always` is pretty simple, see http://cpansearch.perl.org/src/FERREIRA/Carp-Always-0.11/lib/Carp/Always.pm, so if you're not feeling like installing the module, you can easily do the same directly in your code.

If you want to intercept the error to add more information to the message, use `eval`:

```
eval { $self->{unit}->call($rowop) } or confess "Bad rowop argument:\n$@";
```

I have some better ideas about reporting the errors in the nested templated but they need to be implemented and tried out yet.

When the Perl code inside a label or tracer or aggregator or index sorting handler dies, the C++ infrastructure around it catches the error. Here Triceps also has the old and the new way of dealing with it. The old way just prints the error on stderr and continues like nothing has happened. It's still used for the tracers, aggregators and index sorting. The new way unrolls the stack trace through the C++ code and passes the `die` request to the Perl code that called it. The labels have been converted to this new way. When one Perl label calls another Perl label that calls the third Perl label, the call sequence goes in layers of Perl—C++—Perl—C++—Perl—C++—Perl. If that last label has its Perl code die and there are no `eval`s in between, the stack will be correctly unwound back through all these layers and reported in the error message. The C++ code will include the reports of all the chained label calls as well. If one of the intermediate Perl layers wraps the call in `eval`, it will receive the error message with the stack trace up to that point.

4.3. Memory management fundamentals

The memory is managed in Triceps using the reference counters. Each Triceps object has a reference counter in it. In C++ this is done explicitly, in Perl it gets mostly hidden behind the Perl memory management that also uses the reference counters. Mostly.

In C++ the Autoref template is used to produce the reference objects. As the references are copied around between these objects, the reference counts in the target objects are automatically adjusted. When the reference count drops to 0, the target object gets destroyed. While there are live references, the object can't get destroyed from under them. All nice and well and simple, however still possible to get wrong.

The major problem with the reference counters is the reference cycles. If object A has a reference to object B, and object B has a reference (possibly, indirect) to object A, then neither of them will ever be destroyed. Many of these cases can be resolved by keeping a reference in one direction and a plain pointer in the other. This of course introduces the problem of hanging pointers, so extra care has to be taken to not reference them. There also are the unpleasant situations when there is absolutely no way around the reference cycles. For example, the Triceps label's method may keep a reference to the next label, where to send its processed results. If the labels are connected into a loop (a perfectly normal occurrence), this would cause a reference cycle. Here the way around is to know when all the labels are no longer used (before the thread exit), and explicitly tell them to clear their references to the other labels. This breaks up the cycle, and then bits and pieces can be collected by the reference count logic.

The reference cycle problem can be seen all the way up into the Perl level. However Triceps provides the ready solutions for its typical occurences. To explain it, more about Triceps operation has to be explained first, so it's described in detail later in Chapter 8: “*Memory Management*” (p. 69) .

The reference counting may be single-threaded or multi-threaded. If an object may only be used inside one thread, the references to it use the faster single-threaded counting. In C++ it's real important to not access and not reference the single-threaded objects from multiple threads. In Perl, when a new thread is created, only the multithreaded objects from the parent thread become accessible for it, the rest become undefined, so the issue gets handled automatically (as of version 1.0 even the potentially multithreaded objects are still exported to Perl as single-threaded, with no connection between threads yet).

The C++ objects are exported into Perl through wrappers. The wrappers perform the adaptation between Perl reference counting and Triceps reference counting, and sometimes more of the helper functions. Perl sees them as blessed objects, from which you can inherit and otherwise treat like normal objects.

When we say that a Perl variable `$label` contains a Triceps label object, it really means that it contains a *referece* to a label object. When it gets copied like `$label2 = $label`, this copies the reference and now both variables refer to the same label object (more exactly, even to the same wrapper object). Any changes to the object's state done through one reference will also be visible through the other reference.

When the Perl references are copied between the variables, this increases the Perl reference count to the same wrapper object. However if an object goes into the C++ land, and then is extracted back (such as, create a Rowop from a Row, and then extract the Row from that Rowop), a brand new wrapper gets created. It's the same underlying C++ object but with multiple wrappers. You can't tell that it's the same object by comparing the Perl references, because they may be pointing to the different wrappers. However Triceps provides the method `same()` that compares the data inside the wrappers. It can be used as

```
$row1->same($row2)
```

and if it returns true, then both `$row1` and `$row2` point to the same underlying row.

Note also that if you inherit from the Triceps objects and add some extra data to them, none of that data nor even your derived class'es identity will be preserved when a new wrapper is created from the underlying C++ object.

4.4. Triceps constants

Triceps has a number of symbolic constants that are grouped into essentially enums. The constants themselves will be introduced with the classes that use them, but here is the general description common to them all.

In Perl they all are placed into the same namespace. Each group of constants (that can be thought of as an enum) gets its name prefix. For example, the operation codes are all prefixed with OP_, the enqueueing modes with EM_, and so on.

The underlying constants are all integer. The way to give symbolic names to constants in Perl is to define a function without arguments that would return the value. Each constant has such a function defined for it. For example, the opcode for the “insert” operation is the result of function `Triceps::OP_INSERT`.

Most methods that take constants as arguments are also smart enough to recognise the constant names as strings, and automatically convert them to integers. For example, the following calls are equivalent:

```
$label->makeRowop(&Triceps::OP_INSERT, ...);
$label->makeRowop("OP_INSERT", ...);
```

For a while I've thought that the version with `Triceps::OP_INSERT` would be more efficient and might check for correctness of the name at compile time. But as it turns out, no, on both counts. The look-up of the function by name happens at run time, so there is no compile-time check. And that look-up happens to be a little slower than the one done by the Triceps C++ code, so there is no win there either. The string version is not only shorter but also more efficient. The only win with the function is if you call it once, remember the result in a variable and then reuse. Unless you're chasing the last few percent of performance in a tight loop, it's not worth the trouble. Perhaps in the future the functions will be replaced with the module-level variables: *that* would be both faster and allow the compile-time checking with `use strict`.

What if you need to print out a constant in a message? Triceps provides the conversion functions for each group of constants. They generally are named `Triceps::`*`something`*`String`. For example,

```
print &Triceps::opcodeString(&Triceps::OP_INSERT);
```

would print “OP_INSERT” If the argument is out of range of the valid enums, it would return `undef` (but not set any error message in `$!`, since it's not a fatal error).

There also are functions to convert from strings to constant values. They generally are named `Triceps::string`*`Something`*. For example,

```
&Triceps::stringOpcode("OP_INSERT")
```

would return the integer value of `Triceps::OP_INSERT`. If the string name is not valid for this kind of constants, it would also return `undef`.

4.5. Printing the object contents

When debugging the programs, it's important to find from the error messages, what is going on, what kinds of objects are getting involved. Because of this, many of the Triceps objects provide a way to print out their contents into a string. This is done with the method `print()`. The simplest use is as follows:

```
$message = "Error in object " . $object->print();
```

Most of the objects tend to have a pretty complicated internal structure and are printed on multiple lines. They look better when the components are appropriately indented. The default call prints as if the basic message is un-indented, and indents every extra level by 2 spaces.

This can be changed with extra arguments. The general format of `print()` is:

```
$object->print([$indent, [$subindent] ])
```

where *$indent* is the initial indentation, and *$subindent* is the additional indentation for every level. The default `print()` is equivalent to `print("", "  ")`.

A special case is

```
$object->print(undef)
```

It prints the object in a single line, without line breaks.

Here is an example of how a row type object would get printed. The details of the row types will be described later, for now just assume that a row type is defined as:

```
$rt1 = Triceps::RowType->new(
  a => "uint8",
  b => "int32",
  c => "int64",
  d => "float64",
  e => "string",
);
```

Then `$rt1->print()` produces:

```
row {
  uint8 a,
  int32 b,
  int64 c,
  float64 d,
  string e,
}
```

With extra arguments `$rt1->print("++", "--")`:

```
row {
++--uint8 a,
++--int32 b,
++--int64 c,
++--float64 d,
++--string e,
++}
```

The first line doesn't have a “++” because the assumption is that the text gets appended to some other text already on this line, so any prefixes are used only for the following lines.

And finally with an undef argument `$rt1->print(undef)`:

```
row { uint8 a, int32 b, int64 c, float64 d, string e, }
```

The Rows and Rowops do not have the `print()` method. That's largely because the C++ code does not deal with printing the actual data, this is left to the Perl code. So instead they have the method `printP()` that does a similar job. Only it's simpler and doesn't have any of the indenting niceties. It always prints the data in a single line. The “P” in “printP” stands for “Perl”. The name is also different because of this lack of indenting niceties. See more about it in the Section 5.4: “Rows” (p. 28) .

4.6. The Hungarian notation

The Hungarian notation is the idea that the name of each variable should be prefixed with some abbreviation of its type. It has probably become most widely known from the Microsoft operating systems.

Overall it's a complete abomination and brain damage. But I'm using it widely in the examples in this manual. Why? The problem is that there usually too many components for one logical purpose. For a table, there would be a row type, a table type, and the table itself. Rather than inventing separate names for them, it's easier to have a common name and an uniform prefix. Eventually something better would have to be done but for now I've fallen back on the Hungarian notation. One possibility is to just not give names to the intermediate entities. Say just have a named table, and then there would be the the type of the table and the row type of the table.

Among the CEP systems, Triceps is not unique in the Hungarian notation department. Coral8/Sybase CCL has this mess of lots of schemas, input streams, windows and output streams, with the same naming problems. The uniform naming prefixes or suffixes help making this mess more navigable. I haven't actually used StreamBase but from reading the documentation I get the feeling that the Hungarian notation is probably useful for its SQL as well.

Chapter 5. Rows

In Triceps the relational data is stored and passed around as rows (once in a while I call them records, which is the same thing here). Each row belongs to a certain type, that defines the types of the fields. Each field may belong to one of the simple types.

5.1. Simple types

The simple values in Triceps belong to one of the simple types:

- uint8
- int32
- int64
- float64
- string

I like the explicit specification of the data size, so it's not some mysterious “double” but an explicit “float64”.

When the data is stored in the rows, it's stored in the strongly-typed binary format. When it's extracted from the rows for the Perl code to access, it gets converted into the Perl values. And the other way around, when stored into the rows, the conversion is done from the Perl values.

`uint8` is the type intended to represent the raw bytes. So, for example, when they are compared, they should be compared as raw bytes, not according to the locale. Since Perl stores the raw bytes in strings, and its `pack()` and `unpack()` functions operate on strings, The Perl side of Triceps extracts the uint8 values from records into Perl strings, and the other way around.

The string type is intended to represent a text string in whatever current locale (at some point it may become always UTF-8, this question is open for now).

Perl on the 32-bit machines has an issue with int64: it has no type to represent it directly. Because of that, when the int64 values are passed to Perl on the 32-bit machines, they are converted into the floating-point numbers. This gives only 54 bits (including sign) of precision, but that's close enough. Anyway, the 32-bit machines are obsolete by now, and Triceps it targeted towards the 64-bit machines.

On the 64-bit machines both int32 and int64 translate to the Perl 64-bit integers.

Note that there is no special type for timestamps. As of version 1.0 there is no time-based processing inside Triceps, but that does not prevent you from passing around timestamps as data and use them in your logic. Just store the timestamps as integers (or, if you prefer, as floating point numbers). When the time-based processing will be added to Perl, the plan is to still use the int64 to store the number of microseconds since the Unix epoch. My experience with the time types in the other CEP systems is that they cause nothing but confusion. In the meantime, the time-based processing is still possible by driving the notion of time explicitly. It's described in the Chapter 13: “*Time processing*” (p. 211) .

5.2. Row types

A row type is created from a sequence of (field-name, field-type) string pairs, for example:

```
$rt1 = Triceps::RowType->new(
  a => "uint8",
  b => "int32",
  c => "int64",
  d => "float64",
  e => "string",
);
```

Even though the pairs look like a hash, don't use an actual hash to create row types! The order of pairs in a hash is unpredictable, while the order of fields in a row type usually matters.

In an actual row the field may have a value or be NULL. The NULLs are represented in Perl as `undef`.

The real-world records tend to be pretty wide and contain repetitive data. Hundreds of fields are not unusual, and I know of a case when an Aleri customer wanted to have records of two thousand fields (and succeeded). This just begs for arrays. So the Triceps rows allow the array fields. They are specified by adding “[]” at the end of field type. The arrays may only be made up of fixed-width data, so no arrays of strings.

```
$rt2 = Triceps::RowType->new(
  a => "uint8[]",
  b => "int32[]",
  c => "int64[]",
  d => "float64[]",
  e => "string", # no arrays of strings!
) or confess "$!";
```

The arrays are of variable length, whatever array data passed when a row is created determines its length. The individual elements in the array may not be NULL (and if `undef`s are passed in the array used to construct the row, they will be replaced with 0s). The whole array field may be NULL, and this situation is equivalent to an empty array.

The type uint8 is typically used in arrays, “uint8[]” is the Triceps way to define a blob field. In Perl the “uint8[]” is represented as a string value, same as a simple “unit8”.

The rest of array values are represented in Perl as references to Perl arrays, containing the actual values.

The row type objects provide a way for introspection:

```
$rt->getdef()
```

returns back the array of pairs used to create this type. It can be used among other things for the schema inheritance. For example, the multi-part messages with daily unique ids can be defined as:

```
$rtMsgKey = Triceps::RowType->new(
  date => "string",
  id => "int32",
) or confess "$!";

$rtMsg = Triceps::RowType->new(
  $rtMsgKey->getdef(),
  from => "string",
  to => "string",
  subject => "string",
) or confess "$!";

$rtMsgPart = Triceps::RowType->new(
  $rtMsgKey->getdef(),
  type => "string",
  payload => "string",
) or confess "$!";
```

The meaning here is the same as in the CCL example:

```
create schema rtMsgKey (
  string date,
  integer id
);
create schema rtMsg inherits from rtMsgKey (
  string from,
  string to,
  string subject
);
create schema rtMsgPart inherits from rtMsgKey (
  string type,
  string payload
);
```

The grand plan is to provide some better ways of defining the commonality of fields between row types. It should include the ability to rename fields, to avoid conflicts, and to remember this equivalence to be reused in the further joins without the need to write it over and over again. But it has not come to the implementation stage yet.

The other methods are:

```
$rt->getFieldNames()
```

returns the array of field names only.

```
$rt->getFieldTypes()
```

returns the array of field types only.

```
$rt->getFieldMapping()
```

returns the array of pairs that map the field names to their indexes in the field definitions. It can be stored into a hash and used for name-to-index translation. It's used mostly in the templates, to generate code that accesses data in the rows by field index (which is more efficient than access by name). For example, for `rtMsgKey` defined above it would return `(date => 0, id => 1)`.

5.3. Row types equivalence

The Triceps objects are usually strongly typed. A label handles rows of a certain type. A table stores rows of a certain type.

However there may be multiple ways to check whether a row fits for a certain type:

- It may be a row of the exact same type, created with the same RowType object.
- It may be a row of another type but one with the exact same definition.
- It may be a row of another type that has the same number of fields and field types but different field names. The field names (and everything else in Triceps) are case-sensitive.

The row types may be compared for these conditions using the methods:

```
$rt1->same($rt2)
$rt1->equals($rt2)
$rt1->match($rt2)
```

The comparisons are hierarchical: if two type references are the same, they would also be equal and matching; two equal types are also matching.

Most of the objects would accept the rows of any matching type (this may change or become adjustable in the future). However if the rows are not of the same type, this check involves a performance penalty. If the types are the same, the comparison is limited to comparing the pointers. But if not, then the whole type definition has to be compared. So every time a row of a different type is passed, it would involve the overhead of type comparison.

For example:

```
my @schema = (
  a => "int32",
  b => "string"
);

my $rt1 = Triceps::RowType->new(@schema) or confess "$!";
# $rt2 is equal to $rt1: same field names and field types
my $rt2 = Triceps::RowType->new(@schema) or confess "$!";
# $rt3  matches $rt1 and $rt2: same field types but different names
my $rt3 = Triceps::RowType->new(
  A => "int32",
  B => "string"
) or confess "$!";

my $lab = $unit->makeDummyLabel($rt1, "lab") or confess "$!";
# same type, efficient
my $rop1 = $lab->makeRowop(&Triceps::OP_INSERT,
  $rt1->makeRowArray(1, "x")) or confess "$!";
# different row type, involves a comparison overhead
my $rop2 = $lab->makeRowop(&Triceps::OP_INSERT,
  $rt2->makeRowArray(1, "x")) or confess "$!";
# different row type, involves a comparison overhead
my $rop3 = $lab->makeRowop(&Triceps::OP_INSERT,
  $rt3->makeRowArray(1, "x")) or confess "$!";
```

A dummy label used here is a label that does nothing (its usefulness will be explained later).

Once the Rowop is constructed, no further penalty is involved: the row in the Rowop is re-typed to the type of the label from now on. It's physically still the same row with another reference to it, but when you get it back from the Rowop, it will have the label's type. It's all a part of the interesting interaction between C++ and Perl. All the type checking is done in the Perl XS layer. The C++ code just expects that the data is always right and doesn't carry the types around. When the Perl code wants to get the row back from the Rowop, it wants to know the type of the row. The only way to get it is to look, what is the label of this Rowop, and get the row type from the label. This is also the reason why the types have to be checked when the Rowop is constructed: if a wrong row is placed into the Rowop, there will be no later opportunity to check it for correctness, and bad data may cause a crash.

5.4. Rows

The rows in Triceps always belong to some row type, and are always immutable. Once a row is created, it can not be changed. This allows it to be referenced from multiple places, instead of copying the whole row value. Naturally, a row may be passed and shared between multiple threads.

The row type provides the constructor methods for the rows:

```
$row = $rowType->makeRowArray(@fieldValues);
$row = $rowType->makeRowHash($fieldName => $fieldValue, ...);
```

Here `$row` is a reference to the resulting row. As usual, in case of error it will be left as `undef`, with the error message in `$!`.

In the array form, the values for the fields go in the same order as they are specified in the row type (if there are too few values, the rest will be considered NULL, having too many values is an error).

The Perl value of `undef` is treated as NULL.

In the hash form, the fields are specified as name-value pairs. If the same field is specified multiple times, the last value will overwrite all the previous ones. The unspecified fields will be left as NULL. Again, the arguments of the function actually are an array, but if you pass a hash, its contents will be converted to an array on the call stack.

If the performance is important, the array form is more efficient, since the hash form has to translate internally the field names to indexes.

The row itself and its type don't have any concept of keys in general and of the primary key in particular. So any fields may be left as NULL. There is no “NOT NULL” constraint.

Some examples:

```
$row  = $rowType->makeRowArray(@fields) or die "$!";
$row  = $rowType->makeRowArray($a, $b, $c) or die "$!";
$row  = $rowType->makeRowHash(%fields) or die "$!";
$row  = $rowType->makeRowHash(a => $a, b => $b) or die "$!";
```

The usual Perl conversions are applied to the values. So for example, if you pass an integer 1 for a string field, it will be converted to the string “1”. Or if you pass a string “” for an integer field, it will be converted to 0.

If a field is an array (as always, except for `uint8[]` which is represented as a Perl string), its value is a Perl array reference (or `undef`). For example:

```
$rt1 = Triceps::RowType->new(
  a => "uint8[]",
  b => "int32[]",
) or confess "$!";
$row = $rt1->makeRowArray("abcd", [1, 2, 3]) or confess "$!";
```

An empty array will become a NULL value. So the following two are equivalent:

```
$row = $rt1->makeRowArray("abcd", []) or die "$!";
$row = $rt1->makeRowArray("abcd", undef) or die "$!";
```

Remember that an array field may not contain NULL values. Any `undef`s in the array fields will be silently converted to zeroes (since arrays are supported only for the numeric types, a zero value would always be available for all of them). The following two are equivalent:

```
$row = $rt1->makeRowArray("abcd", [undef, undef]) or die "$!";
$row = $rt1->makeRowArray("abcd", [0, 0]) or die "$!";
```

The row also provides a way to copy itself, modifying the values of selected fields:

```
$row2 = $row1->copymod($fieldName => $fieldValue, ...);
```

The fields that are not explicitly specified will be left unchanged. Since the rows are immutable, this is the closest thing to the field assignment. `copymod()` is generally more efficient than extracting the row into an array or hash, replacing a few of them with new values and constructing a new row. It bypasses the binary-to-Perl-to-binary conversions for the unchanged fields.

The row knows its type, which can be obtained with

```
$row->getType()
```

Note that this will create a new Perl wrapper to the underlying type object. So if you do:

```
$rt1 = ...;
$row = $rt1->makeRow...;
$rt2 = $row->getType();
```

then `$rt1` will not be equal to `$rt2` by the direct Perl comparison (`$rt1 != $rt2`). However both `$rt1` and `$rt2` will refer to the same row type object, so `$rt1-&gt;same($rt2)` will be true.

The row references can also be compared for sameness:

```
$row1->same($row2)
```

The row contents can be extracted back into Perl representation as

```
@adata = $row->toArray();
%hdata = $row->toHash();
```

Again, the NULL fields will become `undef`s, and the array fields (unless they are NULL) will become Perl array references. Since the empty array fields are equivalent to NULL array fields, on extraction back they will be treated the same as NULL fields, and become `undef`s.

There is also a convenience function to get one field from a row at a time by name:

```
$value = $row->get("fieldName");
```

If you need to access only a few fields from a big row, `get()` is more efficient (and easier to write) that extracting the whole row with `toHash()` or even with `toArray()`. But don't forget that every time you call `get()`, it creates a new Perl value, which may be pretty involved if the value is an array. So the most efficient way then for the values that get reused many times is to call `get()`, remember the result in a Perl variable, and then reuse that variable.

There is also a way to conveniently print a rows contents, usually for the debugging purposes:

```
$result = $row->printP();
```

The name `printP` is an artifact of implementation: it shows that this method is implemented in Perl and uses the default Perl conversions of values to strings. The `uint8[]` arrays are printed directly as strings. The result is a sequence of `name="value"` or `name=["value", "value", "value"]` for all the non-NULL fields. The backslashes and double quotes inside the values are escaped by backslashes in Perl style. For example, reusing the row type above,

```
$row = $rt1->makeRowArray('ab\ "cd"', [0, 0]) or die "$!";
print $row->printP(), "\n";
```

will produce

```
a="ab\\ \"cd\"" b=["0", "0"]
```

Finally, there is a deep debugging method:

```
$result = $row->hexdump()
```

That dumps the raw bytes of the row's binary format, and is useful only to debug the more weird issues.

Chapter 6. Labels and Row Operations

6.1. Labels basics

In each CEP engine there are two kinds of logic: One is to get some request, look up some state, maybe update some state, and return the result. The other has to do with the maintenance of the state: make sure that when one part of the state is changed, the change propagates consistently through the rest of it. If we take a common RDBMS for an analog, the first kind would be like the ad-hoc queries, the second kind will be like the triggers. The CEP engines are very much like database engines driven by triggers, so the second kind tends to account for a lot of code.

The first kind of logic is often very nicely accommodated by the procedural logic. The second kind often (but not always) can benefit from a more relational, SQLy definition. However the SQLy definitions don't stay SQLy for long. When every every SQL statement executes, it gets compiled first into the procedural form, and only then executes as the procedural code.

The Triceps approach is tilted toward the procedural execution. That is, the procedural definitions come out of the box, and then the high-level relational logic can be defined on top of them with the templates and code generators.

These bits of code, especially where the first and second kind connect, need some way to pass the data and operations between them. In Triceps these connection points are called Labels.

The streaming data rows enter the procedural logic through a label. Each row causes one call on the label. From the functional standpoint they are the same as Coral8 Streams, as has been shown in Section 1.4: “We're not in 1950s any more, or are we?” (p. 3) . Except that in Triceps the labels receive not just rows but operations on rows, as in Aleri: a combination of a row and an operation code.

They are named “labels” because Triceps has been built around the more procedural ideas, and when looked at from that side, the labels are targets of calls and GOTOs.

If the streaming model is defined as a data flow graph, each arrow in the graph is essentially a GOTO operation, and each node is a label.

A Triceps label is not quite a GOTO label, since the actual procedural control always returns back after executing the label's code. It can be thought of as a label of a function or procedure. But if the caller does nothing but immedially return after getting the control back, it works very much like a GOTO label.

Each label accepts operations on rows of a certain type.

Each label belongs to a certain execution unit, so a label can be used only strictly inside one thread and can not be shared between threads.

Each label may have some code to execute when it receives a row operation. The labels without code can be useful too.

A Triceps model contains the straightforward code and the mode complex stateful elements, such as tables, aggregators, joiners (which may be implemented in C++ or in Perl, or created as user templates). These stateful elements would have some input labels, where the actions may be sent to them (and the actions may also be done as direct method calls), and output labels, where they would produce the indications of the changed state and/or responses to the queries. This is shown in the diagram in Figure 6.1 . The output labels are typically the ones without code (“dummy labels”). They do nothing by themselves, but can pass the data to the other labels. This passing of data is achieved by *chaining* the labels: when a label is called, it will first execute its own code (if it has any), and then call the same operation on whatever labels are chained from it. Which may have more labels chained from them in turn. So, to pass the data, chain the input label of the following element to the output label of the previous element.

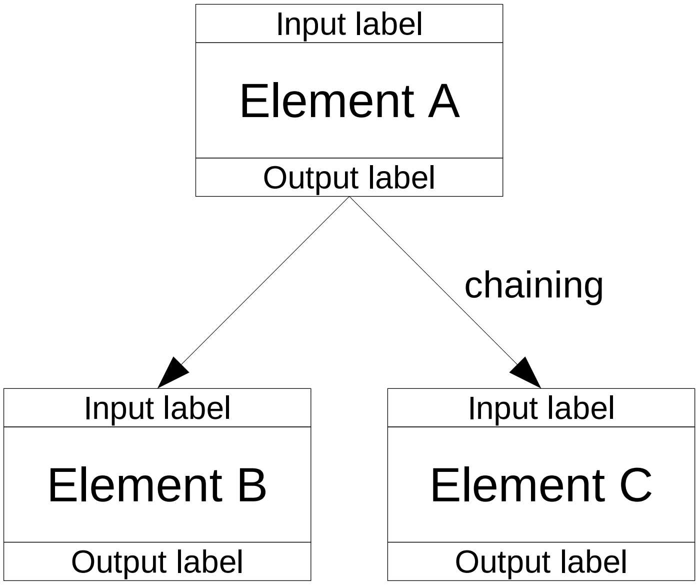

Figure 6.1. Stateful elements with chained labels.

The make things clear, a label doesn't have to be a part of a stateful element. The labels absolutely can exist by themselves. It's just that the stateful elements can use the labels as their endpoints.

6.2. Label construction

The execution unit provides methods to construct labels. A dummy label is constructed as:

```
$label = $unit->makeDummyLabel($rowType, "name") or confess "$!";
```

It takes as arguments the type of rows that the label will accept and the symbolic name of the label. As usual, the name can be any but for the ease of debugging it's better to give the same name as the label variable.

The label with Perl code is constructed as follows:

```
$label = $unit->makeLabel($rowType, "name", \&clearSub,
  \&execSub, @args);
```

The row type and name arguments are the same as for the dummy label. The following arguments provide the references to the Perl functions that perform the actions. `execSub` is the function that executes to handle the incoming rows. It gets the arguments:

```
execSub($label, $rowop, @args)
```

Here `$label` is this label, `$rowop` is the row operation, and `@args` are the same as extra arguments specified at the label creation.

The row operation actually contains the label reference, so why pass it the second time? The reason lies in the chaining. The current label may be chained, possibly through multiple levels, to some original label, and the rowop will refer to that original label. The extra argument lets the code find the current label.

`clearSub` is the function that clears the label. It will be explained in the Section 8.2: “Clearing of the labels” (p. 70) Either of `execSub` and `clearSub` can be specified as `undef`. Though a label with an undefined `execSub` makes the label useless for anything other than clearing. On an attempt to send data to it, it will complain that the label has been cleared. The undefined `clearSub` causes the function `Triceps::clearArgs()` to be used as the default, which provides the correct reaction for most situations.

There is a special convenience constructor for the labels that are used only for clearing an object (their usefulness is discussed in Section 8.2: “Clearing of the labels” (p. 70)).

```
$lb = $unit->makeClearingLabel("name", @args);
```

The arguments would be the references to the objects that need clearing, usually the object's `$self`. They will be cleared with `Triceps::clearArgs()` when the label clearing gets called.

6.3. Other label methods

The chaining of labels is done with the call:

```
$label1->chain($label2) or confess "$!";
```

`$label2` becomes chained to `$label1`. A label can not be chained to itself, neither directly nor through other intermediate labels. The row types of the chained labels must be equal (this is more strict than for queueing up the row operations for labels, and might change one or the other way in the future).

A label's chainings can be cleared with

```
$label1->clearChained();
```

It returns nothing, and clears the chainings from this label. There is no way to unchain only some selected labels.

To check if there are any labels chained from this one, use:

```
$result = $label->hasChained();
```

The same check can be done with

```
@chain = $label->getChain();
```

```
if ($#chain >= 0) { ... }
```

but `hasChained()` is more efficient since it doesn't have to construct that intermediate array.

The whole label can be cleared with

```
$label->clear();
```

This is fully equivalent to what happens when an execution unit clears the labels: it calls the clear function (if any) and clears the chainings. Note that the labels that used to be chained from this one do not get cleared themselves, they're only unchained from this one.

Labels have the usual way of comparing the references:

```
$label1->same($label2)
```

returns true if both references point to the same label object.

The labels introspection can be done with the methods:

```
$rowType = $label->getType();
$rowType = $label->getRowType();
$unit = $label->getUnit();
$name = $label->getName();
```

```
@chainedLabels = $label->getChain();
$execSubRef = $label->getCode();
```

The methods `getType()` and `getRowType()` are the same, they both return the row type of the label. `getType()` is shorter, which looked convenient for a while, but `getRowType()` has the name consistent with the rest of the classes. This consistency comes useful when passing the objects of various types to the same methods, using the Perl's name-based polymorphism. For now both of them are present, but `getType()` will likely be deprecated in the future.

If the label has been cleared, `getUnit()` will return an `undef`. `getChain()` returns an array of references to the chained labels. `getCode()` is actually half-done because it returns just the Perl function reference to the execution handler but not its arguments, nor reference to the clearing function. It will be changed in the future to fix these issues. `getCode()` is not applicable to the dummy labels, and would return an `undef` for them.

The labels actually exist in multiple varieties. The underlying common denominator is the C++ class Label. This class may be extended and the resulting labels embedded into the C++ objects. These labels can be accesses and controlled from Perl but their logic is hardcoded in their objects and is not directly visible from Perl. The dummy labels are a subclass of labels in general, and can be constructed directly from Perl. Another subclass is the labels with the Perl handlers. They can be constructed from Perl, and really only from Perl. The C++ code can access and control them, in a symmetrical relation. The method `getCode()` has meaning only on these Perl labels. Finally, the clearing labels also get dreated from Perl, and fundamentally are Perl labels with many settings hardcoded in the constructor. `getCode()` can be used on them too but since they have no handler code, it would always return `undef`.

There is also a way to change a label's name:

```
$label->setName($name);
```

It returns nothing, and there is probably no reason to call it. It will likely be removed in the future.

The label also provides the constructor methods for the row operations, which are described below.

6.4. Row operations

A row operation (also known as rowop) in Triceps is an unit of work for a label. It's always destined for a particular label (which could also pass the rowop to its chained labels), and has a row to process and an opcode. The opcodes will be described momentarily in the Section 6.5: “Opcodes” (p. 36) .

A row operation is constructed as:

```
$rowop = $label->makeRowop($opcode, $row);
```

The opcode may be specified an integer or as a string. Historically, there is also an optional extra argument for the enqueuing mode but it's already obsolete, so I don't show it here.

Since the labels are single-threaded, the rowops are single-threaded too. The rowops are immutable, just as the rows are. It's possible to keep a rowop around and call it over and over again.

A rowop can be created from a bunch of fields in an array or hash form in two steps:

```
$rowop = $label->makeRowop($opcode, $rt->makeRowHash(
  $fieldName => $fieldValue, ...));
$rowop = $label->makeRowop($opcode, $rt->makeRowArray(@fields));
```

Since this kind of creation happens fairly often, writing out these calls every time becomes tedious. The Label provides the combined constructors to make life easier:

```
$rowop = $label->makeRowopHash($opcode, $fieldName => $fieldValue, ...);
$rowop = $label->makeRowopArray($opcode, @fields);
```

Note that they don't need the row type argument any more, because the label knows the row type and provides it. Internally these methods are currently implemented in Perl, and just wrap the two calls into one. They also use the new error handling convention, and confess on any errors. There is normally no need to check their result. In the future they will be rewritten in C++ for greater efficiency.

There also are the methods that create a rowop and immediately call it. They will be described with the execution unit.

A copy of rowop (not just another reference but an honest separate copied object) can be created with:

```
$rowop2 = $rowop1->copy();
```

However, since the rowops are immutable, a reference is just as good as a copy. This method is historic and will likely be removed or modified.

A more interesting operation is the rowop adoption: it is a way to pass the row and opcode from one rowop to another new one, with a different label.

```
$rowop2 = $label->adopt($rowop1);
```

It is very convenient for building the label handlers that pass the rowops to the other labels unchanged. For example, a label that filters the data and passes it to the next label, can be implemented as follows:

```
my $lab1 = $unit->makeLabel($rt1, "lab1", undef, sub {
  my ($label, $rowop) = @_;
  if ($rowop->getRow()->get("a") > 10) {
    $unit->call($lab2->adopt($rowop));
  }
}) or confess "$!";
```

This code doesn't even look at the opcode in the rowop, it just passes it through and lets the next label worry about it. The functionality of `adopt()` also can be implemented with

```
$rowop2 = $label->makeRowop($rowop1->getOpcode(), $rowop1->getRow());
```

But `adopt()` is easier to call and also more efficient, because less of the intermediate data surfaces from the C++ level to the Perl level.

The references to rowops can be compared as usual:

```
$rowop1->same($rowop2)
```

returns true if both point to the same rowop object.

The rowop data can be extracted back:

```
$label = $rowop->getLabel();
$opcode = $rowop->getOpcode();
$row = $rowop->getRow();
```

A Rowop can be printed (usually for debugging purposes) with

```
$string = $rowop->printP();
```

Just as with a row, the method `printP()` is implemented in Perl. In the future a `print()` done right in C++ may be added, but for now I try to keep all the interpretation of the data on the Perl side. Even though `printP()` is implemented in Perl, it can print the rowops for any kinds of labels. The following example gives an idea of the format in which the rowops get printed:

```
$lb = $unit->makeDummyLabel($rt, "lb");
$rowop = $lb->makeRowop(&Triceps::OP_INSERT, $row);
print $rowop->printP(), "\n";
```

would produce

```
lb OP_INSERT a="123" b="456" c="3000000000000000" d="3.14" e="text"
```

The row contents is printed through `Row::printP()`, so it has the same format.

6.5. Opcodes

The defined opcodes are:

- `&Triceps::OP_NOP` or `"OP_NOP"`
- `&Triceps::OP_INSERT` or `"OP_INSERT"`
- `&Triceps::OP_DELETE` or `"OP_DELETE"`

The meaning is straightforward: NOP does nothing, INSERT inserts a row, DELETE deletes a row. There is no opcode to replace or update a row. The updates are done as two separate operations: first DELETE the old value then INSERT the new value. The order is important: the old value has to be deleted before inserting the new one. But there is no requirement that these operations must go one after another. If you want to update ten rows, you can first delete all ten and then insert the new ten. In the normal processing the end result will be the same, even though it might go through some different intermediate states. It's a good idea to write your models to follow the same principle.

Internally an opcode is always represented as an integer constant. The same constant value can be obtained by calling the functions `&Triceps::OP_*`. However when constructing the rowops, you can also use the string literals `"OP_*"` with the same result, they will be automatically transtaled to the integers. In fact, the string literal form is slightly faster (unless you save the result of the function in a variable and then use the integer value from that variable for the repeated construction).

But when you get the opcodes back from rowops, they are always returned as integers. Triceps provides functions that convert the opcodes between the integer and string constants:

```
$opcode = &Triceps::stringOpcode($opcodeName);
$opcodeName = &Triceps::opcodeString($opcode);
```

They come handy for all kinds of print-outs. If you pass the invalid values, the conversion to integers will return an `undef`.

The conversion of the invalid integers to strings is more interesting. And by the way, you can pass the invalid integer opcodes to the rowop construction too, and they won't be caught. The way they will be processed is a bit of a lottery. The proper integer values are actually bitmasks, and they are nicely formatted to make sense. The invalid values would make some random bitmasks, and they will get processed in some unpredictable way. When converting an invalid integer to a string, `opcodeString` tries to predict and show this way in a set of letters `I` and `D` in square brackets, for INSERT and DELETE flags. If both are present, usually the INSERT flag wins over the DELETE in the processing. If none are present, it's a NOP.

In the normal processing you don't normally read the opcode and then compare it with different values. Instead you check the meaning of the opcode (that is internally a bitmask) directly with the rowop methods:

```
$rowop->isNop()
$rowop->isInsert()
$rowop->isDelete()
```

The typical idiom for the label's handler function is:

```
if ($rowop->isInsert()) {
  # handle the insert logic ...
} elsif($rowop->isDelete()) {
  # handle the delete logic...
}
```

The NOPs get silently ignored in this idiom, as they should be. Generally there is no point in creating the rowops with the `OP_NOP` opcode, unless you want to use them for some weird logic.

The main Triceps package also provides functions to check the integer opcode values directly:

```
Triceps::isNop($opcode)
Triceps::isInsert($opcode)
Triceps::isDelete($opcode)
```

The same-named methods of Rowop are just the more convenient and efficient way to say

```
Triceps::isNop($rowop->getOpcode())
Triceps::isInsert($rowop->getOpcode())
Triceps::isDelete($rowop->getOpcode())
```

They handle the whole logic directly in C++ without an extra Perl conversion of the values.

Chapter 7. Scheduling

7.1. Overview of the scheduling

The scheduling determines, in which order the row operations are processed. If there are multiple operations available, which one should be processed first? The scheduler keeps a queue of the operations and selects, which one to execute next. This has a major effect on the logic of a CEP model.

There are multiple approaches to scheduling. Aleri essentially doesn't have any, except for the flow control between threads, because each its element is a separate thread. Coral8 has an intricate scheduling algorithm. Sybase R5 has the same logic as Coral8 inside each thread. StreamBase presumably also has some.

The scheduling logic in Triceps is different from the other CEP systems. The Coral8 logic looks at first like the only reasonable way to go, but could not be used in Triceps for three reasons: First, it's a trade secret, so it can't be simply reused. If I'd never seen it, that would not be an issue but I've worked on it and implemented its version for R5. Second, it relies on the properties that the compiler computes from the model graph analysis. Triceps has no compiler, and could not do this. Third, in reality it simply doesn't work that well. There are quite a few cases when the Coral8 scheduler comes up with a strange and troublesome execution order.

For a while I've hoped that Triceps would need no scheduler at all, and everything would be handled by the procedural calls. This has proved to have its own limitations, and thus the labels and their scheduling were born. The Triceps scheduling still has issues to resolve, but overall it still feels much better than the Coral8 one.

7.2. No bundling

The most important principle of Triceps scheduling is: No Bundling. Every rowop is for itself. The bundling is what messes up the Coral8 scheduler the most.

What is a bundle? It's a set of records that go through the execution together. If you have a model consisting of two functional elements F1 and F2 connected in a sequential fashion

```
F1->F2
```

and a few loose records R1, R2, R3, the normal execution order without bundling will be:

```
F1(R1), F2(R1), F1(R2), F2(R2), F1(R3), F2(R3)
```

Each row goes through the whole model (a real simple one in this case) before the next one is touched. This allows F2 to take into accont the state of F1 exactly as it was right after processing the same record, without any interventions in between.

If the same records are placed in a bundle (R1, R2, R3), the execution order will be different:

```
F1(R1), F1(R2), F1(R3), F2(R1), F2(R2), F2(R3)
```

The whole bundle goes through F1 before the rows go to F2.

That would not be a problem, and even could be occasionally useful, if the bundles were always created explicitly. In the reality of Coral8, every time a statement produces multiple record from a single one (think of a join that picks multiple records from another side), it creates a bundle and messes up all the logic after it. Some logic gets affected so badly that a few statements in CCL (like ON UPDATE) had to be designated as always ignoring the bundles, otherwise they would not work at all. At DB I wrote a CCL pattern for breaking up the bundles. It's rather heavyweight and thus could not be used all over the place but provides a generic solution for the most unpleasant cases.

Worse yet, the bundles may get created in Coral8 absolutely accidentally: if two records happen to have the same timestamp, for all practical purposes they would act as a bundle. In the models that were designed without the appropriate guards, this leads to the time-based bugs that are hard to catch and debug. Writing these guards correctly is hard, and testing them is even harder.

Another issue with bundles is that they make the large queries slower. Suppose you do a query from a window that returns a million records. All of them will be collected in a bundle, then the bundle will be sent to the interface gateway that would build one huge protocol packet, which will then be sent to the client, which will receive the whole packet and then finally iterate on the records in it. Assuming that nothing runs out of memory along the way, it will be a long time until the client sees the first record. Very, very annoying.

Aleri also has its own version of bundles, called transactions, but a more smart one. Aleri always relies on the primary keys. The condition for a transaction is that it must never contain multiple modification for the same primary key. Since there are no execution order guarantees between the functional elements, in this respect the transactions work in the same way as loose records, only with a more efficient communication between threads. Still, if the primary key changes in an element (say, an aggregator), the condition does not propagate through it. Such elements have to internally collapse the outgoing transactions along the new key, adding overhead.

7.3. Basic scheduling in Triceps

In Triceps the scheduling is done by the execution unit, or simply “unit” as it's often referred to. It provides 3 basic ways of executing of a rowop:

Call:

> Execute the label right now, including all the nested calls. All of this will be completed after the call returns.

Fork:

> Execute the label after the current label returns but before anything else is done. Obviously, if multiple labels are forked, they will execute in order after the current label returns (but before its caller gets the control back). This method has looked promising at one point but has currently fallen out of favor and will likely be removed in the future.

Schedule:

> Execute the label after everything else is done.

This is kind of intuitively clear but the details might sometimes be a bit surprising. So let us look in detail at how it works inside on an example of a fairly convoluted scheduling sequence.

A scheduler in the execution unit keeps a stack of queues. Each queue is essentially a stack frame, so I'll be using the terms `queue` and `frame` interchangeably. The stack always contains at least one queue, which is called the outermost stack frame.

When the new rowops come from the outside world, they are added with `schedule()` to that stack frame. That's what `schedule()` does: always adds rowops to the outermost stack frame. If rowops 1, 2 and 3 are added, the stack looks like this (the brackets denote a stack frame):

```
[1, 2, 3]
```

The unit method `drainFrame()` is then used to run the scheduler and process the rowops. It makes the unit call each rowop on the innermost frame (which is initially the same as outermost frame, since there is only one frame) in order.

First it calls the rowop 1. It's removed from the queue, then a new frame is pushed onto the stack:

```
[ ] ~1
[2, 3]
```

This new frame is the rowop 1's frame, which is marked on the diagram by “~1”. The diagram shows the most recently pushed, innermost, frame on the top, and the oldest, outermost frame on the bottom. The concepts of “innermost” and “outermost” come from the nested calls: the most recent call is nested the deepest in the middle and is the innermost one.

Then the rowop 1 executes. If it calls rowop 4, another frame is pushed onto the stack for it:

```
[ ] ~4
[ ] ~1
[2, 3]
```

Then the rowop 4 executes. The rowop 4 never gets onto any of the queues. The call just pushes a new frame and executes the rowop right away. The identity of rowop being processed is kept in the call context. A call also involves a direct C++ call on the thread stack, and if any Perl code is involved, a Perl call too. Because of this, if you nest the calls too deeply, you may run out of the thread stack space and get it to crash.

After the rowop 4 is finished (not calling any other rowops), the innermost empty frame is popped before the execution of rowop 1 continues. The queue stack reverts to the previous state.

```
[ ] ~1
[2, 3]
```

Suppose then rowop 1 forks rowops 5 and 6. They are appended to the innermost frame in the order they are forked.

```
[5, 6] ~1
[2, 3]
```

If rowop 1 then calls rowop 7, again a frame is pushed onto the stack before it executes:

```
[ ] ~7
[5, 6] ~1
[2, 3]
```

The rowops 5 and 6 still don't execute, they keep sitting on the queue until the rowop 1 would return. After the call of rowop 7 completes, the scheduler stack returns to the previous state.

Suppose now the execution of rowop 1 completes. But its stack frame can not be popped yet, because it is not empty. The scheduler calls `drainFrame()` recursively, which picks the next rowop from the innermost queue (rowop 5), and calls it, pushing a new stack frame and executing the rowop 5 code:

```
[ ] ~5
[6] ~1*
[2, 3]
```

The former rowop 1's frame is now marked with “~1*” for the ease of tracking, even though it has completed.

If rowop 5 forks rowop 8, the stack becomes:

```
[8] ~5
[6] ~1*
[2, 3]
```

When the execution of rowop 5 returns, its queue is also not empty. So the scheduler starts draining the innermost frame again, and calls rowop 8. During its execution the stack is:

```
[ ] ~8
[ ] ~5*
[6] ~1*
[2, 3]
```

Suppose the rowop 8 doesn't call or fork anything else and returns. Its innermost queue is empty, so the call completes and pops the stack frame:

```
[ ] ~5*
[6] ~1*
[2, 3]
```

Now the queue of rowop 5 is also empty, so its draining completes and pops the drained frame:

```
[6] ~1*
[2, 3]
```

The draining of the rowop 1's frame continues by picking the rowop 6 from the queue and calling it:

```
[ ] ~6
[ ] ~1*
[2, 3]
```

Suppose rowop 6 calls `schedule()` of rowop 9. Rowop 9 is then added to the outermost queue:

```
[ ] ~6
[ ] ~1*
[2, 3, 9]
```

Rowop 6 then returns, its queue is empty, so it's popped and its call completes.

```
[ ] ~1*
[2, 3, 9]
```

Now the queue of rowop 1 has become empty, so it's popped from the stack and the call of rowop 1 completes:

```
[2, 3, 9]
```

The unit method `drainFrame()` keeps running on the outermost frame, now taking the rowop 2 and executing it, and so on, until the outermost queue becomes empty, and `drainFrame()` returns.

An interesting question is, what happens with the chained labels? Where do they fit in the order of execution? It turns out to be a bit of a mix between a `call()` and a `fork()`. They get checked after the original label completes its execution and has its frame drained but before that frame gets popped.

If any chained labels are found, they are called one by one. But they don't get a new frame created. They all reuse the frame left over from the parent label. The frame gets popped only after all the chained labels have completed.

7.4. Loop scheduling

The easiest and most efficient way to schedule the loops is to do it procedurally, something like this:

```
foreach my $row (@rowset) {
  $unit->call($lbA->makeRowop(&Triceps::OP_INSERT, $row));
}
```

However the labels topologically connected into a loop can come handy as well. Some logic may be easier to express this way. Suppose the model contains the labels connected in a loop, as in Figure 7.1 .

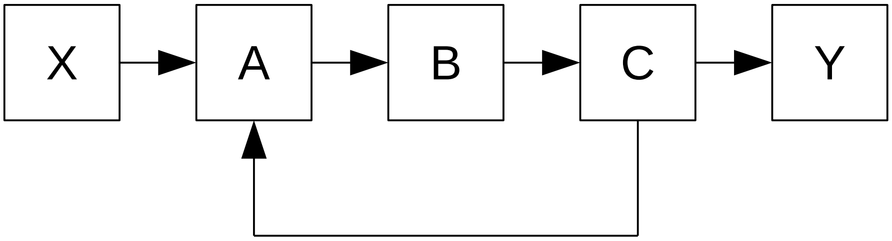

Figure 7.1. Labels forming a loop.

But if handled simple-mindedly, it can use a lot of stack space. Suppose some rowop X1 is scheduled for label X, and causes the loop to be executed twice, with rowops X1, A2, B3, C4, A5, B6, C7, Y8. If each operation is done as a `call()`, the stack grows like this: It starts with X1 scheduled.

```
[X1]
```

Which then gets executed, with its own execution frame (marked as such for clarity):

```
[ ] ~X1
[ ]
```

Which then calls A2:

```
[ ] ~A2
[ ] ~X1
[ ]
```

By the time the execution comes to Y8, the stack looks like this:

```
[ ] ~Y8
[ ] ~C7
[ ] ~B6
[ ] ~A5
[ ] ~C4
[ ] ~B3
[ ] ~A2
[ ] ~X1
[ ]
```

The loop has been converted into recursion, and the whole length of execution is the depth of the recursion. If the loop executes a million times, the stack will be three million levels deep. Worse yet, it's not just the Triceps scheduler stack that grows, it's also the process (C++) stack.

Which is why this kind of recursive calls are explicitly forbidden in Triceps. If you try to do it, on the first recursive call the execution will die with an error.

Would things be better with `fork()` instead of `call()` used throughout the loop? It starts the same way:

```
[X1]
```

Then X1 executes, gets its own frame and forks A2:

```
[A2] ~X1
[ ]
```

Then A2 executes, gets its own frame and forks B3:

```
[B3] ~A2
[ ] ~X1*
[ ]
```

Even though X1 has completed, its stack frame stays until all the rowops forked in it complete too. By the end of the loop the stack picture becomes exactly the same as with `call()`. For a while I've thought that optimizing out the empty stack frames would solve the problem, but no, that doesn't work: the problem is that the C++ process stack keeps growing no matter what. The jump back in the loop needs to be placed into an earlier stack frame to prevent the stack from growing.

One way to do it would be to use the `schedule()` operation in C to jump back to A, placing the rowop A5 back onto the outermost frame. The scheduler stack at the end of C4 would look like:

```
[ ] ~C4
[ ] ~B3
[ ] ~A2
[ ] ~X1
[A5]
```

Then the stack would unwind back to:

```
[A5]
```

And the next iteration of the loop will start afresh. The problem here is that if X1 wanted to complete the loop and then do something, it can't. By the time the second iteration of the loop starts, X1 is completely gone. It would be better to be able to enqueue the next execution of the loop at the specific point of the stack.

Here the concept of the frame mark comes in. A frame mark is a token object, completely opaque to the program. It can be used only in two operations:

- `setMark()` remembers the position in the frame stack, just outside the current frame.
- `loopAt()` enqueues a rowop at the marked frame.

Then the loop wold have its mark object M. The label A will execute `setMark(M)`, and the label C will execute `loopAt(M, rowop(A))`. The rest of the execution can as well use `call()`, as shown in Figure 7.1 .

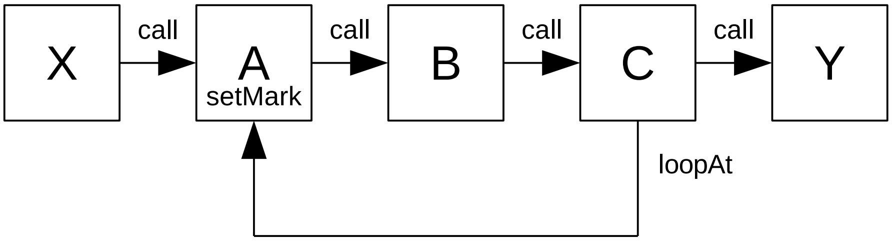

Figure 7.2. Proper calls in a loop.

When A2 calls setMark(M), the stack will look like this:

```
[ ] ~A2
[ ] ~X1, mark M
[ ]
```

The mark M remembers the frame one outer to the current one. The stack at the end of C4, after it has called `loopAt(M, A5)`, is:

```
[ ] ~C4
[ ] ~B3
[ ] ~A2
[A5] ~X1, mark M
[ ]
```

The stack then unwinds until A5 starts its execution:

```
[ ] ~A5
[ ] ~X1*, mark M
[ ]
```

Each iteration starts with a fresh stack, and the stack depth is limited to one iteration. The nested loops can also be properly executed.

Now, why does the mark get placed on the frame that is one out from the current one? After all, this means that X1 can not wait for the loop to complete. It has to return before the second iteration of the loop can start. And then the rest of the loop will run before the control returns to X1's caller. At least the caller of X1 can wait for the loop to complete before continuing its execution. Why all this trouble? Its the result of a compromise. Suppose that it did remember the current frame. Then at the end of C4 the stack will be:

```
[ ] ~C4
[ ] ~B3
[A5] ~A2, mark M
[ ] ~X1
[ ]
```

The stack will unwind until A5. Which would then have its own frame pushed onto the stack, and the code in the label A will call `setMark(M)` again, moving the mark to A5's own frame because it's the topmost frame now:

```
[ ] ~A5, mark M
[ ] ~A2*
[ ] ~X1
[ ]
```

So on each iteration of the loop one extra frame will be pushed onto the stack, and the mark moved by one level. A loop executing a million times will push a million frames, which is bad. Marking the next outer frame prevents this. Another option would have been to put the mark operation in X, but that would mean that every loop must have a preceding label that just marks the frame (well, and potentially could do the other initializations too), which seems to be too annoying.

It's one problem or the other, and the lesser problem won. To further reduce the complexity, I've also added the methods `makeLoopHead()` and `makeLoopAround()` that take care of constructing the whole front part of the loop, including the setting of the mark. They will be described below in Section 7.5: “Execution unit” (p. 46). This is still messy, and I'm still thinking about the ways to improve the situation.

What happens after the stack unwinds past the mark? The mark gets unset. When someone calls `loopAt()` with an unset mark, the rowop is enqueued in the outermost frame, having the same effect as schedule`()`.

This handling of an unset mark comes handy in case if the loop execution takes a pause in the middle. Suppose the label B finds that it can't process the rowop B3 until some other data has arrived. What it can do then is remember B3 somewhere in the thread state and return. The loop has not completed but it can't progress either, so the call unrolls until it becomes empty. Since the frame of X1 is popped off the stack, the mark M gets unset. The knowledge that the loop needs to be continued stays remembered in the state.

After some time that awaited data arrives, as some other rowop. When that rowop gets processed, it finds that remembered state with B3 and makes it continue, maybe by calling `call(B3)` again. So now the logic in B finds all the data it needs and continues with the loop, calling C4. C4 will do its job and call `loopAt(M, A5)`. But the mark M has been unset a while ago! Scheduling A5 at the outermost frame seems to be a logical thing to do at this point. Then whatever current processing will complete and unwind, and the loop will continue after it.

What if `setMark()` is called when there is only one frame on the stack? Then there is no second frame outer to it. The mark will simply be left unset.

But overall pausing and then restarting a loop like this is not such a good idea. The caller of the loop normally expects that it can wait for the loop to complete, and that when the loop returns, it's all done. If a loop may decide to bail out now and continue later, the caller has to be prepared for it.

And the sequence of execution when the loop continues might not be direct. In the normal execution one iteration of the loop follows directly after the previous one because of the orchestration by the label at the head of the loop (X in this example). When C calls `loopAt()`, the rowop gets pushed onto the stack frame of X. It would execute immediately only if X is draining its frame, and only if there are no other rowops queued on that frame in front of this one. I've been seeing it as a feature: X can easily be careful and make sure that the whole loop executes in one go without any interruptions. `makeLoopHead()` and `makeLoopAround()` create such careful labels. But it may also decide to run multiple loops interleaved, with each one making one iteration at a time. To do that, all it needs is fork the rowops to start all these loops and then drain the frame (directly or by returning from its own code). This way you can for example make a batch of records run through the loop (or even through the different loops) with the whole batch going throgh one iteration before the another iteration starts, achieving a kind of bundling. Though, if fork gets removed, this would not be possible any more. Maybe is's a reason for fork to stay, maybe it's a useless feature.

However if a loop decides to pause and then continues on some other event, its following `loopAt()` pushing the rowop onto the outermost frame, there is no caring parent to do the careful orchestration. There is no way to tell, which other rowops have been pushed onto the outermust frame by this time. The loop won't continue until these rowops execute. They may change the state of the model, so if the loop code expects it to stay the same, it will be mightily surprised.

7.5. Execution unit

After discussing the principles of scheduling in Triceps, let's get down to the nuts and bolts.

A unit is created as:

```
$myUnit = Triceps::Unit->new("name") or confess "$!";
```

The name argument is as usual used for later debugging, and by convention should be the same as the name of the unit variable (“myUnit” in this case). The name can also be changed later:

```
$myUnit->setName("newName");
```

It returns no value. Though in practice there is no good reason for changing names, and this call will likely be removed in the future. The name can be read back:

```
$name = $myUnit->getName();
```

Also, as usual, the variable $myName here contains a reference to the actual unit object, and two references can be compared for whether they refer to the same object:

```
$result = $unit1->same($unit2);
```

A unit also keeps an empty row type (one with no fields), primarily for the creation of the clearing labels, but you can use it for any other purposes too. You can get it with the method:

```
$rt = $unit->getEmptyRowType();
```

Each unit has its own instance of an empty row type. Its purely for the conveniece of memory management, they are all equivalent.

The rowops are enqueued with the calls:

```
$unit->call($rowop, ...);
$unit->fork($rowop, ...);
$unit->schedule($rowop, ...);
```

“Enqueued” is an ugly word but since I've already used the word “schedule” for a specific purpose, I needed another word to name all these operations together. Hence “enqueue”.

The “...” shows that multiple rowops may be passed as arguments. So the real signature of these methods is:

```
$unit->call(@rowops);
$unit->fork(@rowops);
$unit->schedule(@rowops);
```

But this way it loos more confusing. Calling these functions with multiple arguments produces the same result as doing multiple calls with one argument at a time. Not only rowops but also *trays* (to be discussed later) of rowops can be used as arguments.

These methods are among those that use the new error handling, that makes the operation to confess on any fatal errors. So there is no need to check their results with "`or confess`".

Also there is a call that selects the enqueueing mode by argument:

```
$unit->enqueque($mode, @rowops);
```

The calling rules are exactly the same for the other enqueueing methods, may have multiple rowops or trays as arguments, no need to check the result. The `$mode` argument is one of:

- `&Triceps::EM_CALL` or `"EM_CALL"`
- `&Triceps::EM_FORK` or `"EM_FORK"`
- `&Triceps::EM_SCHEDULE` or `"EM_SCHEDULE"`

As usual, there are calls to convert between the integer constant and string representations:

```
$string = &Triceps::emString($value);
$value = &Triceps::stringEm($string);
```

And as usual, if the value can not be translated, they return `undef`.

The frame marks for looping are created as their own class:

```
$mark = Triceps::FrameMark->new("name") or confess "$!";
```

The name can be obtained back from the mark:

```
$name = $mark->getName();
```

Other than that, the frame marks are completely opaque, and can be used only for the loop scheduling. Not even the `same()` method is supported for them at the moment, though it probably will be in the future. The mark gets set and used as:

```
$unit->setMark($mark);
$unit->loopAt($mark, @rowops);
```

The rowop arguments of the `loopAt()` are the same as for the other enqueueing functions, and as for other functions they may happen to be trays. These methods also use the new error handling scheme, and will confess on errors. No need to check the results.

There also are the convenience methods that create the rowops from the field values and immediately enqueue them:

```
$unit->makeHashCall($label, $opcode,
  $fieldName => $fieldValue, ...);
$unit->makeArrayCall($label, $opcode, @fieldValues);

$unit->makeHashSchedule($label, $opcode,
  $fieldName => $fieldValue, ...);
$unit->makeArraySchedule($label, $opcode, @fieldValues);

$unit->makeHashLoopAt($mark, $label, $opcode,
  $fieldName => $fieldValue, ...);
$unit->makeArrayLoopAt($mark, $label, $opcode, @fieldValues);
```

These are essentially the shorter ways to make the rowops and enqueue them without the three-deep calls. Only the methods for the most frequently used enqueueing modes are provided, not for all of them. All these methods also confess on errors.

The methods for creation of labels have been already discussed in Section 6.2: "Label construction" (p. 32) . Here is their recap along with the similar methods for creation of tables and trays that will be discussed later:

```
$label = $unit->makeDummyLabel($rowType, "name")
  or confess "$!";
```

```
$label = $unit->makeLabel($rowType, "name",
  $clearSub, $execSub, @args) or confess "$!";

$label = $unit->makeClearingLabel("name", @args);

$table = $unit->makeTable($tableType, $enqMode, "name")
  or confess "$!";

$tray = $unit->makeTray(@rowops) or confess "$!";
```

Of them `makeClearingLabel()` uses the new error handling convention, confessing by itself, and the rest return an undef on errors that has to be checked. It's actually real difficult to make `makeClearingLabel()` fail, only by corrupting some of the Triceps internal variables, and it was a late additiion, so going straight with the new convention for it made sense.

A special thing about the labels is that when a unit creates a label, it keeps a reference to it, for clearing. A label keeps a pointer back to the unit but not a reference (if you call `getUnit()` on a label, the returned value becomes a reference). For a table or a tray, the unit doesn't keep a reference to them. Instead, they keep a reference to the unit. With the tables, it can get pretty involved: A table has labels associated with it. When a table is created, it also creates these labels. The unit keeps references of these labels. The table also keeps references of these labels. The table keeps a reference of the unit. The labels (they are at the C++ level, not Perl level) have pointers to the unit and the table but not references, to avoid the reference cycles.

See more on the memory management and label clearing in the Chapter 8: “*Memory Management*” (p. 69) .

The convenience methods to create the whole front part of the topological loop are:

```
($labelBegin, $labelNext, $frameMark) = $unit->makeLoopHead(
  $rowType, "name", $clearSub, $execSub, @args);

($labelBegin, $labelNext, $frameMark) = $unit->makeLoopAround(
  "name", $labelFirst);
```

You don't have to use them, you can create the loops manually. These methods merely make it more convenient. Remember also that a procedural loop is usually much easier to write and debug and read later than a topological one.

These methods use the new error handling convention, confessing on the errors. There is no need to check the result.

`makeLoopHead()` creates the front part of the loop that starts with a Perl label. It gets the arguments for that label and creates it among the other things. `makeLoopAround()` creates the front part of the loop around an existing label that will be the first one executed in the loop. `makeLoopHead()` is really redundant and can be replaced with a combination of `makeLabel()` and `makeLoopAround()`.

They both return the same results, a triplet:

- The label where you send a rowop to initiate the loop (remember that the loop consists of one row going through the loop at a time), `$labelBegin`.
- The label that you use at the end of the loop in the `loopAt()` to do the next iteration of the loop, `$labelNext`.
- The frame mark that you use in `loopAt()`, `$frameMark`. You don't need to set the frame mark, it will be set for you in the wrapper logic.

The name is used to construct the names of the elements by adding a dotted suffix: “name.begin”, “name.next” for `makeLoopHead()` or “name.wrapnext” for `makeLoopAround()`, “name.mark”.

`makeLoopAround()` takes the row type for its created labels from the first label that is given to it as an argument.

The makeLoop methods may be easier to understand if you look at their source code:

```
sub makeLoopHead # ($self, $rt, $name, $clearSub, $execSub, @args)
{
  my ($self, $rt, $name, $clear, $exec, @args) = @_;

  my $mark = Triceps::FrameMark->new($name . ".mark") or confess "$!";

  my $lbNext = $self->makeLabel($rt, $name . ".next", $clear, sub {
    $self->setMark($mark);
    &$exec(@_);
  }, @args) or confess "$!";

  my $lbBegin = $self->makeLabel($rt, $name . ".begin", undef, sub {
    $self->call($lbNext->adopt($_[1]));
  }) or confess "$!";

  return ($lbBegin, $lbNext, $mark);
}

sub makeLoopAround # ($self, $name, $lbFirst)
{
  my ($self, $name, $lbFirst) = @_;
  my $rt = $lbFirst->getRowType();

  my $mark = Triceps::FrameMark->new($name . ".mark") or confess "$!";

  my $lbWrapNext = $self->makeLabel($rt, $name . ".wrapnext", undef, sub {
    $self->setMark($mark);
  }) or confess "$!";
  $lbWrapNext->chain($lbFirst) or confess "$!";

  my $lbBegin = $self->makeLabel($rt, $name . ".begin", undef, sub {
    $self->call($lbWrapNext->adopt($_[1]));
  }) or confess "$!";

  return ($lbBegin, $lbWrapNext, $mark);
}
```

The label execution handlers in them use `$_[1]` to get their rowop argument, without assigning it to a variable first. An extended example of them will also be shown in Section 7.9: “Example of a topological loop” (p. 56) .

The unit can be checked for the emptiness of its queues:

```
$result = $unit->empty();
```

Also the current depth of the call stack (the number of the stack frames on the queue) can be found with:

```
$result = $unit->getStackDepth();
```

It isn't of any use for the model logic as such but comes handy for debugging, to check in the loops that you haven't accidentally created a stack growing with iterations. When the unit is not running, the stack depth is 1, since the outermost frame always stays on the stack. When a rowop is being executed, the stack depth is at least 2.

The functions for execution from the queues are:

```
$unit->callNext();
$unit->drainFrame();
```

`callNext()` takes one label from the top (innermost) stack frame queue and calls it. If the innermost frame happens to be empty, it does nothing. `drainFrame()` calls the rowops from the top stack frame until it becomes empty. This includes any rowops that may be created and enqueued as part of the execution of the previous rowops. But it doesn't pop the frame from the stack. And of course the method `call()` causes the argument rowops to be executed immediately, without even being technically enqueued.

7.6. Error handling during the execution

When the labels execute, they may produce errors in one of two ways:

- The Perl code in the label might die.

- The call topology might violate the rules.

The rules are basically that you can't make the recursive calls. A label may not make calls directly or through other labels to itself. The idea is to catch the call sequences that are likely to go into the deep recursion and overflow the stack. It catches them early, on the first attempt of recursion. If you need to do the recursion, use `schedule()` or `loopAt()`. That way you avoid overrunning the stack.

Whichever way the error is detected, it causes the Triceps call stack to be unwound. The Perl error messages from `die` or `confess` and the C++ tracing of rowop calls and label chainings get combined into a common stack trace. When the code gets back to Perl, the XS code triggers a `confess`. If that happens to be in the handler of another rowop, it continues the Triceps hybrid stack unwinding. If not caught by `eval`, it keeps going to the topmost `call()` or `drainFrame()` and causes the whole program to die. Which is a reasonable reaction most of the time.

Remember, the root cause is a serious error that is likely to leave the model in an inconsistent state, and it should usually be considered as fatal. If you want to catch the errors, nip them in the bud by wrapping your Perl code in `eval`. Then you can handle the errors before thay have a chance to propagate.

An interesting question is, what happens to the rowops that were in the Triceps stack frames when the stack gets unwound? They get thrown away. The memory gets collected thanks to the reference counting, but the rowops and their sequence order get thrown away. The reason is basically that there may be no catching of the errors until unwinding to the `drainFrame()`. The choice is to either throw away everything after the first error or keep trying to execute the following rowops, collecting the errors. And that might become a lot of errors. I've taken the choice of stopping as early as possible, because the state of the model will probably be corrupted anyway and nothing but garbage would be coming out (if anything would be coming at all and not be stuck in an endless loop).

7.7. The main loop

The execution unit doesn't magically process the data by itself. The data needs to be pushed into it, and the unit has to be told to process it. There has to be some internal code to drive it, that would continuously read the data, schedule, drain.

A typical way of processing the incoming rowops in a loop is:

```
$stop = 0;
while (!$stop) {
  $rowop = &readRowop(); # some user-defined function
  $unit->call($rowop);
  $unit->drainFrame();
}
```

The rowops coming from the outside get executed as they are received, and then any rows left over from them get handled by `drainFrame()` before the next incoming rowop is read. Some of the executed rowops may set `$stop`, and the main loop will exit.

There is also another version of the main loop that has been more historic:

```
$stop = 0;
while (!$stop) {
  $rowop = &readRowop(); # some user-defined function
  $unit->schedule($rowop);
  $unit->drainFrame();
}
```

It uses `schedule()` instead of `call()` for the rowop. As long as only one rowop is scheduled before draining the frame, both versions work equally well. But if you schedule multiple rowops before draining the frame, you can introduce a subtle unpredictability in the execution order. It is described in detail in Section 7.10: “Issues with the Triceps scheduling” (p. 59) . Actually, you can have the same problem if you don't drain the frame after each top-level `call()` too. But mentally `call()` kind of reminds better to feed the rowops one at a time, and also is slightly more efficient, so now I prefer the version with it.

Many of the examples in this manual use the main loop along the following lines (with variations, to fit the examples, and as the main loop was refined over time):

```
while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "lbCur") {
    $unit->makeArrayCall($lbCur, @data);
  } elsif ($type eq "lbPos") {
    $unit->makeArrayCall($lbPos, @data);
  }
  $unit->drainFrame();
}
```

It reads the CSV (Comma-Separated Values) data from stdin, with the label name in the first column, the opcode in the second, and the data fields in the rest. Then dispatches according to the label. Doing a `call()` instead of `schedule()` works just as well, and the following `drainFrame()` takes care of any rowops scheduled from the call.

Many variations are possible. It can be generalized to look up the labels from the hash:

```
while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  $unit->makeArrayCall($labels{$type}, @data);
  $unit->drainFrame();
}
```

Or call the procedural functions for some types:

```
while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "lbCur") {
    $unit->makeArrayCall($lbCur, @data);
  } elsif ($type eq "lbPos") {
    $unit->makeArrayCall($lbPos, @data);
  } elsif ($type eq "clear") { # clear the previous day
    &clearByDate($tPosition, @data);
  }
  $unit->drainFrame();
}
```

Though none of these small examples are production-ready. At the very least, their parsing of the CSV data is primitive. It can't handle the quoting properly and can't parse the data with commas in it.

A better ready way to parse the data will be provided in the future. For now, make your own.

7.8. Main loop with a socket

A fairly typical situation is when a CEP model has to run in a daemon process, receiving and sending data through the network sockets. Here goes an example that does this. It's not production-ready either. It still has the issue with the parsing of the CSV data, its handling of the errors is not well-tested, and it makes a few simplifying assumptions about the buffering (more on this below). Other than that, it's a decent starting point. If you want to copy-and-paste this code for your experiments, it can be found in `perl/Triceps/t/xQuery.t`.

```
use Triceps;
use Carp;
use Errno qw(EINTR EAGAIN);
use IO::Poll qw(POLLIN POLLOUT POLLHUP);
use IO::Socket;
use IO::Socket::INET;

# the socket and buffering control for the main loop;
# they are all indexed by a unique id
our %clients; # client sockets
our %inbufs; # input buffers, collecting the whole lines
our %outbufs; # output buffers
our $poll; # the poll object
our $cur_cli; # the id of the current client being processed
our $srv_exit; # exit when all the client connections are closed

# writing to the output buffers
sub outBuf # ($id, $string)
{
  my $id = shift;
  my $line = shift;
  $outbufs{$id} .= $line;
  # If there is anything to write on a buffer, stop reading from it.
  $poll->mask($clients{$id} => POLLOUT);
}

sub outCurBuf # ($string)
{
  outBuf($cur_cli, @_);
}

sub closeClient # ($id, $h)
{
  my $id = shift;
  my $h = shift;
  $poll->mask($h, 0);
  $h->close();
  delete $clients{$id}; # OK perl Perl manual even when iterating
  delete $inbufs{$id};
  delete $outbufs{$id};
}

# The server main loop. Runs with the specified server socket.
# Uses the labels hash to send the incoming data to Triceps.
sub mainLoop # ($srvsock, $%labels)
{
  my $srvsock = shift;
  my $labels = shift;

  my $client_id = 0; # unique strings
  our $poll = IO::Poll->new();

  $srvsock->blocking(0);
  $poll->mask($srvsock => POLLIN);
```

```
$srv_exit = 0;

while(!$srv_exit || keys %clients != 0) {
  my $r = $poll->poll();
  confess "poll failed: $!" if ($r < 0 && ! $!{EAGAIN} && ! $!{EINTR});

  if ($poll->events($srvsock)) {
    while(1) {
      my $client = $srvsock->accept();
      if (defined $client) {
        $client->blocking(0);
        $clients{++$client_id} = $client;
        # print("Accepted client $client_id\n");
        $poll->mask($client => (POLLIN|POLLHUP));
      } elsif($!{EAGAIN} || $!{EINTR}) {
        last;
      } else {
        confess "accept failed: $!";
      }
    }
  }

  my ($id, $h, $mask, $n, $s);
  while (($id, $h) = each %clients) {
    $cur_cli = $id;
    $mask = $poll->events($h);
    if (($mask & POLLHUP) && !defined $outbufs{$id}) {
      # print("Lost client $client_id\n");
      closeClient($id, $h);
      next;
    }
    if ($mask & POLLOUT) {
      $s = $outbufs{$id};
      $n = $h->syswrite($s);
      if (defined $n) {
        if ($n >= length($s)) {
          delete $outbufs{$id};
          # now can accept more input
          $poll->mask($h => (POLLIN|POLLHUP));
        } else {
          substr($outbufs{$id}, 0, $n) = '';
        }
      } elsif(! $!{EAGAIN} && ! $!{EINTR}) {
        warn "write to client $id failed: $!";
        closeClient($id, $h);
        next;
      }
    }
    if ($mask & POLLIN) {
      $n = $h->sysread($s, 10000);
      if ($n == 0) {
        # print("Lost client $client_id\n");
        closeClient($id, $h);
        next;
      } elsif ($n > 0) {
        $inbufs{$id} .= $s;
      } elsif(! $!{EAGAIN} && ! $!{EINTR}) {
        warn "read from client $id failed: $!";
        closeClient($id, $h);
        next;
      }
      }
      # The way this works, if there is no '\n' before EOF,
      # the last line won't be processed.
      # Also, the whole output for all the input will be buffered
      # before it can be sent.
      while($inbufs{$id} =~ s/^(.*)\n//) {
        my $line = $1;
        chomp $line;
        local $/ = "\r"; # take care of a possible CR-LF
        chomp $line;
        my @data = split(/,/, $line);
        my $lname = shift @data;
        my $label = $labels->{$lname};
        if (defined $label) {
          my $unit = $label->getUnit();
          confess "label '$lname' received from client $id has been cleared"
            unless defined $unit;
          eval {
            $unit->makeArrayCall($label, @data);
            $unit->drainFrame();
          };
          warn "input data error: $@\nfrom data: $line\n" if $@;
        } else {
          warn "unknown label '$lname' received from client $id: $line "
        }
      }
    }
  }
}
```

The general outline follows the single-threaded multiplexing server described in [Babkin10]. `mainLoop()` gets the server socket and a dispatch table of labels as its arguments. It then proceeds with waiting for connections.

Once a connection is received, it gets added to the set of active connections, to get included in the waiting for the input data. The input data is read as simplified CSV (no commas in the middle of values, and no way to reprsent the NULL values othar than for those omitted at the end of the line). It's expected to have the format:

```
name,opcode,data...
```

Such as:

```
window,OP_INSERT,5,AAA,30,30
window.query,OP_INSERT
exit,OP_NOP
```

The name part is then used to find a label in the dispatch table. The rest of the data is used to create a rowop for that label and execute it.

The data is sent back to the client through buffering. To send some data to a client, use

```
&outBuf($id, $text);
```

The `$id` is the unique id of the client. How do you find, what is the id of the client you want to send the data to? When an input line is processed, the main loop knows, from what client it was received. It puts the id of that client in the global variable `$cur_cli`. You can take it from there and remember. If you want to reply to the current client, you don't need to bother yourself with the id at all, just call

```
&outCurBuf($text);
```

If you remember an id for the future use, you'd probably need to check that the client hasn't disconnected before sending data to it:

```
if (exists $clients{$id}) {
  &outBuf($id, $text);
}
```

Otherwise your output attempts would be leaking memory in the output buffer. In any case, if a client has disconnected, the further processing of its requests shoudl usually be stopped. The client ids are not reused, so this check is always safe.

Once some output is buffered to send to a client, the further input from that client stops being accepted until the output buffer drains. But the processing in the Triceps unit scheduler keeps running until it runs out of things to do before it returns to the main loop. All this time the output buffer keeps collecting data without sending it to the client. Also, the input buffer might happen to already contain multiple lines. Then all these lines will be processed before the data from the output buffer starts being sent to the client. If a request produces a large amount of data, all this data will be buffered first. It's a simplification but really the commercial CEP systems aren't doing a whole lot better: when asked for the contents of a table/window/materliaized view, Coral8 and Aleri and Sybase (don't know about StreamBase but it might be not different either) would make a copy of it first before sending the data. In some cases the copy is more efficient because it references the rows rather than copying the whole byte data, but in the grand scheme of things it's all the same.

Internally the information about the client sockets and their buffers is kept in the global hashes `%clients`, `%inbufs`, `%outbufs`. It could be done a a single hash of objects but this was simpler.

The loop exits when the global variable `$srv_exit` gets set (synchronously, i.e. by one of the label handlers) to 1 and all the clients disconnect. The requirement for disconnection of all the clients makes sure that all the output buffers get flushed before exit, and that was the easiest way to achieve this goal.

`mainLoop()` relies on the listening socket being already created, bound and given to it as a parameter. Here is a function used in the examples to create this socket and run the server in a separate process:

```
sub startServer # ($labels)
{
  my $labels = shift;

  my $srvsock = IO::Socket::INET->new(
    Proto => "tcp",
    LocalPort => 0,
    Listen => 10,
  ) or confess "socket failed: $!";
  my $port = $srvsock->sockport() or confess "sockport failed: $!";
  my $pid = fork();
  confess "fork failed: $!" unless defined $pid;
  if ($pid) {
    # parent
    $srvsock->close();
  } else {
    # child
    &mainLoop($srvsock, $labels);
    exit(0);
  }
  return ($port, $pid);
}
```

It binds the socket to the port 0 to request that the OS bind it to a randum unused port. The port number is then read back with `sockport()`. The pair of the port numer and the server's child process id is then returned as the result. The process where the server runs is in this case just a child process, it's not properly daemonized.

For a simple complete example, let's make an echo server that would print back the rows it receives:

```
my $uEcho = Triceps::Unit->new("uEcho");
my $lbEcho = $uEcho->makeLabel($rtTrade, "echo", undef, sub {
  &outCurBuf($_[1]->printP() . "\n");
});
```

```
my $lbEcho2 = $uEcho->makeLabel($rtTrade, "echo2", undef, sub {
  &outCurBuf(join(",", "echo", &Triceps::opcodeString($_[1]->getOpcode()),
    $_[1]->getRow()->toArray()) . "\n");
});
my $lbExit = $uEcho->makeLabel($rtTrade, "exit", undef, sub {
  $srv_exit = 1;
});

my %dispatch;
$dispatch{"echo"} = $lbEcho;
$dispatch{"echo2"} = $lbEcho2;
$dispatch{"exit"} = $lbExit;

my ($port, $pid) = &startServer(\%dispatch);
print STDERR "port=$port pid=$pid\n";
waitpid($pid, 0);
exit(0);
```

It starts the server and waits for it to exit. `waitpid()` is used here in a simplified way too, it should properly be done in a loop until it succeeds or an error other than `EINTR` is returned.

`$rt` is the row type for the expected data. It's not particularly important here, so I didn't show its definition. Two labels, “echo” and “echo2” differ in the way they print the data back: “echo” prints it in the symbolic form while “echo2” prints in CSV. The label “exit” sets the exit flag.

The names in the dispatch table don't have to be the same as the names of the labels. It's often convenient to have them the same but not mandatory.

7.9. Example of a topological loop

How to build the models with the topological loop is much easier to understand with an example. So let's make an example that computes the Fibonacci numbers. It's a real overcomplicated and perverse way of calculating the Fibonacci numbers. But it also is a great fit to the type of problems that get solved with the topological loop, one of a simple kind.

First, a quick reminder of what is a Fibonacci number. Historically it's a solution to the problem of breeding the spherical rabbits in vacuum. But in the mathematical reality it's the sequence of numbers where each number is a sum of the two previous ones. Two initial elements are defined to be equal to 1, and it goes from there:

$F_i = F_{i-1} + F_{i-2}$

$F_1 = 1; F_2 = 1$

The Fibonacci numbers are often used as an example of recursive computations in the beginner's books on programming. The computation of the *n*-th Fibonacci number is usually shown computed like this:

```
sub fib1 # ($n)
{
  my $n = shift;
  if ($n <= 2) {
    return 1;
  } else {
    return &fib1($n-1) + &fib1($n-2);
  }
}
```

However that's not a good way to compute in the real world. When a function calls itself recursively once, its complexity is linear, $O(n)$. When a function calls itself twice or more, its complexity becomes exponential, $O(e^n)$. At first you might think that it's only quadratic $O(n^2)$ because it forks two ways on each step. But these two ways keep forking and forking on each step, and it compounds to exponential. Which is a real bad thing.

To think of it, it's a huge waste, since the (*n*-2)—th number is calculated anyway for the (*n*-1)—th number. Why calculate it separately the second time? We could as well have saved and reused it. The Lisp people have figured this out a long time ago, and the Lisp books (if you can read Finnish or Russian, [Hyvonen86] is a classical one) are full of examples that do exactly that. However I'm too lazy to explain how they work, so we're going to skip it together with the conversion of a tail recursion into a loop and get directly to the loop version. I find the loop version more natural and easier to write than a recursion anyway.

```
sub fibStep2 # ($prev, $preprev)
{
  return ($_[0] + $_[1], $_[0]);
}

sub fib2 # ($n)
{
  my $n = shift;
  my @prev = (1, 0); # n and n-1

  while ($n > 1) {
    @prev = &fibStep2(@prev);
    $n--;
  }
  return $prev[0];
}
```

The split into two functions is not mandatory for the loop version, it just does the clean separation of the loop counter logic and of the computation of the next step of the function. (But for the recursion version if would be mandatory).

I'm going to take this procedural loop version and transform it into a topological loop. It actually happens to be a real good match for the topological loop. In a topological loop a record keeps traveling through it and being transformed until it satisfies the loop exit condition. Here `@prev` is the record contents, and the iteration count will be added to them to keep track of the exit condition.

```
$uFib = Triceps::Unit->new("uFib") or confess "$!";

my $rtFib = Triceps::RowType->new(
  iter => "int32", # iteration number
  cur => "int64", # current number
  prev => "int64", # previous number
) or confess "$!";

my $lbPrint = $uFib->makeLabel($rtFib, "Print", undef, sub {
  print($_[1]->getRow()->get("cur"));
});

my $lbCompute; # will fill in later

my ($lbBegin, $lbNext, $markFib) = $uFib->makeLoopHead(
  $rtFib, "Fib", undef, sub {
    my $iter = $_[1]->getRow()->get("iter");
    if ($iter <= 1) {
      $uFib->call($lbPrint->adopt($_[1]));
    } else {
      $uFib->call($lbCompute->adopt($_[1]));
    }
  }
);

$lbCompute = $uFib->makeLabel($rtFib, "Compute", undef, sub {
  my $row = $_[1]->getRow();
  my $cur = $row->get("cur");
  $uFib->makeHashLoopAt($markFib, $lbNext, $_[1]->getOpcode(),
    iter => $row->get("iter") - 1,
    cur => $cur + $row->get("prev"),
    prev => $cur,
  );
}) or confess "$!";

my $lbMain = $uFib->makeLabel($rtFib, "Main", undef, sub {
  my $row = $_[1]->getRow();
  $uFib->makeHashCall($lbBegin, $_[1]->getOpcode(),
    iter => $row->get("iter"),
    cur => 1,
    prev => 0,
  );
  print(" is a Fibonacci number ", $row->get("iter"), "\n");
}) or confess "$!";

while(<STDIN>) {
  chomp;
  my @data = split(/,/);
  $uFib->makeArrayCall($lbMain, @data);
  $uFib->drainFrame(); # just in case, for completeness
}
```

You can see that it has grown quite a bit. That's why the procedural loops are generally a better idea. However if the computation involves a lot of the SQLy logic, the topological loops are still beneficial. Also, the Triceps `call()` is at the moment not a real call: you pass the arguments, you call the code but you don't directly get the results. You could pass them back through some static variables, or you could use the topological loop to pass them directly to the next iteration of the loop.

The main loop reads the CSV lines with opcodes (which aren't really used here, just passed through and then thrown away before printing) and calls `$lbMain`. Here is an example of an input and output as they would intermix if the input was typed from the keyboard. As in the rest of this manual, the input lines are shown in bold.

```
OP_INSERT,1
1 is a Fibonacci number 1
OP_DELETE,2
1 is a Fibonacci number 2
OP_INSERT,5
5 is a Fibonacci number 5
OP_INSERT,6
8 is a Fibonacci number 6
```

The input lines contain the values only for the field `iter`, which intentionally happens to be the first field in the row type. The other fields will be reset anyway in `$lbMain`, so they are left as NULL.

The point of `$lbMain` is to call the loop begin label `$lbBegin` and then print the message about which Fibonacci number was requested. The value of the computed number is printed at the end of the loop, so when the words “is a Fibonacci number” are printed after it, that demonstrates that the execution of `$lbMain` continues only after the loop is completed.

The loop logic is split into two labels `$lbNext` and `$lbCompute` purely to show that it can be split like this. `$lbNext` handles the loop termination condition, and `$lbCompute` does essentially the work of `fibStep2()`. After the loop terminates, it passes the result row to `$lbPrint` for the priniting of the value.

When the code for `$lbNext` is created, it contains the call of `$lbCompute`. However the label `$lbCompute` has not been created at this time yet! Not a problem, creating in advance an empty variable `$lbCompute` is enough. The closure in `$lbNext` will keep a reference to that variable, and the variable will be filled with the reference to the label later (but before the main loop executes).

And here is the version with `makeLoopAround()`:

```
my ($lbBegin, $lbNext, $markFib); # will fill in later
```

```
$lbCompute = $uFib->makeLabel($rtFib, "Compute", undef, sub {
  my $row = $_[1]->getRow();
  my $cur = $row->get("cur");
  my $iter = $row->get("iter");
  if ($iter <= 1) {
    $uFib->call($lbPrint->adopt($_[1]));
  } else {
    $uFib->makeHashLoopAt($markFib, $lbNext, $_[1]->getOpcode(),
      iter => $row->get("iter") - 1,
      cur => $cur + $row->get("prev"),
      prev => $cur,
    );
  }
}) or confess "$!";

($lbBegin, $lbNext, $markFib) = $uFib->makeLoopAround(
  "Fib", $lbCompute
);
```

The unit, row type, `$lbPrint`, `$lbMain` and the main loop have stayed the same, so they are omitted from this example. The whole loop logic, both the termination condition and the computation step, have been collected into one label `$lbCompute`, to show that it can be done this way too. Then the loop head is created around `$lbCompute`.

7.10. Issues with the Triceps scheduling

As much as I like it, the Triceps scheduling is not perfect, and has some open issues at the moment. Some of them have been already mentioned in the description of the loop scheduling: it's a bit confusing that the frame mark is placed on the next outer scheduling stack frame and not on the current one. This leads to the interesting effects in execution order.

The other one has been mentioned in the main loop discussion: the `schedule()` call, when used from inside the scheduled code, may introduce unpredictability in the execution order. It puts the rowop after the last rowop in the outermost stack frame. But the outermost stack frame may contain a whole queue of rowops that come from the outside. This means that the exact order of execution will depend on the timing of the rowops arriving from outside.

Let me demonstrate it with an example. Suppose the main loop tries to optimize by collecting and schedulng as many incoming rowops as it can before running them:

```
$stop = 0;
while (!$stop) {
  &waitForIncomingData(); # some user-defined function
  while ($rowop = &readRowop()) { # some user-defined function
    $unit->schedule($rowop);
  }
  $unit->drainFrame();
}
```

Suppose the rowops A, B, C, D are being received from the outside. When the rowop A executes, it schedules the rowop E. Then depending on the timing of the packets in the network, the call sequence may be

```
schedule(A)
drainFrame()
schedule(B)
drainFrame()
schedule(C)
drainFrame()
schedule(D)
drainFrame()
```

or

```
schedule(A)
schedule(B)
drainFrame()
schedule(C)
schedule(D)
drainFrame()
```

or

```
schedule(A)
schedule(B)
schedule(C)
schedule(D)
drainFrame()
```

or a few other combinations. In the first case the actual execution order will be A, E, B, C, D. That's because when A schedules E, E will be picked up and executed by the first following frame drain. In the second case the execution order will be A, B, E, C, D. Here when E gets scheduled, B is already on the queue in front of it. In the third case the order will be A, B, C, D, E. And it will fluctuate at random between the runs.

If the repeatable execution order is important (and usually it is), the solution if to feed the rowops one by one and drain the frame right afterwards. Then the execution order will always be A, E, B, C, D. When feeding one by one, `call()` can be used instead of `schedule()`, and even slightly more efficient. Just don't forget to drain the frame after each call.

The same issue happens with the topological loops that have been temporarily stopped and then resumed on arrival of more data from outside. The mark of such a loop will be unset when the loop continues, and looping at this mark will be equivalent to `schedule()`, having the same repeatability problem. The same solution works for this issue too.

The method `fork()` is not exactly useful. It was created when I've thought that it's the solution to the problem of the loops. Which it has turned out to not solve, and another solution had to be devised. Now it really doesn't have much use, and will probably be removed in the future.

I have a few ideas for better solutions of these issues, but they will need a bit more experimentation. Just keep in mind that the scheduling will be refined in the future. It will still have the same general shape but differ in detail.

7.11. Trays, or yes bundling

Even though Triceps does no bundling in scheduling, there still is a need to store the sequences of row operations. This is an important distinction, since the stored sequences are to be scheduled somewhere in the future (or maybe not scheduled at all, but iterated through manually). If and when they get scheduled, they will be unbundled. The ordered storage only provides the order for that future scheduling or iteration.

The easiest way to store rowops is to put them into the Perl arrays, like:

```
my @ops = ($rowop1, $rowop2);
push @ops, $rowop3;
```

However the C++ internals of Triceps do not know about the Perl arrays. And some of them can work directly with the sequences of rowops. So Triceps defines an internal sort-of-equivalent of Perl array for rowops, called a *Tray*.

The trays have first been used to “catch” the side effects of operations on the stateful elements, so the name “tray” came from the metaphor “put a tray under it to catch the drippings”.

The trays get created as:

```
$tray = $unit->makeTray(@rowops) or confess "$!";
```

A tray always stores rowops for only one unit. It can be only used in one thread. A tray can be used in all the enqueueing methods, just like the direct rowops:

```
$unit->call($tray);
$unit->fork($tray);
$unit->schedule($tray);
$unit->enqueue($mode, $tray);
$unit->loopAt($mark, $tray);
```

Moreover, multiple trays may be passed, and the loose rowops and trays can be mixed in the multiple arguments of these functions, for example:

```
$unit->call($rowopStartPkg, $tray, $rowopEndPkg);
```

In this example nothing really stops you from placing the start and end rowops into the tray too. A tray may contain the rowops of any types mixed in any order. This is by design, and it's an important feature that allows to build the protocol blocks out of rowops and perform an orderly data exchange. This feature is an absolute necessity for proper inter-process and inter-thread communication.

The ability to send the rows of multiple types through the same channel in order is a must, and its lack makes the communication with some other CEP systems exceedingly difficult. Coral8 supports only one stream per connection. Aleri (and I believe Sybase R5) allows to send multiple streams through the same connection but has no guarantees of order between them. I don't know about the others, check yourself.

To iterate on a tray in the Perl code, it can be converted to a Perl array:

```
@array = $tray->toArray();
```

The size of the tray (the count of rowops in it) can be read directly without a conversion, and the unit can be read back too:

```
$size = $tray->size();
$traysUnit = $tray->getUnit();
```

Another way to create a tray is by copying an existing one:

```
$tray2 = $tray1->copy();
```

This copies the contents (which is the references to the rowops) and does not create any ties between the trays. The copying is really just a more efficient way to do

```
$tray2 = $tray1->getUnit()->makeTray($tray1->toArray());
```

The tray references can be compared for whether they point to the same tray object:

```
$result = $tray1->same($tray2);
```

The contents of a tray may be cleared. Which is more convenient and more efficient than discarding a tray and creating another one:

```
$tray->clear();
```

The data may be added to the back of a tray:

```
$tray->push(@rowops);
```

Multiple rowops can be pushed in a single call. There are no other Perl-like operations on a tray: it's either create from a set of rowops, push, or convert to a Perl array.

Note that the trays are mutable, unlike the rows and rowops. Multiple references to a tray will see the same contents. If a tray is changed through one reference, the others will see the changes too.

7.12. Tracing the execution

When developing the CEP models, there always comes the question: WTF had just happened? How did it manage get this result? Followed by subscribing to many intermediate results and trying to piece together the execution order.

Triceps provides two solutions for this situation: First, the procedural approach should make the logic much easier to follow. Second, it has a ready way to trace the execution and then read the trace in one piece. It can also be used to analyze any variables on the fly, and possibly stop the execution and enter some manual mode.

The idea here is simple: provide the Unit with a method that will be called:

- before a label executes,
- after the label executes but before draining its frame,
- after the frame is drained but before the chained labels execute,
- after all the execution caused by the label is completed.

For the simple tracing, there is a small simple tracer provided. It actually executes directly as compiled in C++ so it's quite efficient:

```
$tracer = Triceps::UnitTracerStringName(option => $value, ...)
  or confess "$!";
```

The arguments are specified as the option name-value pairs.

The only option supported is `verbose`, which may be 0 (default) or non-0. If it's 0 (false), the tracer will record a message only before executing each label. If true, it will record a message after each stage. The class is named UnitTracerStringName because it records the execution trace in the string format, including the names of the labels. The tracer is set into the unit:

```
$unit->setTracer($tracer);
```

The unit's current tracer can also be read back:

```
$oldTracer = $unit->getTracer();
```

If no tracer was previously set, `getTracer()` will return `undef`. And `undef` can also be used as an argument of `setTracer()`, to cancel any previously set tracing. `setTracer()` has the new-style error handling and confesses on errors.

The tracer references can be compared for whether they refer to the same underlying object:

```
$result = $tracer1->same($tracer2);
```

There are multiple kinds of tracer objects, and `same()` can be called safely for either kind of tracer, including mixing them together. Of course, the tracers of different kinds definitely would not be the same tracer object.

As the unit runs, the tracing information gets collected in the tracer object. It can be extracted back with:

```
$data = $tracer->print();
```

This does not reset the trace. To reset it, use:

```
$tracer->clearBuffer();
```

Here is a code sequence designed to produce a fairly involved trace:

```
$sntr = Triceps::UnitTracerStringName->new(verbose => 1);
$u1->setTracer($sntr);
```

```
$c_lab1 = $u1->makeDummyLabel($rt1, "lab1")
  or confess "$!";
$c_lab2 = $u1->makeDummyLabel($rt1, "lab2")
  or confess "$!";
$c_lab3 = $u1->makeDummyLabel($rt1, "lab3")
  or confess "$!";

$c_op1 = $c_lab1->makeRowop(&Triceps::OP_INSERT, $row1)
  or confess "$!";
$c_op2 = $c_lab1->makeRowop(&Triceps::OP_DELETE, $row1)
  or confess "$!";

$v = $c_lab1->chain($c_lab2) or confess "$!";
$v = $c_lab1->chain($c_lab3) or confess "$!";
$v = $c_lab2->chain($c_lab3) or confess "$!";

$u1->schedule($c_op1);
$u1->schedule($c_op2);

$u1->drainFrame();
```

The trace is:

```
unit 'u1' before label 'lab1' op OP_INSERT {
unit 'u1' drain label 'lab1' op OP_INSERT
unit 'u1' before-chained label 'lab1' op OP_INSERT
unit 'u1' before label 'lab2' (chain 'lab1') op OP_INSERT {
unit 'u1' drain label 'lab2' (chain 'lab1') op OP_INSERT
unit 'u1' before-chained label 'lab2' (chain 'lab1') op OP_INSERT
unit 'u1' before label 'lab3' (chain 'lab2') op OP_INSERT {
unit 'u1' drain label 'lab3' (chain 'lab2') op OP_INSERT
unit 'u1' after label 'lab3' (chain 'lab2') op OP_INSERT }
unit 'u1' after label 'lab2' (chain 'lab1') op OP_INSERT }
unit 'u1' before label 'lab3' (chain 'lab1') op OP_INSERT {
unit 'u1' drain label 'lab3' (chain 'lab1') op OP_INSERT
unit 'u1' after label 'lab3' (chain 'lab1') op OP_INSERT }
unit 'u1' after label 'lab1' op OP_INSERT }
unit 'u1' before label 'lab1' op OP_DELETE {
unit 'u1' drain label 'lab1' op OP_DELETE
unit 'u1' before-chained label 'lab1' op OP_DELETE
unit 'u1' before label 'lab2' (chain 'lab1') op OP_DELETE {
unit 'u1' drain label 'lab2' (chain 'lab1') op OP_DELETE
unit 'u1' before-chained label 'lab2' (chain 'lab1') op OP_DELETE
unit 'u1' before label 'lab3' (chain 'lab2') op OP_DELETE {
unit 'u1' drain label 'lab3' (chain 'lab2') op OP_DELETE
unit 'u1' after label 'lab3' (chain 'lab2') op OP_DELETE }
unit 'u1' after label 'lab2' (chain 'lab1') op OP_DELETE }
unit 'u1' before label 'lab3' (chain 'lab1') op OP_DELETE {
unit 'u1' drain label 'lab3' (chain 'lab1') op OP_DELETE
unit 'u1' after label 'lab3' (chain 'lab1') op OP_DELETE }
unit 'u1' after label 'lab1' op OP_DELETE }
```

In non-verbose mode the same trace would be:

```
unit 'u1' before label 'lab1' op OP_INSERT
unit 'u1' before label 'lab2' (chain 'lab1') op OP_INSERT
unit 'u1' before label 'lab3' (chain 'lab2') op OP_INSERT
unit 'u1' before label 'lab3' (chain 'lab1') op OP_INSERT
unit 'u1' before label 'lab1' op OP_DELETE
unit 'u1' before label 'lab2' (chain 'lab1') op OP_DELETE
unit 'u1' before label 'lab3' (chain 'lab2') op OP_DELETE
unit 'u1' before label 'lab3' (chain 'lab1') op OP_DELETE
```

The verbose trace has the "before" and "after" lines marked with the curly braces, so that when it gets loaded into an editor, it can be navigated relatively easily.

The actual contents of the rows is not printed in either case. This is basically because the tracer is implemented in C++, and I've been trying to keep the knowledge of the meaning of the simple data types out of the C++ code as much as possible for now. But it can be implemented with a Perl tracer.

A Perl tracer is created with:

```
$tracer = Triceps::UnitTracerPerl->new($sub, @args)
  or confess "$!";
```

The arguments are a reference to a function, and optionally arguments for it. The resulting tracer can be used in the unit's `setTracer()` as usual.

The function of the Perl tracer gets called as:

```
&$sub($unit, $label, $fromLabel, $rowop, $when, @args)
```

The arguments are:

- `$unit` is the usual unit reference.
- `$label` is the current label being traced.
- `$fromLabel` is the parent label in the chaining (would be `undef` if the current label is called directly, without chaining from anything).
- `$rowop` is the current row operation.
- `$when` is an integer constant showing the point when the tracer is being called. It's value may be one of `&Triceps::TW_BEFORE`, `&Triceps::TW_BEFORE_DRAIN`, `&Triceps::TW_BEFORE_CHAINED`, `&Triceps::TW_AFTER`; the prefix `TW` stands for "tracer when".
- `@args` are the extra arguments passed from the tracer creation.

The `TW_*` constants can as usual be converted to and from strings with the calls

```
$string = &Triceps::tracerWhenString($value);
$value = &Triceps::stringTracerWhen($string);
```

There also are the conversion functions with strings more suitable for the human-readable messages: "before", "drain", "before-chained", "drain". These are actually the conversions used in the UnitTracerStringName. The functions for them are:

```
$string = &Triceps::tracerWhenHumanString($value);
$value = &Triceps::humanStringTracerWhen($string);
```

The Perl tracers allow to execute any arbitrary actions when tracing. They can act as breakpoints by looking for certain conditions and opening a debugging session when those are met.

For an example of a Perl tracer, let's start with a tracer function that works like UnitTracerStringName:

```
sub tracerCb()
{
  my ($unit, $label, $from, $rop, $when, @extra) = @_;
  our $history;

  my $msg = "unit '" . $unit->getName() . "' "
    . Triceps::tracerWhenHumanString($when) . " label '"
    . $label->getName() . "' ";
  if (defined $fromLabel) {
    $msg .= "(chain '" . $fromLabel->getName() . "') ";
  }
  $msg .= "op " . Triceps::opcodeString($rop->getOpcode());
  if ($when == &Triceps::TW_BEFORE) {
    $msg .= " {";
  } elsif ($when == &Triceps::TW_AFTER) {
    $msg .= " }";
  }
  $msg .= "\n";
  $history .= $msg;
}

undef $history;
$ptr = Triceps::UnitTracerPerl->new(\&tracerCb);
$u1->setTracer($ptr);
```

It's slightly different, in the way that it always produces the verbose trace, and that it collectes the trace in the global variable `$history`. But the resulting text is the same as with UnitTracerStringName.

Now let's improve on it by printing the whole rowop contents too. In a “proper” way this advanced tracer would be defined as a class constructing the tracer objects. But to reduce the amount of code let's just make it a standalone function to be used with the Perl tracer constructor.

And for extra nicety let's make the result nicely indented, with two spaces per the indenting level. The indenting is actually not such a great idea: with the long sequences of the calls between the labels, the nesting levels would also be deep, and the output would be indented way off the right end of the screen. That's why it's not done in UnitTracerStringName (though it might be a good idea as an option). But for the small short examples it works well. The function would take 3 extra arguments:

- Verbosity, a boolean value.
- Reference to an array variable where to append the text of the trace. This is more flexible than the fixed `$history`. The array will contain the lines of the trace as its elements. And appending to an array should be more efficient than appending to the end of a potentially very long string.
- Reference to a scalar variable that would be used to keep the indenting level. The value of that variable will be updated as the tracing happens. Its initial value will determine the initial indenting level. Keeping the indenting is actually not easy because the indenting level can be changed for two reasons, the label chaining and the label calls. To get the logic working, one indenting level gets added before in advance, in front of the outermost trace lines. So, to make the outermost lines appear with no indenting, initialize this variable to -1 and not 0.

```
sub traceStringRowop
{
  my ($unit, $label, $fromLabel, $rowop, $when,
    $verbose, $rlog, $rnest) = @_;

  if ($verbose) {
    ${$rnest}++ if ($when == &Triceps::TW_BEFORE);
    ${$rnest}-- if ($when == &Triceps::TW_AFTER);
  } else {
    return if ($when != &Triceps::TW_BEFORE);
  }


  my $msg =  "unit '" . $unit->getName() . "' "
    . Triceps::tracerWhenHumanString($when) . " label '"
    . $label->getName() . "' ";
  if (defined $fromLabel) {
    $msg .= "(chain '" . $fromLabel->getName() . "') ";
  }
  my $tail = "";
  if ($when == &Triceps::TW_BEFORE) {
    $tail = " {";
  } elsif ($when == &Triceps::TW_AFTER) {
    $tail = " }";
  }
  push (@{$rlog}, ("  " x ${$rnest}) . $msg . "op "
    . $rowop->printP() . $tail);

  if ($verbose) {
    ${$rnest}++ if ($when == &Triceps::TW_BEFORE);
    ${$rnest}-- if ($when == &Triceps::TW_AFTER);
  }
}

undef @history;
my $tnest =  -1; # keeps track of the tracing nesting level
$ptr = Triceps::UnitTracerPerl->new(\&traceStringRowop, 1, \@history, \$tnest);
$u1->setTracer($ptr);
```

For the same call sequence as before, the output will be as follows (I've tried to wrap the long lines in a logically consistent way but it still spoils the effect of indenting a bit):

```
unit 'u1' before label 'lab1' op lab1 OP_INSERT a="123" b="456"
    c="789" d="3.14" e="text"  {
  unit 'u1' drain label 'lab1' op lab1 OP_INSERT a="123" b="456"
      c="789" d="3.14" e="text"
  unit 'u1' before-chained label 'lab1' op lab1 OP_INSERT a="123"
      b="456" c="789" d="3.14" e="text"
    unit 'u1' before label 'lab2' (chain 'lab1') op lab1 OP_INSERT
        a="123" b="456" c="789" d="3.14" e="text"  {
      unit 'u1' drain label 'lab2' (chain 'lab1') op lab1 OP_INSERT
          a="123" b="456" c="789" d="3.14" e="text"
      unit 'u1' before-chained label 'lab2' (chain 'lab1') op lab1
          OP_INSERT a="123" b="456" c="789" d="3.14" e="text"
        unit 'u1' before label 'lab3' (chain 'lab2') op lab1 OP_INSERT
            a="123" b="456" c="789" d="3.14" e="text"  {
          unit 'u1' drain label 'lab3' (chain 'lab2') op lab1
              OP_INSERT a="123" b="456" c="789" d="3.14" e="text"
        unit 'u1' after label 'lab3' (chain 'lab2') op lab1 OP_INSERT
            a="123" b="456" c="789" d="3.14" e="text"  }
    unit 'u1' after label 'lab2' (chain 'lab1') op lab1 OP_INSERT
        a="123" b="456" c="789" d="3.14" e="text"  }
    unit 'u1' before label 'lab3' (chain 'lab1') op lab1 OP_INSERT
        a="123" b="456" c="789" d="3.14" e="text"  {
      unit 'u1' drain label 'lab3' (chain 'lab1') op lab1 OP_INSERT
          a="123" b="456" c="789" d="3.14" e="text"
    unit 'u1' after label 'lab3' (chain 'lab1') op lab1 OP_INSERT
        a="123" b="456" c="789" d="3.14" e="text"  }
unit 'u1' after label 'lab1' op lab1 OP_INSERT a="123" b="456" c="789"
    d="3.14" e="text"  }
unit 'u1' before label 'lab1' op lab1 OP_DELETE a="123" b="456"
    c="789" d="3.14" e="text"  {
  unit 'u1' drain label 'lab1' op lab1 OP_DELETE a="123" b="456"
      c="789" d="3.14" e="text"
  unit 'u1' before-chained label 'lab1' op lab1 OP_DELETE a="123"
      b="456" c="789" d="3.14" e="text"
    unit 'u1' before label 'lab2' (chain 'lab1') op lab1 OP_DELETE
        a="123" b="456" c="789" d="3.14" e="text"  {
      unit 'u1' drain label 'lab2' (chain 'lab1') op lab1 OP_DELETE
```

```
          a="123" b="456" c="789" d="3.14" e="text"
      unit 'u1' before-chained label 'lab2' (chain 'lab1') op lab1
          OP_DELETE a="123" b="456" c="789" d="3.14" e="text"
        unit 'u1' before label 'lab3' (chain 'lab2') op lab1 OP_DELETE
            a="123" b="456" c="789" d="3.14" e="text"  {
          unit 'u1' drain label 'lab3' (chain 'lab2') op lab1
              OP_DELETE a="123" b="456" c="789" d="3.14" e="text"
        unit 'u1' after label 'lab3' (chain 'lab2') op lab1 OP_DELETE
            a="123" b="456" c="789" d="3.14" e="text"  }
    unit 'u1' after label 'lab2' (chain 'lab1') op lab1 OP_DELETE
        a="123" b="456" c="789" d="3.14" e="text"  }
    unit 'u1' before label 'lab3' (chain 'lab1') op lab1 OP_DELETE
        a="123" b="456" c="789" d="3.14" e="text"  {
      unit 'u1' drain label 'lab3' (chain 'lab1') op lab1 OP_DELETE
          a="123" b="456" c="789" d="3.14" e="text"
    unit 'u1' after label 'lab3' (chain 'lab1') op lab1 OP_DELETE
        a="123" b="456" c="789" d="3.14" e="text"  }
unit 'u1' after label 'lab1' op lab1 OP_DELETE a="123" b="456" c="789"
    d="3.14" e="text"  }
```

As mentioned before, each label produces two levels of indenting: one for everything after “before”, another one for the nested labels.

Eventually this tracing should become another standard class in Triceps.

Chapter 8. Memory Management

8.1. Reference cycles

Remember that the Triceps memory management uses the reference counting, which does not like the reference cycles, as has been mentioned in Section 4.3: “Memory management fundamentals” (p. 21) . The reference cycles cause the objects to be never freed. It's no big deal if the data structures exist until the program exit anyway but it becomes a memory leak if they keep being created and deleted dynamically.

The problems come not with the data that goes through the models but with the models themselves. The data gets reference-counted without any issues. The reference cycles can get formed only between the elements of the models: labels, tables etc. If you don't need them destroyed until the program exits (or more exactly, until the Perl interpreter instance exits), there is no problem. The leaks could happen only if the model elements get created and destroyed as the program runs, such as if you use them to parse and process the short-lived ad-hoc queries.

These leaks are pretty hard to diagnose. There are some packages, like Devel::Cycle, but they won't detect the loops that involve a reference at C++ level. And when the Perl interpreter exits, it clears up all the variables used, even the ones involved in the loops, so if you run it under valgrind, valgrind doesn't show any leaks. There is a package Devel::LeakTrace that should be able to detect all these left-over variables. However I can't tell for sure yet, so far I haven't had enough patience to build all the dependencies for it.

One possibility is to use the weak references (using the module Scalar::Util). But the problem is that you need to not forget weakening the references manually. Too much work, too much attention, too easy to forget.

The mechanism used in Triceps works by breaking up the reference cycles when the data needs to be cleared. The execution unit keeps track of all its labels, and when it gets destoryed, clears them up, breaking up the cycles. It's also possible to clear the labels individually, by a manual call.

The clearing of a label clears all the chainings. The chained labels get cleared too in their turn, and eventually the whole chain clears up. This removes the links in the forward direction, and if any cycles were present, they become open. More on the details of label clearing in the Section 8.2: “Clearing of the labels” (p. 70) .

Another potential for reference cycles is between the execution unit and the labels. A unit keeps a reference to all its labels. So the labels can not keep a reference to the unit. And they don't. Internally they have a plain C++ pointer to the unit. However the Perl level may present a problem.

In many cases the labels have a Perl reference to the template object where they belong. And that object is likely to have a Perl reference to the unit. It's one more opportunity for the reference cycle. This code usually looks like this:

```
package MyTemplate;

sub new # ($class, $unit, $name, $rowType, ...)
{
  my $class = shift;
  my $unit = shift;
  my $name = shift;
  my $rowType = shift;
  my $self = {};

  ...

  $self->{unit} = $unit;
  $self->{inputLabel} = $unit->makeLabel($rowType, $name . ".in",
    sub { ... }, sub { ... }, $self);

  ...
  bless $self, $class;
  return $self;
}
```

So the unit refers to the label at the C++ level, the label has a `$self` reference to the Perl object that owns it, and the object's `$self->{unit}` refers back to the unit. Once the label clearing happens, the link from the unit will disappear and the cycle would unroll. But the clearing would not happen by itself because the unit can't get automatically defererenced and destroyed.

Because of this, the unit provides an explicit way to trigger the clearing:

```
$unit->clearLabels();
```

If you want to get rid of an execution unit with all its components without exiting the whole program, use this call. It will start the chain reaction of destruction. Of course, don't forget to undefine all the other references in your program to these objects being destroyed.

There is also a way to trigger this chain reaction automatically. It's done with a helper object that is created as follows:

```
my $clearUnit = $unit->makeClearingTrigger();
```

When the reference to `$clearUnit` gets destroyed, it will call `$unit->clearLabels()` and trigger the destruction of the whole unit. Obviously, don't copy the `$clearUnit` variable, keep it on one place.

If you put it into a block variable, the unit will get destroyed on exiting the block. If you put it into a global variable in a thread, the unit will get destroyed when the thread exits (though I'm a bit hazy on the Perl memoery management with threads yet, it might get all cleared by itself without any special tricks too).

8.2. Clearing of the labels

To remind, a label that executes the Perl code is created with:

```
$label = $unit->makeLabel($rowType, "name", \&clearSub,
  \&execSub, @args);
```

The function `clearSub` deals with the destruction.

The clearing of a label drops all the references to `execSub`, `clearSub` and arguments, and clears all the chainings. And of course the chained labels get cleared too. But before anything else is done, `clearSub` gets a chance to execute and clear any application-level data. It gets as its arguments all the arguments from the label constructor, same as `execSub`:

```
clearSub($label, @args)
```

A typical case is to keep the state of a stateful element in a hash:

```
package MyTemplate;

sub new # ($class, $unit, $name, $rowType, ...)
{
  my $class = shift;
  my $unit = shift;
  my $name = shift;
  my $rowType = shift;
  my $self = {};

  ...

  $self->{unit} = $unit;
  $self->{inputLabel} = $unit->makeLabel($rowType, $name . ".in",
    \&clear, \&handle, $self);
  ...

  bless $self, $class;
  return $self;
}
```

These elements may end up pointing to the other elements. It's fairly common to keep the pointers to the other elements (especially tables) that provide inputs to this one. In general, these references "up" should be safe because the clearing of the labels would destroy the references "down" and open the cycles. But the way things get connected in the heat of the moment, you never know. It's better to be safe than sorry. To be on the safe side, the clearing function can wipe out the whole state of the element by undefining its hash:

```
sub clear # ($label, $self)
{
  my ($label, $self) = @_;
  undef %$self;
}
```

The whole contents of the hash becomes lost, all the refrences from it disappear. And if you use this approach in every object, the complete destruction reigns and everything is nicely laid to waste.

Writing these clear methods for each class quickly becomes tedious and easy to forget. Triceps is a step ahead: it provides a ready function `Triceps::clearArgs()` that does all this destruction. It can undefine the contents of various things passed as its arguments, and then also undefines these arguments themselves. Just reuse it:

```
$self->{inputLabel} = $unit->makeLabel($rowType, $name . ".in",
  \&Triceps::clearArgs, \&handle, $self);
```

But that's not all. Triceps is actually *two* steps ahead. If the `clearSub` is specified as `undef`, Triceps automatically treats it to be `Triceps::clearArgs()`. The last snippet and the following one are equivalent:

```
$self->{inputLabel} = $unit->makeLabel($rowType, $name . ".in",
  undef, \&handle, $self);
```

No need to think, the default will do the right thing for you. Of course, if by some reason you don't want this destruction to happen, you'd have to override it with an empty function "`sub {}`".

8.3. The clearing labels

Some templates don't have their own input labels, instead they just combine and tie together a few internal objects, and use the input labels of some of these internal objects as their inputs. Among the templates included with Triceps, JoinTwo is one of them, it just combines two LookupJoins. Without an input label, there would be no clearing, and the template object would never get undefined.

This can be solved by creating an artificial label that is not connected anywhere and has no code to execute. Its only purpose in life would be to clear the object when told so. To make life easier, rather than abusing `makeLabel()`, there is a way to create the special clearing-only labels:

```
$lb = $unit->makeClearingLabel("name", @args);
```

The arguments would be the references to the objects that need clearing, usually `$self`. For a concrete usage example, here is how JoinTwo uses it:

```
$self->{clearingLabel} = $self->{unit}->makeClearingLabel(
  $self->{name} . ".clear", $self);
```

Since this call "should never fail", on any errors it will confess. There is no need to check the result. The result can be saved in a variable or can be simply ignored. If you throw away the result, you won't be able to access that label from the Perl code but it won't be lost: it will be still referenced from the unit, until the unit gets cleared.

Note how the clearing label doesn't have a row type. In reality every label does how a row type, just it would be silly to abuse the random row types to create the clearing-only labels. Because of this, the clearing labels are created with a special empty row type that has no fields in it. If you ever want to use this row type for any other purposes, you can get it with the method

```
$rt = $unit->getEmptyRowType();
```

Under the hood, the clearing label is the same as a normal label with Perl code, only with the special default values used for its construction. The normal Perl label methods would work on it like on a normal label.

Chapter 9. Tables

9.1. Hello, tables!

The tables are the fundamental elements of state-keeping in Triceps. Let's start with a basic example:

```
my $hwunit = Triceps::Unit->new("hwunit") or confess "$!";
my $rtCount = Triceps::RowType->new(
  address => "string",
  count => "int32",
) or confess "$!";

my $ttCount = Triceps::TableType->new($rtCount)
  ->addSubIndex("byAddress",
    Triceps::IndexType->newHashed(key => [ "address" ])
  )
or confess "$!";
$ttCount->initialize() or confess "$!";

my $tCount = $hwunit->makeTable($ttCount, &Triceps::EM_CALL, "tCount")
  or confess "$!";

while(<STDIN>) {
  chomp;
  my @data = split(/\W+/);

  # the common part: find if there already is a count for this address
  my $rhFound = $tCount->findBy(
    address => $data[1]
  );
  my $cnt = 0;
  if (!$rhFound->isNull()) {
    $cnt = $rhFound->getRow()->get("count");
  }

  if ($data[0] =~ /^hello$/i) {
    my $new = $rtCount->makeRowHash(
      address => $data[1],
      count => $cnt+1,
    ) or confess "$!";
    $tCount->insert($new);
  } elsif ($data[0] =~ /^count$/i) {
    print("Received '", $data[1], "' ", $cnt + 0, " times\n");
  } else {
    print("Unknown command '$data[0]'\n");
  }
}
```

What happens here? The main loop reads the lines from standard input, splits into words and uses the first word as a command and the second word as a key. Note that it's not CSV format, it's words with the non-alphanumeric characters separating the words. “Hello, table!”, “hello world”, “count world” are examples of the valid inputs. For someting different, the commands are compared with their case ignored (but the case matters for the key).

The example counts, how many times each key has been `hello`-ed, and prints this count back on the command `count`. Here is a sample, with the input lines printed in bold:

```
Hello, table!
Hello, world!
```

```
Hello, table!
count world
Received 'world' 1 times
Count table
Received 'table' 2 times
```

In this example the table is read and modified using the direct procedural calls. As you can see, there isn't even any need for unit scheduling and such. There is a scheduler-based interface to the tables too, it will be shown soon. But in many cases the direct access is easier. Indeed, this particular example could have been implemented with the plain Perl hashes. Nothing wrong with that either. Well, the Perl tables provide many more intersting ways of indexing the data. But if you don't need them, they don't matter. And at some future point the tables will be supporting the on-disk persistence, but no reason to bother much about that now: things are likely to change a dozen times yet before that happens. Feel free to just use the Perl data structures if they make the code easier.

A table is created through a table type. This allows to stamp out duplicate tables of the same type, which can get handy when the multithreading will be added. A table is local to a thread. A table type can be shared between threads. To look up something in another thread's table, you'd either have to ask it through a request-reply protocol or to keep a local copy of the table. Such a copy can be easily done by creating a copy table from the same type.

In reality, right now all the business with table types separated from the tables is more pain than gain. It not only adds extra steps but also makes difficult to define a template that acts on a table by defining extra features on it. Something will be done about it, I have a few ideas.

The table type gets first created and configured, then initialized. After a table type is initialized, it can not be changed any more. That's the point of the initialization call: tell the type that all the configuration has been done, and it can go immutable now. Fundamentally, configuting a table type just makes it collect bits and pieces. Nothing but the most gross errors can be detected at that point. At initialization time everything comes together and everything gets checked for consistency. A table type must be fully initialized in one thread before it can be shared with other threads. The historic reason for this API is that it mirrors the C++ API, which has turned out not to look that good in Perl. It's another candidate for a change.

A table type gets the row type and at least one index. Here it's a hashed index by the key field `address`. "Hashed" means that you can look up the rows by the key value but there are no promises about any specific row order. And the hashing is used to make the key comparisons more efficient. The key of a hashed index may consist of multiple fields.

The table is then created from the table type, enqueueing mode (another hold-over, just always use `EM_CALL`, this argument will be removed in the future), and given a name.

The rows can then be inserted into the table (and removed too, not shown in this example yet). The default behavior of the hashed index is to replace the old row if a new row with the same key is inserted.

The search in the table is done by the method `findBy()` with the key fields of the index. Which returns a RowHandle object. A RowHandle is essentially an iterator in the table. Even if the row is not found, a RowHandle will be still returned but it will be NULL, which is checked for by `$rh->isNull()`.

No matter which command will be used, it's always useful to look up the previous row for the key: its contents would be either printed or provide the previous value for the increase. So the model does it first and gets the count from it. If it's not found, then the count is set to 0.

Then it looks at the command and does what it's been told. Updating the count amounts to creating a new row with the new values and inserting it into the table. It replaces the previous one.

This is just the tip of the iceberg. The tables in Triceps have a lot more features.

9.2. Tables and labels

A table does not have to be operated in a procedural way. It can be plugged into the the scheduler machinery. Whenever a table is created, three labels are created with it.

- The input label is for sending the modification rowops to the table. The table provides the handler for it that applies the incoming rowops to the table.
- The output label propagates the modifications done to the table. It is a dummy label, and does nothing by itself. It's there for chaining the other labels to it. The output rowop comes quite handy to propagate the table's modifications to the rest of the state.
- The pre-modification label is also a dummy label, for chaining other labels to it. It sends the rowops right before they are applied to the table. This comes very handy for the elements that need to act depending on the previous state of the table, such as joins. The pre-modification label doesn't simply mirror the input label. The rows received on the input label may trigger the automatic changes to the table, such as an old row being deleted when a new row with the same key is inserted. All these modifications, be they automatic or explicit, will be reported to the pre-modification label. Since the pre-modification label is used relatively rarely, it contains a special optimization: if there is no label chained to it, no rowop will be sent to it in the first place. Don't be surprised if you enable the tracing and don't see it in the trace.

Again, the rowops coming through these labels aren't necessarily the same. If a DELETE rowop comes to the input label, referring to a row that is not in the table, it will not propagate anywhere. If an INSERT rowop comes in and causes another row to be replaced, the replaced row will be sent to the pre-modification and output labels as a DELETE rowop first.

Anf of course the table may be modified through the procedural interface. These modifications also produce rowops on the pre-modification and output labels.

The labels of the table have names. They are produced by adding suffixes to the table name. They are "tablename.in", "tablename.pre" and "tablename.out".

In the “no bundling” spirit, a rowop is sent to the pre-modification label right before it's applied to the table, and to the output label right after it's applied. If the labels executed from there need to read the table, they can, and will find the table in the exact state with no intervening modifications. However, they can't modify the table neither directly nor by calling its input label. When these labels are called, the table is in the middle of a modification and it can't accept another one. Such attempts are treated as recursive modifications, forbidden, and the program will die on them. If you need to modify the table, use `schedule()` or `loopAt()` to have the next modification done later. However there are no guarantees about other modifications getting done in between. When the looped rowop executes, it might need to check the state of the table again and decide if its operation still makes sense.

So, let's make a version of “Hello, table” example that passes the modification requests as rowops through the labels. It will print the information about the updates to the table as they happen, so there is no more use having a separate command for that. But for another demonstration let's add a command that would clear the counter of hellos. Here is its code:

```
my $hwunit = Triceps::Unit->new("hwunit") or confess "$!";
my $rtCount = Triceps::RowType->new(
  address => "string",
  count => "int32",
) or confess "$!";

my $ttCount = Triceps::TableType->new($rtCount)
  ->addSubIndex("byAddress",
    Triceps::IndexType->newHashed(key => [ "address" ])
  )
or confess "$!";
$ttCount->initialize() or confess "$!";

my $tCount = $hwunit->makeTable($ttCount, "EM_CALL", "tCount") or confess "$!";

my $lbPrintCount = $hwunit->makeLabel($tCount->getRowType(),
  "lbPrintCount", undef, sub { # (label, rowop)
    my ($label, $rowop) = @_;
    my $row = $rowop->getRow();
    print(&Triceps::opcodeString($rowop->getOpcode), " '",
      $row->get("address"), "', count ", $row->get("count"), "\n");
  } ) or confess "$!";
$tCount->getOutputLabel()->chain($lbPrintCount) or confess "$!";

# the updates will be sent here, for the tables to process
my $lbTableInput = $tCount->getInputLabel();

while(<STDIN>) {
  chomp;
  my @data = split(/\W+/);

  # the common part: find if there already is a count for this address
  my $rhFound = $tCount->findBy(
    address => $data[1]
  );
  my $cnt = 0;
  if (!$rhFound->isNull()) {
    $cnt = $rhFound->getRow()->get("count");
  }

  if ($data[0] =~ /^hello$/i) {
    $hwunit->makeHashSchedule($lbTableInput, "OP_INSERT",
      address => $data[1],
      count => $cnt+1,
    );
  } elsif ($data[0] =~ /^clear$/i) {
    $hwunit->makeHashSchedule($lbTableInput, "OP_DELETE",
      address => $data[1]
    );
  } else {
    print("Unknown command '$data[0]'\n");
  }
  $hwunit->drainFrame();
}
```

The table creation is the same as last time. The only difference is that it uses `"EM_CALL"` instead of `&Triceps::EM_CALL`, both being equivalent. The row finding in the table is also the same.

The printing of the modifications to the table is done with `$lbPrintCount`, which is connected to the table's output label. It prints the opcode, the address of the greeting, and the count of greetings. It will show us what is happening to the table as soon as it happens. An unit trace could be used instead but a custom printout contains less noise. The pre-modification label is of no interest here, so it's not used.

The references to the labels of a table are gotten with:

```
$label = $table->getInputLabel();
$label = $table->getPreLabel();
$label = $table->getOutputLabel();
```

The deletion does not require an exact row to be sent in. All it needs is a row with the keys for deletion, the rest of the fields in it are ignored. So the “clear” command puts only the key field in it.

Here is an example of input (in bold) and output:

```
Hello, table!
OP_INSERT 'table', count 1
Hello, world!
OP_INSERT 'world', count 1
Hello, table!
OP_DELETE 'table', count 1
```

```
OP_INSERT 'table', count 2
clear, table
OP_DELETE 'table', count 2
Hello, table!
OP_INSERT 'table', count 1
```

An interesting thing happens after the second “Hello, table!”: the code send only an `OP_INSERT` but the output shows an `OP_DELETE` and `OP_INSERT`. The `OP_DELETE` for the old row gets automatically generated when a row with repeated key is inserted. Now, depending on what you want, just sending in the first place the consequent inserts of rows with the same keys, and relying on the table's internal consistency to turn them into updates, might be a good thing or not. Overall it's a dirty way to write but sometimes it comes convenient. The clean way is to send the explicit deletes first. When the data goes through the table, it gets automatically cleaned. The subscribers to the table's output and pre-modification labels get the clean and consistent picture: a row never gets simply replaced, they always see an `OP_DELETE` first and only then an `OP_INSERT`.

9.3. Basic iteration through the table

Let's add a dump of the table contents to the "Hello, table" example, either version of it. For that, the code needs to go through every record in the table:

```
  elsif ($data[0] =~ /^dump$/i) {
    for (my $rhi = $tCount->begin(); !$rhi->isNull(); $rhi = $rhi->next()) {
      print($rhi->getRow->printP(), "\n");
    }
  }
```

As you can see, the row handle works kind of like an STL iterator. Only the end of iteration is detected by receiving a NULL row handle. Calling `next()` on a NULL row handle is OK but it would just return another NULL handle. And there is no decrementing the iterator, you can only go forward with `next()`. The backwards iteration is in the plans but not implemented yet.

An example of this fragment's output would be:

```
Hello, table!
Hello, world!
Hello, table!
count world
Received 'world' 1 times
Count table
Received 'table' 2 times
dump
address="world" count="1"
address="table" count="2"
```

The order of the rows in the printout is the same as the order of rows in the table's index. Which is no particular order, since it's a hashed index. As long as you stay with the same 64-bit AMD64 architecture (with LSB-first byte order), it will stay the same on consecutive runs. But switching to a 32-bit machine or to an MSB-first byte order (such as a SPARC, if you can still find one) will change the hash calculation, and with it the resulting row order. There are the ordered indexes as well, they will be described later.

9.4. Deleting a row

Deleting a row from a table through the input label is simple: send a rowop with `OP_DELETE`, it will find the row with the matching key and delete it, as was shown above. In the procedural way the same can be done with the method `deleteRow()`. The added row deletion code for the main loop of “Hello, table” (either version, but particularly relevant for the one from Section 9.1: “Hello, tables!” (p. 73)) is:

```
  elsif ($data[0] =~ /^delete$/i) {
    my $res = $tCount->deleteRow($rtCount->makeRowHash(
      address => $data[1],
    ));
    print("Address '", $data[1], "' is not found\n") unless $res;
  }
```

The result allows to differentiate between the situations when the row was found and deleted and the row was not found. On any error the call confesses. The `insert()` method also follows this new convention and confesses on errors.

However we already find the row handle in advance in `$rhFound`. For this case a more efficient form is available, and it can be added to the example as:

```
    elsif ($data[0] =~ /^remove$/i) {
      if (!$rhFound->isNull()) {
        $tCount->remove($rhFound);
      } else {
        print("Address '", $data[1], "' is not found\n");
      }
    }
```

It removes a specific row handle from the table. In whichever way you find it, you can remove it. An attempt to remove a NULL handle would be an error (and this method also confesses on errors).

The reason why `remove()` is more efficient than `deleteRow()` is that `deleteRow()` amounts to finding the row handle by key and then removing it. And the `OP_DELETE` rowop sent to the input label calls `deleteRow()`.

`deleteRow()` never deletes more than one row, even if multiple rows match (yes, the indexes don't have to be unique). There isn't any method to delete multiple rows at once. Every row has to be deleted by itself. As an example, here is the implementation of the command “clear” for “Hello, table” that clears all the table contents by iterating through it:

```
  elsif ($data[0] =~ /^clear$/i) {
    my $rhi = $tCount->begin();
    while (!$rhi->isNull()) {
      my $rhnext = $rhi->next();
      $tCount->remove($rhi);
      $rhi = $rhnext;
    }
  }
```

After a handle is removed from the table, it continues to exist, as long as there are references to it. It could even be inserted back into the table. However until (and unless) it's inserted back, it can not be used for iteration any more. Calling `next()` on a handle that is not in the table would just return a NULL handle. So the next row has to be found before removing the current one.

9.5. A closer look at the RowHandles

A few uses of the RowHandles have been shown by now. So, what is a RowHandle? As Captain Obvious would say, RowHandle is a class (or package, in Perl terms) implementing a row handle.

A row handle keeps a table's service information (including the index data) for a single data row, including of course a reference to the row itself. Each row is stored in the table through its handle. The row handle is also an iterator in the table, and a special one: it's an iterator for *all* the table's indexes at once. For you SQLy people, an iterator is essentially a cursor on an index. For you Java people, an iterator can be used to do more than step sequentially through rows. So far only the table types with one index have been shown, but in reality multiple indexes are supported, potentially with quite complicated arrangements. More on the indexes later, for now just keep it in mind. A row handle can be found through one index and then used to iterate through another one. Or you can iterate through one index, find a certain row handle and continue iterating through another index starting from that handle. If you remember a reference on a particular row handle, you can always continue iteration from that point later. (unless the row handle gets removed from the table).

A RowHandle always belongs to a particular table, the RowHandles can not be shared nor moved between two tables, even if the tables are of the same type. Since the tables are single-threaded, obviously the RowHandles may not be shared between the threads either.

However a RowHandle may exist without being inserted into a table. In this case it still has a spiritual connection to that table but is not included in the index (the iteration attempts with it would just return “end of the index”), and will be destroyed as soon as all the references to it disappear.

The insertion of a row into a table actually happens in two steps:

1. A RowHandle is created for a row.

2. This new handle is inserted into the table.

This is done with the following code:

```
$rh = $table->makeRowHandle($row) or confess "$!";
$table->insert($rh);
```

Only it just so happens that to make life easier, the method `insert()` has been made to accept either a row handle or directly a row. If it finds a row, it makes a handle for it behind the curtains and then proceeds with the insertion of that handle. Passing a row directly is also more efficient (if you don't have a handle already created for it for some other reason) because the row handle creation then happens entirely in the C++ code, without surfacing into Perl.

A handle can be created for any row of a type matching the table's row type. For a while it was accepting only equal types but that was not consistent with what the labels are doing, so I've changed it.

The method `insert()` has a return value. It's often ignored but occasionally comes handy. 1 means that the row has been inserted successfully, and 0 means that the row has been rejected. On errors it confesses. An attempt to insert a NULL handle or a handle that is already in the table will cause a rejection, not an error. Also the table's index may reject a row with duplicate key (though right now this option is not implemented, and the hash index silently replaces the old row with the new one).

There is a method to find out if a row handle is in the table or not:

```
$result = $rh->isInTable();
```

Though it's used mostly for debugging, when some strange things start going on.

The searching for rows in the table by key has been previously shown with the method `findBy()`. Which happens to be a wrapper over a more general method `find()`: it constructs a row from its argument fields and then calls `find()` with that row as a sample of data to find. The method `find()` is similar to `insert()` in the handling of its arguments: the “proper” way is to give it a row handle argument, but the more efficient way is to give it a row argument, and it will create the handle for it as needed before performing a search.

Now you might wonder: huh, `find()` takes a row handle and returns a row handle? What's the point? Why not just use the first row handle? Well, those are different handles:

- The argument handle is normally not in the table. It's created brand new from a row that contains the keys that you want to find, just for the purpose of searching.

- The returned handle is always in the table (of course, unless it's NULL). It can be further used to extract back the row data, and/or for iteration.

Though nothing really prevents you from searching for a handle that is already in the table. You'll just get back the same handle, after gratuitously spending some CPU time. (There are exceptions to this, with the more complex indexes that will be described later).

Why do you need to create new a row handle just for the search? Due to the internal mechanics of the implementation. A handle stores the helper information for the index. For example, the hash index calculates the hash value of all the row's key fields once and stores it in the row handle. Despite it being called a hash index, it really stores the data in a tree, with the hash value used to speed up the comparisons for the tree order. It's much easier to make both the `insert()` and `find()` work with the hash value and row reference stored in the same way in a handle than to implement them differently. Because of this, `find()` uses the exactly same row handle argument format as `insert()`.

Can you create multiple row handles referring to the same row? Sure, knock yourself out. From the table's perspective it's the same thing as multiple row handles for multiple copies of the row with the same values in them, only using less memory.

There is more to the row handles than has been touched upon yet. It will all be revealed when more of the table features are described. The internal structure of the row handles will be described in the Section 9.10: “The index tree” (p. 93) .

9.6. A window is a FIFO

A fairly typical situation in the CEP world is when a model needs to keep a limited history of events. For a simple example, let's discuss, how to remember the last two trades per stock symbol. The size of two has been chosen to keep the sample input and outputs small.

This is normally called a window logic, with a sliding window. You can think of it in a mechanical analogy: as the trades become available, they get printed on a long tape. However the tape is covered with a masking plate. The plate has a window cut in it that lets you see only the last two trades.

Some CEP systems have the special data structures that implement this logic, that are called windows. Triceps has a feature on a table instead that makes a table work as a window. It's not unique in this department: for example Coral8 does the opposite, calls everything a window, even if some windows are really tables in every regard but name.

Here is a Triceps example of keeping the window for the last two trades and iteration over it:

```
our $uTrades = Triceps::Unit->new("uTrades") or confess "$!";
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
) or confess "$!";

our $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
      ->addSubIndex("last2",
        Triceps::IndexType->newFifo(limit => 2)
      )
  )
  or confess "$!";
$ttWindow->initialize() or confess "$!";
our $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

# remember the index type by symbol, for searching on it
our $itSymbol = $ttWindow->findSubIndex("bySymbol") or confess "$!";
# remember the FIFO index, for finding the start of the group
our $itLast2 = $itSymbol->findSubIndex("last2") or confess "$!";

# print out the changes to the table as they happen
```

```
our $lbWindowPrint = $uTrades->makeLabel($rtTrade, "lbWindowPrint",
  undef, sub { # (label, rowop)
    print($_[1]->printP(), "\n"); # print the change
  }) or confess "$!";
$tWindow->getOutputLabel()->chain($lbWindowPrint) or confess "$!";

while(<STDIN>) {
  chomp;
  my $rTrade = $rtTrade->makeRowArray(split(/,/)) or confess "$!";
  my $rhTrade = $tWindow->makeRowHandle($rTrade) or confess "$!";
  $tWindow->insert($rhTrade);
  # There are two ways to find the first record for this
  # symbol. Use one way for the symbol AAA and the other for the rest.
  my $rhFirst;
  if ($rTrade->get("symbol") eq "AAA") {
    $rhFirst = $tWindow->findIdx($itSymbol, $rTrade);
  } else  {
    # $rhTrade is now in the table but it's the last record
    $rhFirst = $rhTrade->firstOfGroupIdx($itLast2) or confess "$!";
  }
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2) or confess "$!";
  print("New contents:\n");
  for (my $rhi = $rhFirst;
      !$rhi->same($rhEnd); $rhi = $rhi->nextIdx($itLast2)) {
    print("  ", $rhi->getRow()->printP(), "\n");
  }
}
```

This example reads the trade records in CSV format, inserts them into the table, and then prints the actual modifications reported by the table and the new state of the window for this symbol. And here is a sample log, with the input lines in bold:

```
1,AAA,10,10
tWindow.out OP_INSERT id="1" symbol="AAA" price="10" size="10"
New contents:
  id="1" symbol="AAA" price="10" size="10"
2,BBB,100,100
tWindow.out OP_INSERT id="2" symbol="BBB" price="100" size="100"
New contents:
  id="2" symbol="BBB" price="100" size="100"
3,AAA,20,20
tWindow.out OP_INSERT id="3" symbol="AAA" price="20" size="20"
New contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
4,BBB,200,200
tWindow.out OP_INSERT id="4" symbol="BBB" price="200" size="200"
New contents:
  id="2" symbol="BBB" price="100" size="100"
  id="4" symbol="BBB" price="200" size="200"
5,AAA,30,30
tWindow.out OP_DELETE id="1" symbol="AAA" price="10" size="10"
tWindow.out OP_INSERT id="5" symbol="AAA" price="30" size="30"
New contents:
  id="3" symbol="AAA" price="20" size="20"
  id="5" symbol="AAA" price="30" size="30"
6,BBB,300,300
tWindow.out OP_DELETE id="2" symbol="BBB" price="100" size="100"
tWindow.out OP_INSERT id="6" symbol="BBB" price="300" size="300"
New contents:
  id="4" symbol="BBB" price="200" size="200"
  id="6" symbol="BBB" price="300" size="300"
```

You can see that the window logic works: at no time is there more than two rows in each group. As more rows are inserted, the oldest rows get deleted.

Now let's dig into the code. The first thing to notice is that the table type has two indexes (strictly speaking, index types, but most of the time they can be called indexes without creating a confusion) in it. Unlike your typical database, the indexes in this example are nested.

```
TableType
+-IndexType Hash "bySymbol"
  +-IndexType Fifo "last2"
```

If you follow the nesting, you can see, that the first call `addSubIndex()` adds an index type to the table type, while the textually second `addSubIndex()` adds an index to the previous index.

The same can also be written out in multiple separate calls, with the intermediate results stored in the variables:

```
$itLast2 = Triceps::IndexType->newFifo(limit => 2);
$itSymbol = Triceps::IndexType->newHashed(key => [ "symbol" ]);
$itSymbol->addSubIndex("last2", $itLast2);
$ttWindow = Triceps::TableType->new($rtTrade);
$ttWindow->addSubIndex("bySymbol", $itSymbol);
```

I'm not perfectly happy with the way the table types are constructed with the index types right now, since the parenthesis levels have turned out a bit hard to track. This is another example of following the C++ API in Perl that didn't work out too well, and it will change in the future. But for now please bear with it.

The index nesting is kind of intuitively clear, but the details may take some time to get your head wrapped around them. You can think of it as the inner index type creating the miniature tables that hold the rows, and then the outer index holding not individual rows but those miniature tables. So, to find the rows in the table you go through two levels of indexes: first through the outer index, and then through the inner one. The table takes care of these details and makes them transparent, unless you want to stop your search at an intermediate level: such as, to find *all* the transactions with a given symbol, you need to do a search in the outer index, but then from that point iterate through all rows in the found inner index. For this you obviously have to tell the table, where do you want to stop in the search.

The outer index is the hash index that we've seen before, the inner index is a FIFO index. A FIFO index doesn't have any key, it just keeps the rows in the order they were inserted. You can search in a FIFO index but most of the time it's not the best idea: since it has no keys, it searches linearly through all its rows until it finds an exact match (or runs out of rows). It's a reasonable last-resort way but it's not fast and in many cases not what you want. This also sends a few ripples through the row deletion. Remember that the method `deleteRow()` and sending the `OP_DELETE` to the table's input label invoke `find()`, which would cause the linear search on the FIFO indexes. So when you use a FIFO index, it's usually better to find the row handle you want to delete in some other way and then call `remove()` on it, or use another approach that will be shown later. Or just keep inserting the rows and never delete them, like this example does.

A FIFO index may contain multiple copies of an exact same row. It doesn't care, it just keeps whatever rows were given to it in whatever order they were given.

By default a FIFO index just keeps whatever rows come to it. However it may have a few options. Setting the option `limit` limits the number of rows stored in the index (not per the whole table but per one of those “miniature tables”). When you try to insert one row too many, the oldest row gets thrown out, and the limit stays unbroken. That's what creates the window behavior: keep the most recent N rows.

If you look at the sample output, you can see that inserting the rows with ids 1-4 generates only the insert events on the table. But the rows 5 and 6 start overflowing their FIFO indexes, and cause the oldest row to be automatically deleted before completing the insert of the new one.

A FIFO index doesn't have to be nested inside a hash index. If you put a FIFO index at the top level, it will control the whole table. So it would be not two last record per key but two last records inserted in the whole table.

Continuing with the example, the table gets created, and then the index types get extracted back from the table type. Now, why not just write out the table type creation with intermediate variables as shown above and remember the index references? At some point in the past this actually would have worked but not any more. It has to do with the way the table type and its index types are connected. It's occasionally convenient to create one index type and then reuse it in multiple table types. However for the whole thing to work, the index type must be tied to its particular table type. This tying together happens when the table type is initialized. If you put the same index type into two table types, then when the first table type is initialized, the index type will get tied to it. The second table type would then fail to initialize because an index in it is already tied elsewhere. To get around this dilemma, now when you call `addSubIndex()`, it doesn't connect the original index type, instead it makes a copy of it. That copy then gets tied with the table type and later gets returned back with `findSubIndex()`.

The table methods that take an index type argument absolutely require that the index type must be tied to that table's type. If you try to pass a seemingly the same index type that has not been tied, or has been tied to a different table type, that is an error.

One last note on this subject: there is no interdependency between the methods `makeTable()` and `findSubIndex()`, they can be done in either order.

The example output comes from two sources. The running updates on the table's modifications (the lines with `OP_INSERT` and `OP_DELETE`) are printed from the label `$lbWindowPrint`. The new window contents is printed from the main loop.

The main loop reads the trade records in the simple CSV format without the opcode, and for simplicity inserts directly into the table with the procedural API, bypassing the scheduler. After the row is inserted, the contents of its index group (that “miniature table”) gets printed. The insertion could as well have been done with passing directly the row reference, without explicitly creating a handle. But that handle will be used to demonstrate an interesting point.

To print the contents of an index group, we need to find its boundaries. In Triceps these boundaries are expressed as the first row handle of the group, and as the row handle right after the group. There is an internal logic to that, and it will be explained later, but for now just take it on faith.

With the information we have, there are two ways to find the first row of the group:

- With the table's method `findIdx()`. It's very much like `find()`, only it has an extra argument of a specific index type. If the index type given has no further nesting in it, `findIdx()` works exactly like `find()`. In fact, `find()` is exactly such a special case of `findIdx()` with an automatically chosen index type. If you use an index type with further nesting under it, `findIdx()` will return the handle of the first row in the group under it (or the usual NULL row handle if not found).

- If we create the row handle explicitly before inserting it into the table, as was done in the example, that will be the exact row handle inserted into the table. Not a copy or anything but this particular row handle. After a row handle gets inserted into the table, it knows its position in the indexes. It knows, in which group it is. And we still have a reference to it. So then we can use this knowledge to navigate within the group, jump to the first row handle in the group with `firstOfGroupIdx()`. It also takes an index type but in this case it's the type that controls the group, the FIFO index in out case.

The example shows both ways. As a demonstration, it uses the first way if the symbol is “AAA” and the second way for all the other symbols.

The end boundary is found by calling `nextGroupIdx()` on the first row's handle. The handle of the newly inserted row could have also been used for `nextGroupIdx()`, or any other handle in the group. For any handle belonging to the same group, the result is exactly the same.

And finally, after the iteration boundaries have been found, the iteration on the group can run. The end condition comparison is done with `same()`, to compare the row handle references and not just their Perl-level wrappers. The stepping is done with `nextIdx()`, with is exactly like `next()` but according to a particular index, the FIFO one. This has actually been done purely to show off this method. In this particular case the result produced by `next()`, `nextIdx()` on the FIFO index type and `nextIdx()` on the outer hash index type is exactly the same. We'll come to the reasons of that yet.

Checking the results of find and iteration methods by `or confess` would be quite inconvenient, so these methods have been already made to confess on errors.

Looking forward, as you iterate through the group, you could do some manual aggregation along the way. For example, find the average price of the last two trades, and then do something useful with it.

There is also a piece of information that you can find without iteration: the size of the group.

```
$size = $table->groupSizeIdx($idxType, $row_or_rh);
```

This information is important for the joins, and iterating every time through the group is inefficient if all you want to get is the group size. Since when you need this data you usually have the row and not the row handle, this operation accepts either and implicitly performs a `findIdx()` on the row to find the row handle. Moreover, even if it receives the argument of a row handle that is not in the table, it will also automatically perform a `findIdx()` on it (though calling it for a row handle in the table is more efficient because the group would not need to be looked up first).

If there is no such group in the table, the result will be 0.

The `$idxType` argument is the non-leaf parent index of the group. (Using a leaf index type is not an error but it always returns 0, because there are no groups under it). It's basically the same index type as you would use in `findIdx()` to find the first row of the group or in `firstOfGroupIdx()` or `nextGroupIdx()` to find the boundaries of thr group. Remember, a non-leaf index type defines the groups, and the nested index types under it define the order in those groups (and possibly further break them down into sub-groups).

It's a bit confusing, so let's recap with another example. If you have a table type defined as:

```
our $ttPosition = Triceps::TableType->new($rtPosition)
  ->addSubIndex("primary",
    Triceps::IndexType->newHashed(key => [ "date", "customer", "symbol" ])
  )
  ->addSubIndex("currencyLookup", # for joining with currency conversion
    Triceps::IndexType->newHashed(key => [ "date", "currency" ])
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
  ->addSubIndex("byDate", # for cleaning by date
    Triceps::SimpleOrderedIndex->new(date => "ASC")
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
```

then it would make sense to call `groupSizeIdx()`, `firstOfGroupIdx()` and `nextGroupIdx()` with the indexes “currencyLookup” or “byDate” but not with “primary”, “currencyLookup/grouping” nor “byDate/grouping”. You can call `findIdx()` with any index, but for “currencyLookup” or “byDate” it would return the first row of the group while for “primary”, “currencyLookup/grouping” or “byDate/grouping” it would return the only matching row. On the other hand, for iteration in a group, it makes sense to call `nextIdx()` only on “primary”, “currencyLookup/grouping” or “byDate/grouping”. Calling `nextIdx()` on the non-leaf index types is not an error but it would in effect resolve to the same thing as using their first leaf sub-indexes.

9.7. Secondary indexes

The last example dealt only with the row inserts, because it could not handle the deletions that well. What if the trades may get cancelled and have to be removed from the table? There is a solution to this problem: add one more index. Only this time not nested but in parallel. The indexes in the table type become tree-formed:

```
TableType
```

```
+-IndexType Hash "byId" (id)
+-IndexType Hash "bySymbol" (symbol)
  +-IndexType Fifo "last2"
```

It's very much like the common relational databases where you can define multiple indexes on the same table. Both indexes `byId` and `bySymbol` (together with its nested sub-index) refer to the same set of rows stored in the table. Only `byId` allows to easily find the records by the unique id, while `bySymbol` is responsible for keeping then grouped by the symbol, in FIFO order. It could be said that `byId` is the primary index (since it has a unique key) and `bySymbol` is a secondary one (since it does the grouping) but from the Triceps'es standpoint they are pretty much equal and parallel to each other.

To illustrate the point, here is a modified version of the previous example. Not only does it manage the deletes but also computes the average price of the collected transactions as it iterates through the group, thus performing a manual aggregation.

```
our $uTrades = Triceps::Unit->new("uTrades") or confess "$!";
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
) or confess "$!";

our $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
      ->addSubIndex("last2",
        Triceps::IndexType->newFifo(limit => 2)
      )
  )
  or confess "$!";
$ttWindow->initialize() or confess "$!";
our $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

# remember the index type by symbol, for searching on it
our $itSymbol = $ttWindow->findSubIndex("bySymbol") or confess "$!";
# remember the FIFO index, for finding the start of the group
our $itLast2 = $itSymbol->findSubIndex("last2") or confess "$!";

# remember, which was the last row modified
our $rLastMod;
our $lbRememberLastMod = $uTrades->makeLabel($rtTrade, "lbRememberLastMod",
  undef, sub { # (label, rowop)
    $rLastMod = $_[1]->getRow();
  }) or confess "$!";
$tWindow->getOutputLabel()->chain($lbRememberLastMod) or confess "$!";

# Print the average price of the symbol in the last modified row
sub printAverage # (row)
{
  return unless defined $rLastMod;
  my $rhFirst = $tWindow->findIdx($itSymbol, $rLastMod);
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2) or confess "$!";
  print("Contents:\n");
  my $avg;
  my ($sum, $count);
  for (my $rhi = $rhFirst;
      !$rhi->same($rhEnd); $rhi = $rhi->nextIdx($itLast2)) {
    print("  ", $rhi->getRow()->printP(), "\n");
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  if ($count) {
    $avg = $sum/$count;
  }
  print("Average price: ", (defined $avg? $avg: "Undefined"), "\n");
}

while(<STDIN>) {
  chomp;
  my @data = split(/,/);
  my $op = shift @data; # string opcode, if incorrect then will die later
  my $rTrade = $rtTrade->makeRowArray(@data) or confess "$!";
  my $rowop = $tWindow->getInputLabel()->makeRowop($op, $rTrade)
    or confess "$!";
  $uTrades->call($rowop) or confess "$!";
  &printAverage();
  undef $rLastMod; # clear for the next iteration
  $uTrades->drainFrame(); # just in case, for completeness
}
```

And an example of its work, with the input lines shown in bold:

```
OP_INSERT,1,AAA,10,10
Contents:
  id="1" symbol="AAA" price="10" size="10"
Average price: 10
OP_INSERT,2,BBB,100,100
Contents:
  id="2" symbol="BBB" price="100" size="100"
Average price: 100
OP_INSERT,3,AAA,20,20
Contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
Average price: 15
OP_INSERT,4,BBB,200,200
Contents:
  id="2" symbol="BBB" price="100" size="100"
  id="4" symbol="BBB" price="200" size="200"
Average price: 150
OP_INSERT,5,AAA,30,30
Contents:
  id="3" symbol="AAA" price="20" size="20"
  id="5" symbol="AAA" price="30" size="30"
Average price: 25
OP_INSERT,6,BBB,300,300
Contents:
  id="4" symbol="BBB" price="200" size="200"
  id="6" symbol="BBB" price="300" size="300"
Average price: 250
OP_DELETE,3
Contents:
  id="5" symbol="AAA" price="30" size="30"
Average price: 30
OP_DELETE,5
Contents:
Average price: Undefined
```

The input has changed: now an extra column is prepended to it, containing the opcode for the row. The updates to the table are not printed any more, but the calculated average price is printed after the new contents of the group.

In the code, the first obvious addition is the extra index in the table type. The label that used to print the updates is gone, and replaced with another one, that remembers the last modified row in a global variable.

That last modified row is then used in the function `printAverage()` to find the group for iteration. Why? Could not we just remember the symbol from the input data? Not always. As you can see from the last two input rows with `OP_DELETE`, the trade id is the only field required to find and delete a row using the index `byId`. So these trade cancellation rows take a shortcut and only provide the trade id, not the rest of the fields. If we try to remember the symbol fields from them, we'd remember an `undef`. Can we just look up the row by id after the incoming rowop has been processed? Not after the deletion. If we try to find the symbol by looking up the row after the deletion, we will find nothing, because the row will already be deleted. We could look up the row in the table before the deletion, and remember it, and afterwards do the look-up of the group by it. But since on deletion the row with will come to the table's output label anyway, we can just ride the wave and remember it instead of doing the manual look-up. And this also spares the need of creating a row with the last symbol for searching: we get a ready pre-made row with the right symbol in it.

Note that in this example, unlike the previous one, there are no two ways of finding the group any more: after deletion the row handle will not be in the table any more, and could not be used to jump directly to the beginning of its group. `findIdx()` has to be used to find the group.

By the time `printAverage()` executes, it could happen that all the rows with that symbol will be gone, and the group will disappear. This situation is handled nicely in an automatic way: `findIdx()` will return a NULL row handle, for which then `nextGroupIdx()` will also return a NULL row handle. The for-loop will immediately satisfy the condition of `$rhi->same($rhEnd)`, it will make no iterations, the `$count` and `$avg` will be left undefined. In result no rows will be printed and the average value will be printed as “Undefined”, as you can see in the reaction to the last input row in the sample output.

The main loop becomes reduced to reading the input, splitting the line, separating the opcode, calling the table's input label, and printing the average. The auto-conversion from the opcode name is used when constructing the rowop. Normally it's not a good practice, since the program will die if it finds a bad rowop in the input, but good enough for a small example. The direct use of `$uTrades->call()` guarantees that by the time it returns, the last modified row will be remembered in `$rLastMod`, available for `printAverage()` to use.

After the average is calculated, `$rLastMod` is reset to prevent it from accidentally affecting the next row. If the next row is an attempt to delete a trade id that is not in the table any more, the DELTE operation will have no effect on the table, and nothing will be sent from the table's output label. `$rLastMod` will stay undefined, and `printAverage()` will check it and immediately return. An attempt to pass an `undef` argument to `findIdx()` would be an error.

The final `$uTrades->drainFrame()` is there purely for completeness. In this case we know that nothing will be scheduled by the labels downstream from the table, and there will be nothing to drain.

Now, an interesting question is: how does the table know, that to delete a row, it has to find it using the field `id`? Or, since the deletion internally uses `find()`, the more precise question is: how does `find()` know that it has to use the index `byId`? It doesn't use any magic. It simply goes by the first index defined in the table. That's why the index `byId` has been very carefully placed before `bySymbol`. The same principle applies to all the other functions like `next()`, that use an index but don't receive one as an argument: the first index is always the default index. There is a bit more detail to it, but that's the rough principle.

9.8. Sorted index

The hashed index provides a way to store rows indexed by a key. It is fast but it has a price to pay for that speed: when iterating through it, the records come in an unpredictable (though repeatable, within a particular machine architecture) order determined by the hash function. If the order doesn't matter, that's fine. But often the order does matter, and is desirable even at the tradeoff of the reduced performance.

The sorted index provides a solution for this problem. It is created with:

```
$it = Triceps::IndexType->newPerlSorted($sortName,
  \&initFunc, \&compareFunc, @args);
```

The “Perl” in “newPerlSorted” refers to the fact that the sorting order is specified as a Perl comparison function.

`$sortName` is just a symbolic name for printouts. It's used when you call `$it->print()` (directly or as a recursive call from the table type print) to let you know what kind of index type it is, since it can't print the compiled comparison function. It is also used in the error messages if something dies inside the comparison function: the comparison is executed from deep inside the C++ code, and by that time the `$sortName` is the only way to identify the source of the problems. It's not the same name as used to connect the index type into the table type hierarchy with `addSubIndex()`. As usual, an index type may be reused in multiple hierarchies, with different names, but in all cases it will also keep the same `$sortName`. This may be easier to show with an example:

```
$rt1 = Triceps::RowType->new(
  a => "int32",
  b => "string",
) or confess "$!";

$it1 = Triceps::IndexType->newPerlSorted("basic", undef, \&compBasic)
  or confess "$!";

$tt1 = Triceps::TableType->new($rt1)
  ->addSubIndex("primary", $it1)
  or confess "$!";

$tt2 = Triceps::TableType->new($rt1)
  ->addSubIndex("first", $it1)
  or confess "$!";

print $tt1->print(), "\n";
print $tt2->print(), "\n";
```

The print calls in it will produce:

```
table (
  row {
    int32 a,
    string b,
  }
) {
  index PerlSortedIndex(basic) primary,
}
table (
  row {
    int32 a,
    string b,
  }
) {
  index PerlSortedIndex(basic) first,
}
```

Both the name of the index type in the table type and the name of the sorted index type are printed, but in different spots.

The `initFunc` and/or `compareFunc` references specify the sorting order. One of them may be left undefined but not both. `@args` are the optional arguments that will be passed to both functions.

The easiest but least flexible way is to just use the `compareFunc`. It gets two Rows (not RowHandles!) as arguments, plus whatever is specified in `@args`. It returns the usual Perl-style “<=>” result. For example:

```
sub compBasic # ($row1, $row2)
{
  return $_[0]->get("a") <=> $_[1]->get("a");
}
```

Don't forget to use “<=>” for the numbers and “cmp” for the strings. The typical Perl idiom for sorting by more than one field is to connect them by “||”.

Or, if we want to specify the field names as arguments, we could define a sort function that sorts first by a numeric field in ascending order, then by a string field in descending order:

```
sub compAscDesc # ($row1, $row2, $numFldAsc, $strFldDesc)
{
  my ($row1, $row2, $numf, $strf) = @_;
  return $row1->get($numf) <=> $row2->get($numf)
    || $row2->get($strf) cmp $row1->get($strf); # backwards for descending
}

my $sit = Triceps::IndexType->newPerlSorted("by_a_b", undef,
  \&compAscDesc, "a", "b") or confess "$!";
```

This assumes that the row type will have a numeric field “a” and a string field “b”. The problem is that if it doesn't then this will not be discovered until you create a table and try to insert some rows into it, which will finally call the comparison function. Even then it won't be exactly obvious because this comparison function never checks `"$!"` after `get()`, and you'll see no failures but all the rows will be considered equal and will replace each other.

You could check that the arguments match the row type (`$row1->getType()`) in the comparison function but that would add extra overhead, and the Perl comparisons are slow enough as they are.

The `initFunc` provides a way to do that check and more. It is called at the table type initialization time. By this time all this extra information is known, and it gets the references to the table type, index type (itself, but with the class stripped back to Triceps::IndexType), row type, and whatever extra arguments that were passed. It can do all the checks once.

The init function's return value is kind of backwards to everything else: on success it returns `undef`, on error it returns the error message. It could die too, but simply returning an error message is somewhat nicer. The returned error messages may contain multiple lines separated by “\n”, so it should try to collect all the error information it can.

The init function that would check the arguments for the last example can be defined as:

```
sub initNumStr # ($tabt, $idxt, $rowt, @args)
{
  my ($tabt, $idxt, $rowt, @args) = @_;
  my %def = $rowt->getdef(); # the field definition
  my $errors; # collect as many errors as possible
  my $t;

  if ($#args != 1) {
    $errors .= "Received " . ($#args + 1) . " arguments, must be 2.\n"
  } else {
    $t = $def{$args[0]};
    if ($t !~ /int32$|int64$|float64$/) {
      $errors .= "Field '" . $args[0] . "' is not of numeric type.\n"
    }
    $t = $def{$args[1]};
    if ($t !~ /string$|uint8/) {
      $errors .= "Field '" . $args[1] . "' is not of string type.\n"
    }
  }

  if (defined $errors) {
    # help with diagnostics, append the row type to the error listing
    $errors .= "the row type is:\n";
    $errors .= $rowt->print();
  }
  return $errors;
}

my $sit = Triceps::IndexType->newPerlSorted("by_a_b", \&initNumStr,
  \&compAscDesc, "a", "b") or confess "$!";
```

The init function can do even better: it can create and set the comparison function. It's done with:

```
$idxt->setComparator(\&compareFunc)
  or return "Failed to set comparator: $!";
```

Instead of the usual “`or confess`”, this snippet shows “`or return`” because this is the error indication convention of the init function. But “`or confess`” woudl work too.

When the init function sets the comparator, the compare function argument in `newPerlSorted()` can be left undefined, because `setComparator()` would override it anyway. But one way or the other, the compare function must be set, or the index type initialization and with it the table type initialization will fail.

By the way, the sorted index type init function is **not** of the same kind as the aggregator type init function. The aggregator type could use an init function of this kind too, but at the time it looked like too much extra complexity. It probably will be added in the future. But more about aggregators later.

A fancier example of the init function will be shown in the next section.

Internally the implementation of the sorted index shares much with the hashed index. They both are implemented as trees but they compare the rows in different ways. The hashed index is aimed for speed, the sorted index for flexibility. The common implementation means that they share certain traits. Both kinds have the unique keys, there can not be two rows with the same key in an index of either kind. Both kinds allow to nest other indexes in them.

9.9. Ordered index

To specify the sorting order in a more SQL-like fashion, Triceps has the class SimpleOrderedIndex. It's implemented entirely in Perl, on top of the sorted index. Besides being useful by itself, it shows off two concepts: the initialization function of the sorted index, and the template with code generation on the fly.

First, how to create an ordered index:

```
$it = Triceps::SimpleOrderedIndex->new($fieldName => $order, ...)
  or confess "$!";
```

The arguments are the key fields. `$order` is one of `"ASC"` for ascending and `"DESC"` for descending. Here is an example of a table with this index:

```
my $tabType = Triceps::TableType->new($rowType)
  ->addSubIndex("sorted",
    Triceps::SimpleOrderedIndex->new(
      a => "ASC",
      b => "DESC",
    )
  ) or confess "$!";
```

When it gets translated into a sorted index, the comparison function gets generated automatically. It's smart enough to generate the string comparisons for the `string` and `uint8` fields, and the numeric comparisons for the numeric fields. It's not smart enough to do the locale-specific comparisons for the strings and locale-agnostic for the `uint8`, it just uses whatever you have set up in `cmp` for both. It treats the NULL field values as numeric 0 or empty strings. It doesn't handle the array fields at all but can at least detect such attempts and flag them as errors.

A weird artifact of the boundary between C++ and Perl is that when you get the index type back from the table type like

```
$sortIdx = $tabType->findSubIndex("sorted") or confess "$!";
```

the reference stored in `$sortIdx` will be of the base type Triceps::IndexType. That's because the C++ internals of the TableType object know nothing about any derived Perl types. But it's no big deal, since there are no other useful methods for SimpleOrderedIndex anyway. For the future, I have an idea of a workaround, but it has to wait for the future.

If you call `$sortIdx->print()`, it will give you an idea of how it was constructed:

```
PerlSortedIndex(SimpleOrder a ASC, b DESC, )
```

The contents of the parenthesis is a sort name from the sorted index'es standpoint. It's an arbitrary string. But when the ordered index prepares this string to pass to the sorted index, it puts its arguments into it.

Now the interesting part, I want to show the implementation of the ordered index. It's not too big and it shows the flexibility and the extensibility of Triceps:

```
package Triceps::SimpleOrderedIndex;
use Carp;

our @ISA = qw(Triceps::IndexType);

# Create a new ordered index. The order is specified
# as pairs of (fieldName, direction) where direction is a string
# "ASC" or "DESC".
sub new # ($class, $fieldName => $direction...)
{
  my $class = shift;
  my @args = @_; # save a copy

  # build a descriptive sortName
  my $sortName = 'SimpleOrder ';
  while ($#_ >= 0) {
    my $fld = shift;
    my $dir = shift;
    $sortName .= quotemeta($fld) . ' ' . quotemeta($dir) . ', ';
  }

  $self = Triceps::IndexType->newPerlSorted(
    $sortName, \&init, undef, @args
  ) or confess "$!";
  bless $self, $class;
  return $self;
}

# The initialization function that actually parses the args.
sub init # ($tabt, $idxt, $rowt, @args)
{
  my ($tabt, $idxt, $rowt, @args) = @_;
  my %def = $rowt->getdef(); # the field definition
  my $errors; # collect as many errors as possible
  my $compare = "sub {\n"; # the generated comparison function
  my $connector = "return"; # what goes between the comparison operators

  while ($#args >= 0) {
    my $f = shift @args;
    my $dir = uc(shift @args);

    my ($left, $right); # order the operands depending on sorting direction
    if ($dir eq "ASC") {
      $left = 0; $right = 1;
    } elsif ($dir eq "DESC") {
      $left = 1; $right = 0;
    } else {
      $errors .= "unknown direction '$dir' for field '$f', use 'ASC' or 'DESC'\n";
      # keep going, may find more errors
    }

    my $type = $def{$f};
    if (!defined $type) {
      $errors .= "no field '$f' in the row type\n";
      next;
    }

    my $cmp = "<=>"; # the comparison operator
    if ($type eq "string"
    || $type =~ /^uint8.*/) {
      $cmp = "cmp"; # string version
    } elsif($type =~ /\]$/) {
      $errors .= "can not order by the field '$f', it has an array type '$type', not
 supported yet\n";
      next;
    }

    my $getter = "->get(\"" . quotemeta($f) . "\")";

    $compare .= "  $connector \$_[$left]$getter $cmp \$_[$right]$getter\n";

    $connector = "||";
  }

  $compare .= "  ;\n";
  $compare .= "}";

  if (defined $errors) {
    # help with diagnostics, append the row type to the error listing
    $errors .= "the row type is:\n";
    $errors .= $rowt->print();
  } else {
    # compile the comparison
    #print STDERR "DEBUG Triceps::SimpleOrderedIndex::init: comparison function:\n$compare
\n";
    my $cmpfunc = eval $compare
      or return "Triceps::SimpleOrderedIndex::init: internal error when compiling the
 compare function:\n"
        . "$@\n"
        . "The generated comparator was:\n"
        . $compare;
    $idxt->setComparator($cmpfunc)
      or return "Triceps::SimpleOrderedIndex::init: internal error: can not set the
 compare function:\n"
      . "$!\n";
  }
  return $errors;
}
```

The class constructor simply builds the sort name from the arguments and offloads the rest of logic to the init function. It can't really do much more: when the index type object is constructed, it doesn't know yet, where it will be used and what row type it will get. It tries to enquote nicely the weird characters in the arguments when they go into the sort name. Not that much use is coming from it at the moment: the C++ code that prints the table type information doesn't do the same, so there still is a chance of misbalanced quotes in the result. But perhaps the C++ code will be fixed at some point too.

The init function is called at the table type initialization time with all the needed information. It goes through all the arguments, looks up the fields in the row type, and checks them for correctness. It tries to collect as much of the error information as possible. The returned error messages may contain multiple lines separated by “\n”, and the ordered index makes use of it. The error messages get propagated back to the table type level, nicely indented and returned from the table initialization. If the init function finds any errors, it appends the printout of the row type too, to make finding what went wrong easier. A result of a particularly bad call to a table type initialization may look like this:

```
index error:
  nested index 1 'sorted':
    unknown direction 'XASC' for field 'z', use 'ASC' or 'DESC'
    no field 'z' in the row type
    can not order by the field 'd', it has an array type 'float64[]', not supported yet
    the row type is:
    row {
      uint8 a,
      uint8[] b,
      int64 c,
      float64[] d,
      string e,
    }
```

Also as the init goes through the arguments, it constructs the text of the compare function in the variable `$compare`. Here the use of `quotemeta()` for the user-supplied strings is important to avoid the syntax errors in the generated code. If no errors are found in the arguments, the compare function gets compiled with `eval`. There should not be any errors, but it's always better to check. Finally the compiled compare function is set in the sorted index with

```
$idxt->setComparator($cmpfunc)
```

If you uncomment the debugging printout line (and run “`make`”, and maybe “`make install`” afterwards), you can see the auto-generated code printed on stderr when you use the simple ordered index. It will look somewhat like this:

```
sub {
  return $_[0]-&gt;get("a") cmp $_[1]-&gt;get("a")
  || $_[1]-&gt;get("c") &lt;=&gt; $_[0]-&gt;get("c")
  || $_[0]-&gt;get("b") cmp $_[1]-&gt;get("b")
  ;
}
```

That's it! An entirely new piece of functionality added in a smallish Perl snippet. This is your typical Triceps template: collect the arguments, use them to build Perl code, and compile it. Of course, if you don't want to deal with the code generation and compilation, you can just call your class methods and whatnot to interpret the arguments. But if the code will be reused, the compilation is more efficient.

9.10. The index tree

The index types in a table type can form a pretty much arbitrary tree. Following the common tree terminology, the index types that have no other index types nested in them, are called the *leaf* index types. Since there seems to be no good one-word naming for the index types that have more index types nested in them ("inner"? "nested" is too confusing), I simply call them *non-leaf*.

At the moment the Hashed, Sorted and Ordered index types can be used only in both leaf and non-leaf positions. The FIFO index types must always be in the leaf position, they don't allow the further nesting.

Now is the time to look deeper into what is going on inside a table. Note that I've been very carefully talking about “index types” and not “indexes”. In this section the difference matters. The index types are in the table type, the indexes are in the table. One index type may generate multiple indexes.

This will become clearer after you see the illustrations. First, the legend in the Figure 9.1 .

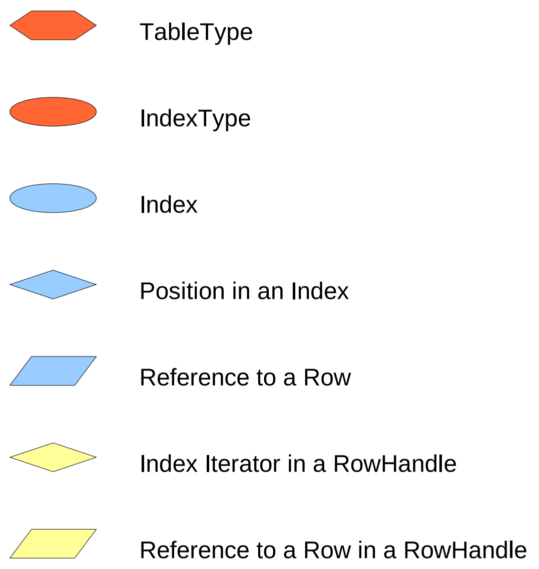
TableType
IndexType
Index
Position in an Index
Reference to a Row
Index Iterator in a RowHandle
Reference to a Row in a RowHandle

Figure 9.1. Drawings legend.

The nodes belonging to the table type are shown in red, the nodes belonging to the table are shown in blue, and the contents of the RowHandle is shown separately in yellow. The lines on the drawings represent not exactly pointers as such but more of the logical connections that may be more complicated than the simple pointers.

The lines in the RowHandle don't mean anything at all, they just show that the parts go together. In reality a RowHandle is a chunk of memory, with various elements placed in that memory. As far as indexes are concerned, the RowHandle contains an iterator for every index where it belongs. This lets it know its position in the table, to iterate along every index, and, most importantly, to be removed quickly from every index. A RowHandle belongs to one index of each index type, and contains the matching number of iterators in it.

The table type is shown as a normal flat tree. But the table itself is more complex and becomes 3-dimensional. Its “view from above” matches the table type's tree but the data grows “up” in the third dimension.

Let's start with the simplest case: a table type with only one index type. Whether the index type is hash or FIFO, doesn't matter here.

```
TableType
+-IndexType "A"
```

Figure 9.2 shows the table structure.

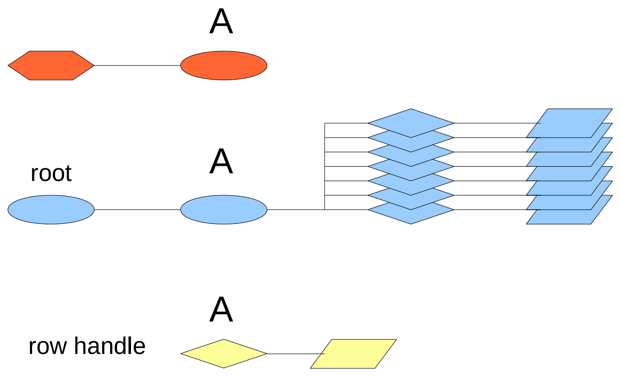

Figure 9.2. One index type.

The table here always contains exactly one index, matching the one defined index type, and the root index. The root index is very dumb, its only purpose is to tie together the multiple top-level indexes into a tree.

The only index of type A provides an ordering of the records, and this ordering is used for the iteration on the table.

For the next example let's look at the straight nesting in Figure 9.3 .

```
TableType
+-IndexType "A"
  +-IndexType "B"
```

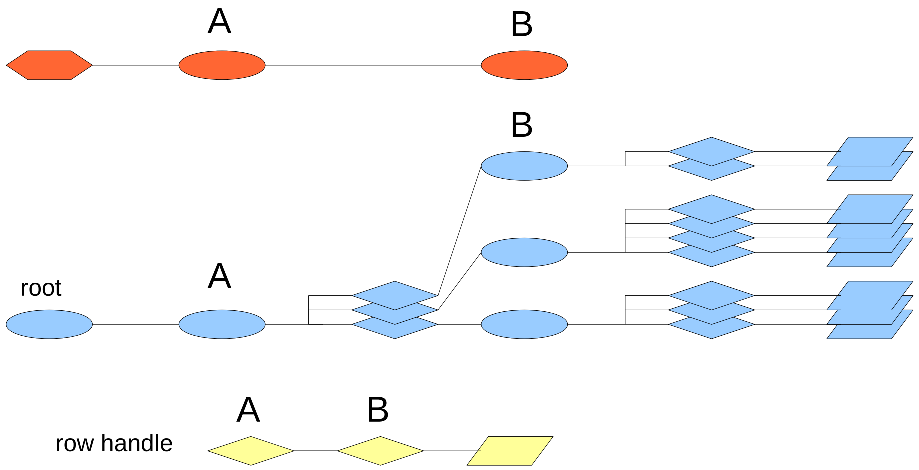

Figure 9.3. Straight nesting.

The stack of row references is shown visually divided to match the indexing, but in reality there is no special division. This was done purely to make the picture easier to read.

There is still only one index of type A. And this is always the case with the top-level indexes, there is only one of them. This index divides the rows into 3 *groups*. Just like the rows in a leaf index, the groups in a non-leaf index are ordered in some index-specific way.

Each group then has its own second-level index of type B. Which then defines an order for the rows in it. To reiterate: the index of type A splits the rows by groups, then the group's index of type B defines the order of the rows in the group.

So what happens when we iterate through the table and ask for the next row handle? The current row handle contains the iterators in the indexes of types A and B. The easy thing is to advance the iterator of type B. Yeah, but in *which* index? The Figure 9.3 shows three indexes of type B, let's call them B1, B2 and B3. The iterator of type B in the row handle tells the relative position in the index, but it doesn't tell, which index it is. We need to step back and look at the index type A. It's the top-level index type, so there is always only one index for it. Then we take the iterator of type A and find this row's group in the index A. The group contains the index of type B, say B1. We can then take this index B1, take the iterator of type B from the row handle, and advance this iterator in this index. If the advance succeeded, then great, we've got the next row handle. But if the current row was the last row in B1, we need to step back to the index A again, advance the current row handle's iterator of type A there, find its index B2, and pick the first row handle of B2.

This process is what happens when we use `$rh->nextIdx($itB)`. The iteration goes by the leaf index type B, however it relies on all the index types in the path from the table type to B. If we do `$rh->next()`, the result is the same because the *first leaf* index type is used as the default index type for the iteration.

If we do `$rh->next($itA)`, the semantics is still the same: return the next row handle (not the next group). There is no way to get to the row handle without going all the way through a leaf index. So when a non-leaf index type is used for the iteration, it gets implicitly extended to its first nested leaf index type.

What would happen if a new row gets inserted, and the index type A determines that it does not belong to any of the existing groups? A new group will be created and inserted in the appropriate position in A's order. This group will have a new index of type B created, and the new row inserted in that index.

What would happen if both rows in B1 are removed? B1 will become empty and will be collapsed. The index A will delete the B1's group and B1 itself, and will remain with only two groups. The effect propagates upwards: if all the rows are removed, the last index of type B will collapse, then the index A will become empty and also collapse and be deleted. The only thing left will be the root index that stays in the existence no matter what.

When a table is first created, it has only the root index. The rest of the indexes pop into the existence as the rows get inserted. If you wonder, yes, this does apply to a table type with only one index type as well. Just this point has not been brought up until now.

Among all this froth of creation and collapse the iterators stay stable. Once a row is inserted, the indexes leading to it are not going anywhere (at least until that row gets removed). But since other rows and groups may be inserted around it, the notion of what row is next, will change over time.

Let's go through how the other index-related operations work.

The iteration through the whole table starts with `begin()` or `beginIdx()`, the first being a form of the second that always uses the first leaf index type. `beginIdx()` is fairly straightforward: it just follows the path from the root to the leaf, picking the first position in each index along the way, until it hits the RowHandle, as is shown in Figure 9.4 . That found RowHandle becomes its result. If the table is empty, it returns the NULL row handle.

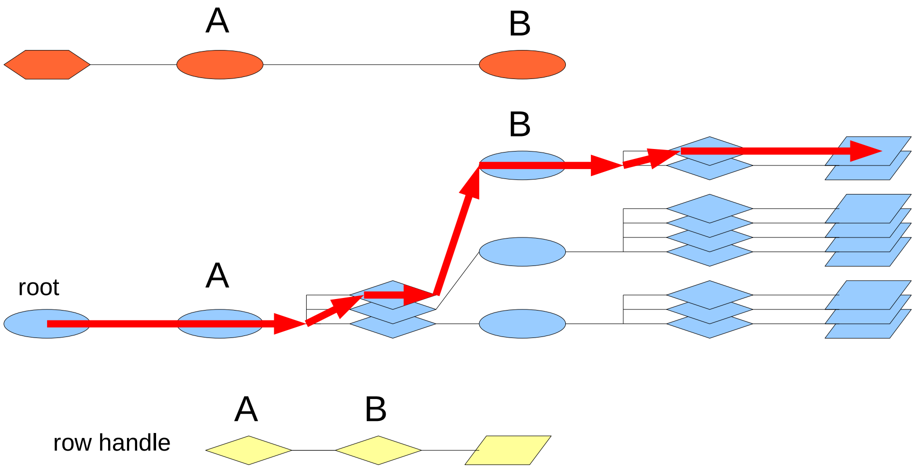

Figure 9.4. `begin()`, `beginIdx($itA)` and `beginIdx($itB)` work the same for this table.

The next pair is `find()` and `findIdx()` (and `findBy()` and `findIdxBy()` are wrappers around those). As usual, `find()` is the same thing as `findIdx()` on the table's first leaf index type. It also follows the path from the root to the target index type. On each step it tries to find a matching position in the current index. If the position could not be found, the search fails and a NULL row handle is returned. If found, it is used to progress to the next index.

As has been mentioned in Section 9.5: “A closer look at the RowHandles” (p. 78), the search always works internally on a RowHandle argument. If a plain Row is used as an argument, a new temporary RowHandle will be created for it, searched, and then freed after the search. This works well for two reasons. First, the indexes already have the functions for comparing two row handles to build their ordering. The same functions are reused for the search. Second, the row handles contain not only the index iterators but also the cached information from the rows, to make the comparisons faster. The exact kind of cached information varies by the index type. The FIFO, Sorted and Ordered indexes use none. The Hashed indexes calculate a hash of the key field values, that will be used as a quick differentiator for the search. This information gets created when the row handle gets created. Whether the row handle is then used to insert into the table or to search in it, the hash is then used in the same way, to speed up the comparisons.

In `findIdx()`, the non-leaf index type arguments behave differently than the leaf ones: up to and including the index of the target type, the search works as usual. But then at the next level the logic switches to the same as in `beginIdx()`, going for the first row handle of the first leaf sub-index. This lets you find the first row handle of the matching group under the target index type.

If you use `$table->findIdx($itA, $rh)`, on Figure 9.5 it will go through the root index to the index A. There it will try to find the matching position. If none is found, the search ends and returns a NULL row handle. If the position is found, the search progresses towards the first leaf sub-index type. Which is the index type B, and which conveniently sits in this case right under A. The position in the index A determines, which index of type B will be used for the next step. Suppose it's the second position, so the second index of type B is used. Since we're now past the target index A, the logic used is the same as for `beginIdx()`, and the first position in B2 is picked. Which then leads to the first row handle of the second sub-stack of handles.

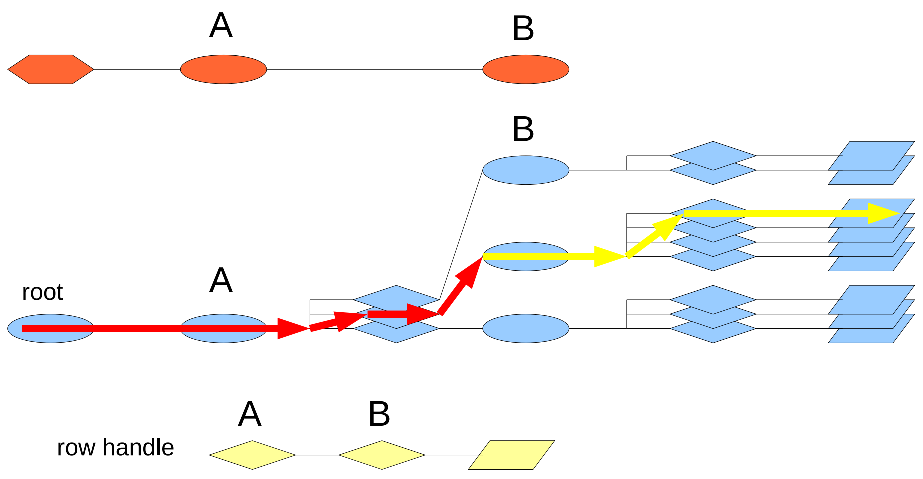

Figure 9.5. `findIdx($itA, $rh)` goes through A and then switches to the `beginIdx()` logic.

The method `firstOfGroupIdx()` allows to navigate within a group, to jump from some row somewhere in the group to the first one, and then from there iterate through the group. The example in Section 9.6: “A window is a FIFO” (p. 80) made use of it.

The Figure 9.6 shows an example of `$table->firstOfGroupIdx($itB, $rh)`, where `$rh` is pointing to the third record in B2. What it needs to do is go back to B2, and then execute the `begin()` logic from there on. However, remember, the row handle does not have a pointer to the indexes in the path, it only has the iterators. So, to find B2, the method does not really back up from the original row. It has to start all the way back from the root and follow the path to B2 using the iterators in `$rh`. Since it uses the ready iterators, this works fast and requires no row comparisons. But logically it's equivalent to backing up by one level, and I'll continue calling it that for simpicity. Once B2 (an index of type B) is reached, the `begin()` logic goes for the first row in there.

`firstOfGroupIdx()` works on both leaf and non-leaf index type arguments in the same way: it backs up from the reference row to the index of that type and executes the `begin()` logic from there. Obviously, if you use it on a non-leaf index type, the `begin()`-like part will follow its first leaf index type.

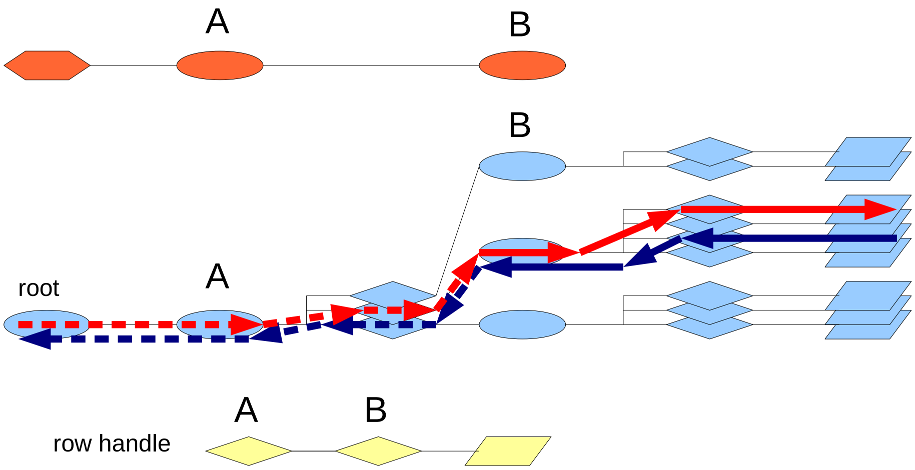

Figure 9.6. **`firstOfGroupIdx($itB, $rh)`**.

The method `nextGroupIdx()` jumps to the first row of the next group, according to the argument index type. To do that, it has to retrace one level higher than `firstOfGroupIdx()`. Figure 9.7 shows that `$table->nextGroupIdx($itB, $rh)` that starts from the same row handle as in Figure 9.6 , has to logically back up to the index A, go to the next iterator there, and then follow to the first row of B3.

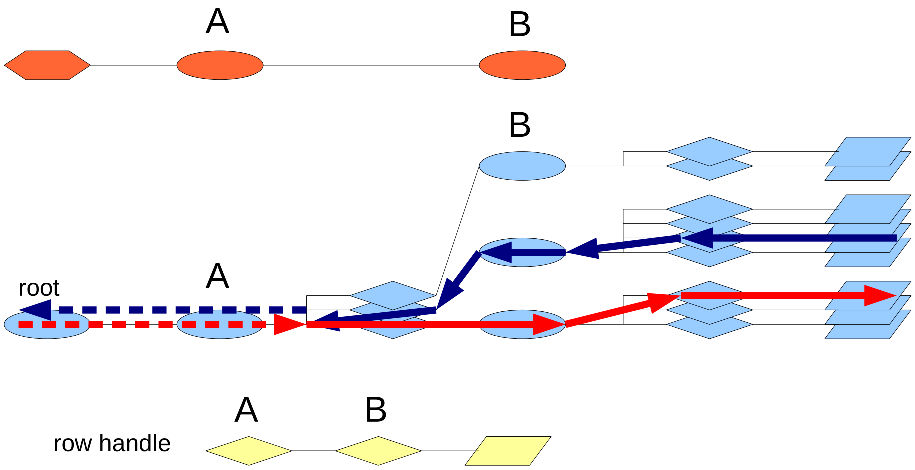

Figure 9.7. **`nextGroupIdx($itB, $rh)`**.

As before, in reality there is no backing up, just the path is retraced from the root using the iterators in the row handle. Once the parent of index type B is reached (which is the index of type A), the path follows not the iterator from the row handle but the next one (yes, copied from the row handle, increased, followed). This gives the index of type B that contains the next group. And from there the same `begin()`-like logic finds its first row.

Same as `firstOfGroupIdx()`, `nextGroupIdx()` may be used on both the leaf and non-leaf indexes, with the same logic.

It's kind of annoying that `firstOfGroupIdx()` and `nextGroupIdx()` take the index type inside the group while `findIdx()` uses takes the parent index type to act on the same group. But as you can see, each of them follows its own internal logic, and I'm not sure if they can be reconciled to be more consistent.

At the moment the only navigation is forward. There is no matching `last()`, `prev()` or `lastGroupIdx()` or `prevGroupIdx()`. They are in the plan, but so far they are the victims of corner-cutting. Though there is a version of `last()` in the AggregatorContext, since it happens to be particularly important for the aggregation.

Continuing our excursion into the index nesting topologies, the next example is of two parallel leaf index types:

```
TableType
+-IndexType A
+-IndexType B
```

The resulting internal arrangement is shown in Figure 9.8 .

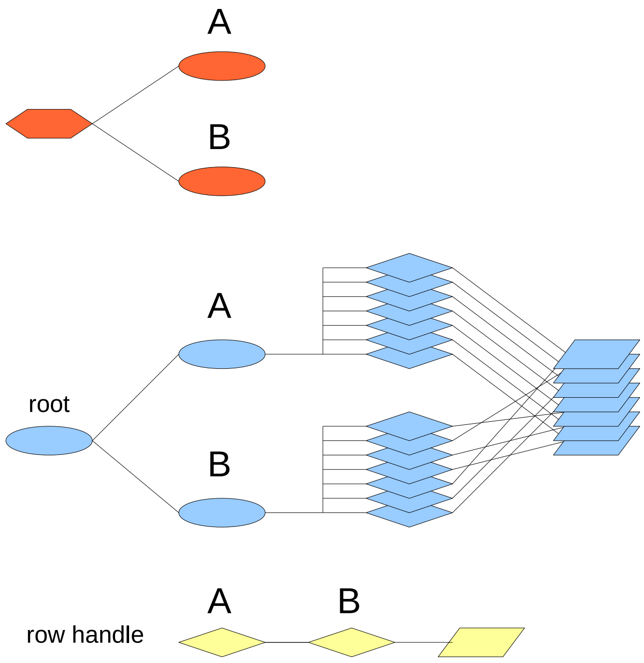

Figure 9.8. Two top-level index types.

Each index type produces exactly one index under the root (since the top-level index types always produce one index). Both indexes contain the same number of rows, and exactly the same rows. When a row is added to the table, it's added to all the leaf index types (one actual index of each type). When a row is deleted from the table, it's deleted from all the leaf index types. So the total is always the same. However the order of rows in the indexes may differ. The drawing shows the row references stacked in the same order as the index A because the index A is of the first leaf index type, and as such is the default one for the iteration.

The row handle contains the iterators for both paths, A and B. It's pretty normal to find a row through one index type and then iterate from there using the other index type.

The next example in Figure 9.9 has a “primary” index with a unique key and a “secondary” index that groups the records:

```
TableType
+-IndexType A
+-IndexType B
  +-IndexType C
```

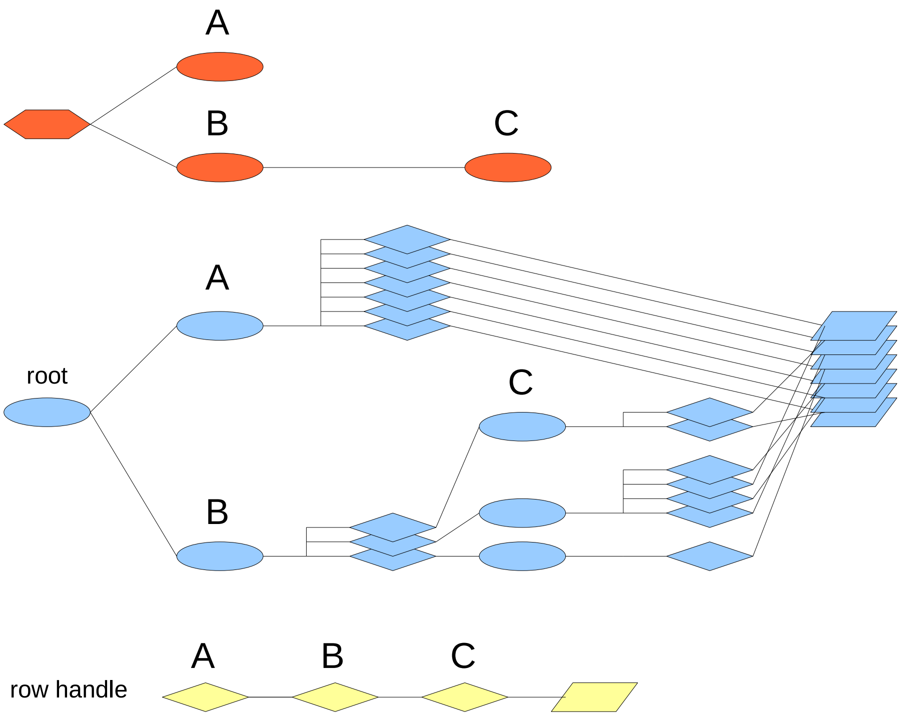

Figure 9.9. A “primary” and “secondary” index type.

The index type A still produces one index and references all the rows directly. The index of type B produces the groups, with each group getting an index of type C. The total set of rows referrable through A and through B is still the same but through B they are split into multiple groups.

And Figure 9.10 shows two leaf index types nested under one non-leaf.

```
TableType
+-IndexType A
  +-IndexType B
  +-IndexType C
```

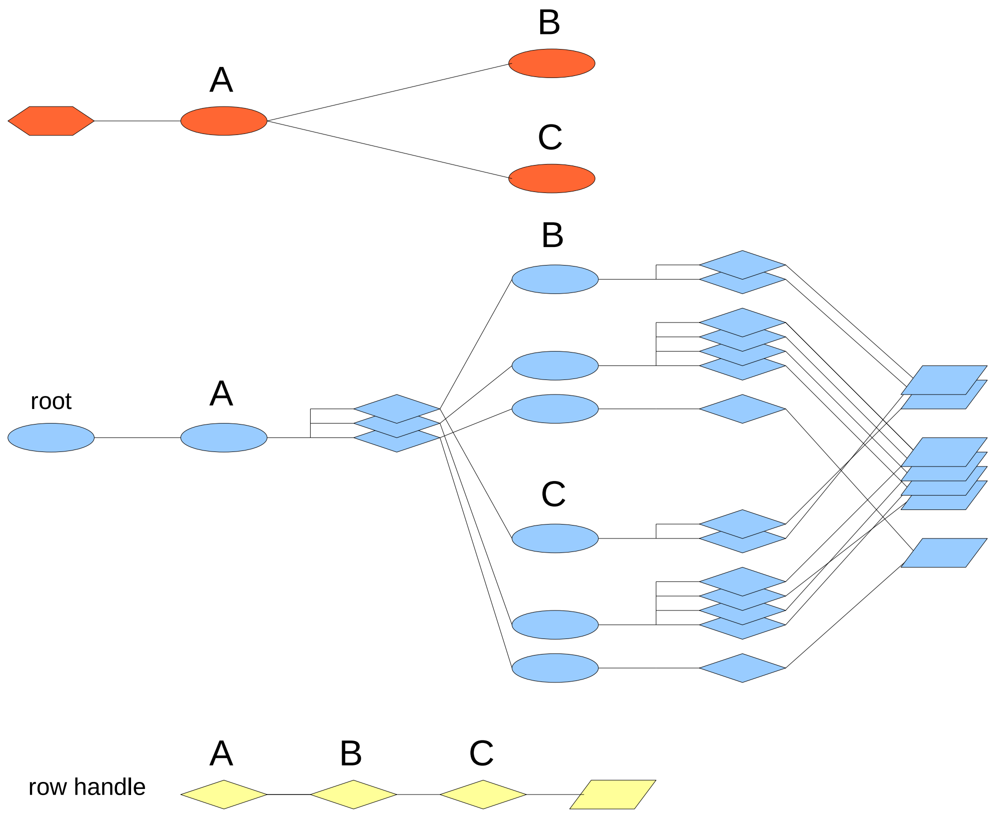

Figure 9.10. Two index types nested under one.

As usual, there is only one index of type A, and it splits the rows into groups. The new item in this picture is that each group has two indexes in it: one of type B and one of type C. Both indexes in the group contain the same rows. They don't decide, which rows they get. The index A decides, which rows go into which group. Then if the group 1 contains two rows, indexes B1 and C1, would both contain two rows each, the exact same set. The stack of row references has been visually split by groups to make this point more clear.

This happens to be a pretty useful arrangement: for example, B might be a hash index type, or a sorted index type, allowing to find the records by the key (and for the sorted index, to iterate in the order of keys), while C might be a FIFO index, keeping the insertion order, and maybe keeping the window size limited.

That's pretty much it for the basic index topologies. Some much more complex index trees can be created, but they would be the combinations of the examples shown. Also, don't forget that every extra index type adds overhead in both memory and CPU time, so avoid adding indexes that are not needed.

One more fine point has to do with the replacement policies. Consider that we have a table that contains the rows with a single field:

```
id int32
```

And the table type has two indexes:

```
TableType
+-IndexType "A" HashIndex key=(id)
+-IndexType "B" FifoIndex limit=3
```

And we send there the rowops:

```
INSERT id=1
INSERT id=2
INSERT id=3
INSERT id=2
```

The last rowop that inserts the row with id=2 for the second time triggers the replacement policy in both index types. In the index A it is a duplicate key and will cause the removal of the previous row with id=2. In the index B it overflows the limit and pushes out the oldest row, the one with id=1. If both records get deleted, the resulting table contents will be 2 rows (shown in FIFO order):

```
id=3
id=2
```

Which is probably not the best outcome. It might be tolerable with a FIFO index and a hashed index but gets even more annoying if there are two FIFO index types in the table: one top-level limiting the total number of rows, another one nested under a hashed index, limiting the number of rows per group, and they start conflicting this way with each other.

The Triceps FIFO index is actually smart enough to avoid such problems: it looks at what the preceding indexes have decided to remove, checks if any of these rows belong to its group, and adjusts its calculation accordingly. In this example the index B will find out that the row with id=2 is already displaced by the index A. That leaves only 2 rows in the index B, so adding a new one will need no displacement. The resulting table contents will be

```
id=1
id=3
id=2
```

However here the order of index types is important. If the table were to be defined as

```
TableType
+-IndexType "B" FifoIndex limit=3
+-IndexType "A" HashIndex key=(id)
```

then the replacement policy of the index type B would run first, find that nothing has been displaced yet, and displace the row id=1. After that the replacement policy of the index type A will run, and being a hashed index, it doesn't have a choice, it has to replace the row id=2. And both rows end up displaced.

If the situations with automatic replacement of rows by the keyed indexes may arise, always make sure to put the keyed leaf index types before the FIFO leaf index types. However if you always diligently send a DELETE before the INSERT of the new version of the recond, then this problem won't occur and the order of index types will not matter.

9.11. Table and index type introspection

A lot of information about a table type and the index types in it can be read back from them.

```
$result = $tabType->isInitialized();
$result = $idxType->isInitialized();
```

return whether a table or index type has been initialized. The index type gets initialized when the table type where it belongs gets initialized. After a table or index type has been initialized, it can not be changed any more, and any methods that change it will return an error. When an index type becomes initialized, it becomes tied to a particular table type. This table type can be read with

```
$tabType = $idxType->getTabtype() or confess "$!";
```

Even though an initialized index type can't be tied to another table, when you add it to another table or index type, a deep copy with all its sub-indexes will be made automatically, and that copy will be uninitialized. So it will be able to get initialized and tied to the new table. However if you want to add more sub-indexes to it, do a manual copy first:

```
$idxTypeCopy = $idxType->copy();
```

The information about the nested indexes can be found with:

```
$itSub = $tabType->findSubIndex("indexName") or confess "$!";
@itSubs = $tabType->getSubIndexes();

$itSub = $idxType->findSubIndex("indexName") or confess "$!";
@itSubs = $idxType->getSubIndexes();
```

The `findSubIndex()` has been already shown in Section 9.7: “Secondary indexes” (p. 84). It allows to find the index types on the next level of nesting, starting down from the table, and going recursively into the sub-indexes. `getSubIndexes()` returns the information about the index types of the next level at once, as the name => value pairs. The result array can be placed into a hash but that would lose the order of the sub-indexes, and the order is important for the logic.

This finds the index types step by step. An easier way to find an index type in a table type by the “path of the index” is with

```
$idxType = $tabType->findIndexPath(\@idxNames);
```

The arguments in the array form a path of names in the index type tree. If the path is not found, the function would confess. An empty path is also illegal and would cause the same result. Yes, the argument is not an array but a reference to array. This array is used essentially as a path object. For example the index from the Section 9.7: “Secondary indexes” (p. 84) could be found as:

```
$itLast2 = $ttWindow->findIndexPath([ "bySymbol", "last2" ]);
```

The key (the set of fields that uniquely identify the rows) of the index type can be found with

```
@keys = $it->getKey();
```

It can be used on any kind of index types but actually returns the data only for the Hashed index types. On the other index types it returns an empty array, though a better support will be available for the Sorted and Ordered indexes in the future.

A fairly common need is to find an index by its name path, and also all the key fields that are used by all the indexes in this path. It's used for such purposes as joins, and it allows to treat a nested index pretty much as a composition of all the indexes in its path. The method

```
($idxType, @keys) = $tableType->findIndexKeyPath(\@path);
```

solves this problem and finds by path an index type that allows the direct look-up by key fields. It requires that every index type in the path returns a non-empty array of fields in `getKey()`. In practice it means that every index in the path must be a Hashed index. Otherwise the method confesses. When the Sorted and maybe other index types will support `getKey()`, they will be usable with this method too.

Besides checking that each index type in the path works by keys, this method builds and returns the list of all the key fields required for a look-up in this index. Note that `@keys` is an actual array and not a reference to array. The return protocol of this method is a little weird: it returns an array of values, with the first value being the reference to the index type, and the rest of them the names of the key fields. If the table type were defined as

```
$tt = Triceps::TableType->new($rt)
  ->addSubIndex("byCcy1",
    Triceps::IndexType->newHashed(key => [ "ccy1" ])
    ->addSubIndex("byCcy12",
      Triceps::IndexType->newHashed(key => [ "ccy2" ])
    )
  )
  ->addSubIndex("byCcy2",
    Triceps::IndexType->newHashed(key => [ "ccy2" ])
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
```

then `$tt->findIndexKeyPath([ "byCcy1", "byCcy12" ])` would return `($ixtref, "ccy1", "ccy2")`, where `$ixtref` is the reference to the index type. When assigned to `($ixt, @keys)`, `$ixtref` would go into `$ixt`, and `("ccy1", "ccy2")` would go into `@keys`.

The key field names in the result go in the order they occurred in the definition, from the outermost to the innermost index. The key fields must not duplicate. It's possible to define the index types where the key fields duplicate in the path, say:

```
$tt = Triceps::TableType->new($rt)
  ->addSubIndex("byCcy1",
    Triceps::IndexType->newHashed(key => [ "ccy1" ])
    ->addSubIndex("byCcy12",
      Triceps::IndexType->newHashed(key => [ "ccy2", "ccy1" ])
    )
  )
or confess "$!";
```

And they would even work fine, with just a little extra overhead from duplication. But `findIndexKeyPath()` will refuse such indexes and confess.

The kind of the index type is also known as the type id. It can be found for an index type with

```
$id = $idxType->getIndexId();
```

It's an integer constant, matching one of the values:

- `&Triceps::IT_HASHED`
- `&Triceps::IT_FIFO`
- `&Triceps::IT_SORTED`

There is no different id for the ordered index, because it's built on top of the sorted index, and would return `&Triceps::IT_SORTED`.

The conversion between the strings and constants for index type ids is done with

```
$intId = &Triceps::stringIndexId($stringId);
$stringId = &Triceps::indexIdString($intId);
```

If an invalid value is supplied, the conversion functions will return `undef`.

There is also a way to find the first index type of a particular kind. It's called somewhat confusingly

```
$itSub = $idxType->findSubIndexById($indexTypeId) or confess "$!";
```

where `$indexTypeId` is one of either of Triceps constants or the matching strings `"IT_HASHED"`, `"IT_FIFO"`, `"IT_SORTED"`.

Technically, there is also IT_ROOT but it's of little use for this situation since it's the root of the index type tree hidden inside the table type, and would never be a sub-index type. It's possible to iterate through all the possible index type ids as

```
for ($i = 0; $i < &Triceps::IT_LAST; $i++) { ... }
```

The first leaf sub-index type, that is the default for iteration, can be found explicitly as

```
$itSub = $tabType->getFirstLeaf();
$itSub = $idxType->getFirstLeaf();
```

If an index is already a leaf, `getFirstLeaf()` on it will return itself. The “leaf-ness” of an index type can be found with:

```
$result = $idxType->isLeaf();
```

The usual reference comparison methods are:

```
$result = $tabType1->same($tabType2);
$result = $tabType1->equals($tabType2);
$result = $tabType1->match($tabType2);

$result = $idxType1->same($idxType2);
$result = $idxType1->equals($idxType2);
$result = $idxType1->match($idxType2);
```

Two table types are considered equal when they have the equal row types, and exactly the same set of index types, with the same names.

Two table types are considered matching when they have the matching row types, and matching set of index types, although the names of the index types may be different.

Two index types are considered equal when they are of the same kind (type id), their type-specific parameters are equal, they have the same number of sub-indexes, with the same names, and equal pair-wise. They must also have the equal aggregators, which will be described in detail in the Chapter 11: “*Aggregation*” (p. 133) .

Two index types are considered matching when they are of the same kind, have matching type-specific parameters, they have the same number of sub-indexes, which are matching pair-wise, and the matching aggregators. The names of the sub-indexes may differ. As far as the type-specific parameters are concerned, it depends on the kind of the index type. The FIFO type considers any parameters matching. For a Hashed index the key fields must be the same. For a Sorted index the sorted condition must also be the same, and by extension this means the same condition for the Ordered index.

9.12. The copy tray

The table methods `insert()`, `remove()` and `deleteRow()` have an extra optional argument: the copy tray.

If used, it will put a copy of all the rowops produced during the operation (including the output of the aggregators, which will be described in Chapter 11: “*Aggregation*” (p. 133)) into that tray. The idea here is to use it in cases if you don't want to connect the output labels of the table directly, but instead collect and process the rows from the tray manually afterwards. Like this:

```
$ctr = $unit->makeTray();
$table->insert($row, $ctr);
foreach my $rop ($ctr->toArray()) {
  ...
}
```

However in reality it didn't work out so well. The processing loop would have to have all the lengthy if-else sequences to branch first by the label (if there are any aggregators) and then by opcode. It looks too difficult. Well, it could work in the simple situations but not more than that.

In the future this feature will likely be deprecated unless it proves itself useful, and I already have a better idea. Because of this, I see no point in going into the more extended examples.

9.13. Table wrap-up

Not all of the table's features have been shown yet. The table class is the cornerstone of Triceps, and everything is connected to it. The aggregators work with the tables and are a whole separate big subject with their own chapter. There also are many more options and small methods that haven't been touched upon yet. They are enumerated in the reference chapter, please refer there.

Chapter 10. Templates

10.1. Comparative modularity

The templates are the Triceps term for the reusable program modules. I've adopted the term from C++ because that was my inspiration for flexibility. But the Triceps templates are much more flexible yet. The problem with the C++ templates is that you have to write in them like in a functional language, substituting loops with recursion, with perverse nested calls for branching, and the result is quite hard to diagnose. Triceps uses the Perl's compilation on the fly to make things easier and more powerful.

Triceps is not unique in the desire for modularity. The other CEP systems have it too, but they tend to have it even more rigid than the C++ templates. Let me show on a simple example.

Coral8 doesn't provide a way to query the windows directly, especially when the CCL is compiled without debugging. So you're expected to make your own. People at a company where I've worked have developed a nice pattern that goes approximately like this:

```
// some window that we want to make queryable
create window w_my schema s_my
keep last per key_a per key_b
keep 1 week;

// the stream to send the query requests
// (the schema can be shared by all simple queries)
create schema s_query (
  qqq_id string // unique id of the query
);
create input stream query_my schema s_query;

// the stream to return the results
// (all result streams will inherit a partial schema)
create schema s_result (
  qqq_id string, // returns back the id received in the query
  qqq_end boolean, // will be TRUE in the special end indicator record
);
create output stream result_my schema inherits from s_result, s_my;

// now process the query
insert into result_my
select q.qqq_id, NULL, w.*
from s_query as q, w_my as w;

// the end marker
insert into result_my (qqq_id, qqq_end)
select qqq_id, TRUE
from s_query;
```

To query the window, a program would select a unique query id, subscribe to result_my with a filter `(qqq_id = unique_id)` and send a record of `(unique_id)` into `query_my`. Then it would sit and collect the result rows. Finally it would get a row with `qqq_end = TRUE` and disconnect.

This is a fairly large amount of code to be repeated for every window. What I would like to to instead is to just write:

```
create window w_my schema s_my
keep last per key_a per key_b
keep 1 week;

make_queryable(w_my);
```

and have the template `make_queryable` expand into the rest of the code (obviously, the schema definitions would not need to be expanded repeatedly, they would go into an include file).

To make things more interesting, it would be nice to have the query filter the results by some field values. Nothing as fancy as SQL, just by equality to some fields. Suppose, s_my includes the fields field_c and field_d, and we want to be able to filter by them. Then the query can be done as:

```
create input stream query_my schema inherits from s_query (
  field_c integer,
  field_d string
);

// result_my is the same as before...

// query with filtering (in a rather inefficient way)
insert into result_my
select q.qqq_id, NULL, w.*
from s_query as q, w_my as w
where
  (q.field_c is null or q.field_c = w.field_c)
  and (q.field_d is null or q.field_d = w.field_d);

// the end marker is as before
insert into result_my (qqq_id, qqq_end)
select qqq_id, TRUE
from s_query;
```

It would be nice then to create this kind of query as a template instantiation

```
make_query(w_my, (field_c, field_d));
```

Or even better, have the template determine the non-NULL fields in the query record and compile the right query on the fly.

But the Coral8 modules (nor the later Sybase CEP R5) aren't flexible enough to do any of it. A CCL module requires a fixed schema for all its interfaces. The StreamBase language is more flexible and allows to achieve some of the flexibility through the capture fields, where the “logically unimportant” fields are carried through the module as one combined payload field. But they don't allow the variable lists of fields as parameters either, nor generation of different model topologies depending on the parameters.

10.2. Template variety

A template in Triceps is generally a function or class that creates a fragment of the model based on its arguments. It provides the access points used to connect this fragment to the rest of the model.

There are different ways do do this. They can be broadly classified in the order of increasing complexity as:

- A function that creates a single Triceps object and returns it. The benefit is that the function would automatically choose some complex object parameters based on the function parameters, thus turning a complex creation into a simple one.
- A class that similarly creates multiple fixed objects and interconnects them properly. It would also provide the accessor methods to export the access points of this sub-model. Since the Perl functions may return multiple values, this functionality sometimes can be conveniently done with a function as well, returning the access points in the return array.
- A class or function that creates multiple objects, with their number and connections dependent on the parameters. For a simple example, a template might receive multiple functions/closures as arguments and then create a pipeline of computational labels, each of them computing one function (of course, this really makes sense only when each label runs in a separate thread).
- A class or function that automatically generates the Perl code that will be used in the created objects. For a simple example, given the pairs of field names and values, a template can generate the code for a filter label that would pass

only the rows where these fields have these values. The same effect can often be achieved by the interpretation as well: keep the arguments until the evaluation needs to be done, and then interpret them. But the early code generation with compilation improves the efficiency of the computation. It's the same idea as in the C++ templates: do more of the hard work at the compile time and then run faster.

The more complex and flexible is the template, the more difficult it's generally to write and debug, but then it just works, encapsulating a complex problem with a simpler interface. There is also the problem of user errors: when the user gives an incorrect argument to a complex template, understanding what exactly went wrong when the error manifests itself, may be quite difficult. The C++ templates are a good example of this. However the use of Perl, a general programming language, as a template language in Triceps provides a good solution for this problem: just check the arguments early in the template and produce the meaningful error messages. It may be a bit cumbersome to write but then easy to use. I also have plans for improving the automatic error reports, to make tracking through the layers of templates easier with minimal code additions in the templates.

I will show the examples of all the template types by implementing the table querying, the same I have shown in CCL in Section 10.1: “Comparative modularity” (p. 109) , only now in Triceps.

10.3. Simple wrapper templates

The query examples will be using the main loop with sockets from the Section 7.8: “Main loop with a socket” (p. 51) . It has two repeating tasks: requesting the socket server to exit, and sending the rows from some label back into the socket. These tasks can be nicely handled with the simple templates:

```
package ServerHelpers;
use Carp;

# Exiting the server.
sub makeExitLabel # ($unit, $name)
{
  my $unit = shift;
  my $name = shift;
  return $unit->makeLabel($unit->getEmptyRowType(), $name, undef, sub {
    $srv_exit = 1;
  });
}

# Sending of rows to the server output.
sub makeServerOutLabel # ($fromLabel)
{
  my $fromLabel = shift;
  my $unit = $fromLabel->getUnit();
  my $fromName = $fromLabel->getName();
  my $lbOut = $unit->makeLabel($fromLabel->getType(),
    $fromName . ".serverOut", undef, sub {
      &main::outCurBuf(join(",", $fromName,
        &Triceps::opcodeString($_[1]->getOpcode()),
        $_[1]->getRow()->toArray()) . "\n");
    });
  $fromLabel->chain($lbOut) or confess "$!";
  return $lbOut;
}
```

Each function is a separate template, they're wrapped into a common package only for the packaging reasons.

`makeExitLabel()` is quite simple, it creates a label with hardcoded function of setting the variable `$srv_exit`. Even its row type is hardcoded to the empty rows.

`makeServerOutLabel()` is more interesting. It prints the rows received from another label into the socket in the simple CSV (as usual, no commas in the values) format, the same as is expected by the socket server. It finds the unit and row type from that parent label, creates the printing label and chains it off the parent label. The newly created label is returned. The return value can be kept in a variable or immediately discarded; since the created label is already chained, it won't disappear. Tha name of the new label is produced from the name of the parent label by appending “.serverOut” to it.

Another similar template that is used throughout the following chapters creates a label that prints the rowop contents:

```
# a template to make a label that prints the data passing through another label
sub makePrintLabel($$) # ($print_label_name, $parent_label)
{
  my $name = shift;
  my $lbParent = shift;
  my $lb = $lbParent->getUnit()->makeLabel($lbParent->getType(), $name,
    undef, sub { # (label, rowop)
      print($_[1]->printP(), "\n");
    }) or die "$!";
  $lbParent->chain($lb) or die "$!";
  return $lb;
}
```

It works very much the same as `makeServerOutLabel()`, only prints to a different destination.

10.4. Templates of interconnected components

Let's move on to the query template. It will work a little differently than the CCL version. First, the socket main loop allows to send the response directly to the same client who issued the request. So there is no need for adding the request id field in the response and for the client filtering by it. Second, Triceps rows have the opcode field, which can be used to signal the end of the response. For example, the data rows can be sent with the opcode INSERT and the indication of the end of response can be sent with the opcode NOP and all fields NULL. The query template can then be made as follows:

```
package Query1;

sub new # ($class, $table, $name)
{
  my $class = shift;
  my $table = shift;
  my $name = shift;

  my $unit = $table->getUnit();
  my $rt = $table->getRowType();

  my $self = {};
  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{table} = $table;
  $self->{inLabel} = $unit->makeLabel($rt, $name . ".in", undef, sub {
    # This version ignores the row contents, just dumps the table.
    my ($label, $rop, $self) = @_;
    my $rh = $self->{table}->begin();
    for (; !$rh->isNull(); $rh = $rh->next()) {
      $self->{unit}->call(
        $self->{outLabel}->makeRowop("OP_INSERT", $rh->getRow()));
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{outLabel}, "OP_NOP");
  }, $self);
  $self->{outLabel} = $unit->makeDummyLabel($rt, $name . ".out");

  bless $self, $class;
  return $self;
}
```

```
sub getInputLabel # ($self)
{
  my $self = shift;
  return $self->{inLabel};
}

sub getOutputLabel # ($self)
{
  my $self = shift;
  return $self->{outLabel};
}

sub getName # ($self)
{
  my $self = shift;
  return $self->{name};
}
```

It creates the input label that does the work and the dummy output label that is used to send the result. The logic is easy: whenever a rowop is received on the input label, iterate through the table and send the contents to the output label. The contents of that received rowop doesn't even matter. The getter methods allow to get the endpoints.

Now this example can be used in a program. Most of it is the example infrastructure: the function to start the server in background and connect a client to it, the creation of the row type and table type to query, and then finally near the end the interesting part: the usage of the query template.

```
# The common client that connects to the port, sends and receives data,
# and waits for the server to exit.
sub run # ($labels)
{
  my $labels = shift;

  my ($port, $pid) = startServer($labels);
  my $sock = IO::Socket::INET->new(
    Proto => "tcp",
    PeerAddr => "localhost",
    PeerPort => $port,
  ) or confess "socket failed: $!";
  while(<STDIN>) {
    $sock->print($_);
    $sock->flush();
  }
  $sock->print("exit,OP_INSERT\n");
  $sock->flush();
  $sock->shutdown(1); # SHUT_WR
  while(<$sock>) {
    print($_);
  }
  waitpid($pid, 0);
}

# The basic table type to be used as template argument.
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
) or confess "$!";
```

```
our $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("bySymbol",
    Triceps::SimpleOrderedIndex->new(symbol => "ASC")
      ->addSubIndex("last2",
        Triceps::IndexType->newFifo(limit => 2)
      )
  )
  or confess "$!";
$ttWindow->initialize() or confess "$!";

my $uTrades = Triceps::Unit->new("uTrades");
my $tWindow = $uTrades->makeTable($ttWindow, "EM_CALL", "tWindow")
  or confess "$!";
my $query = Query1->new($tWindow, "qWindow");
my $srvout = &ServerHelpers::makeServerOutLabel($query->getOutputLabel());

my %dispatch;
$dispatch{$tWindow->getName()} = $tWindow->getInputLabel();
$dispatch{$query->getName()} = $query->getInputLabel();
$dispatch{"exit"} = &ServerHelpers::makeExitLabel($uTrades, "exit");

run(\%dispatch);
```

The function `run()` takes care of making the example easier to run: it starts the server in the background, reads the input data and sends it to the server, then reads the responses and prints them back, and finally waits for the server process to exit. It also takes care of sending the exit request to the server when the input reaches EOF. The approach with first sending all the data there and then reading all the responses back is not very good. It works only if either the data gets sent without any responses, or a small amount of data (not to overflow the TCP buffers along the way) gets sent and then it's all the responses coming back. But it's simple, and it works good enough for the small examples. And actually many of the commercial CEP interfaces work exacly like this: they either publish the data to the model or send a small subscription request and print the data received from the subscription.

The row type and table type have been just copied from some other example. There is no particular meaning to why such fields were selected or why the table has such indexes. They have been selected semi-randomly. The only triucky thing that affects the result is that this table implements a window with a limit of 2 rows per symbol.

After the table is created, the template instantiation is a single call, `Query1->new()`. Then the output label of the query template gets connected to a label that sends the output back to the client, and that's it.

Here is an example of a run, with the input rows printed as always in bold.

```
tWindow,OP_INSERT,1,AAA,10,10
tWindow,OP_INSERT,3,AAA,20,20
qWindow,OP_INSERT
tWindow,OP_INSERT,5,AAA,30,30
qWindow,OP_INSERT
qWindow.out,OP_INSERT,1,AAA,10,10
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_NOP,,,,
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,5,AAA,30,30
qWindow.out,OP_NOP,,,,
```

Because of the way `run()` works, all the input rows are printed before the output ones. If it were smarter and knew, when to expect the responses before sending more inputs, the output would have been:

```
tWindow,OP_INSERT,1,AAA,10,10
tWindow,OP_INSERT,3,AAA,20,20
qWindow,OP_INSERT
qWindow.out,OP_INSERT,1,AAA,10,10
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_NOP,,,,
tWindow,OP_INSERT,5,AAA,30,30
qWindow,OP_INSERT
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,5,AAA,30,30
qWindow.out,OP_NOP,,,,
```

Two rows get inserted into the table, then a query is done, then one more row is inserted, then another query sent. When the third row is inserted, the first row gets thrown away by the window limit, so the second query also returns two rows albeit different than the first query does.

It is possible to fold the table and the client send label creation into the template as well. It will then be used as follows:

```
my $window = $uTrades->makeTableQuery2($ttWindow, "window");

my %dispatch;
$dispatch{$window->getName()} = $window->getInputLabel();
$dispatch{$window->getQueryLabel()->getName()} = $window->getQueryLabel();
$dispatch{"exit"} = &ServerHelpers::makeExitLabel($uTrades, "exit");
```

The rest of the infrastructure would stay unchanged. Just to show how it can be done, I've even added a factory method `Unit::makeTableQuery2()`. The implementation of this template is:

```
package TableQuery2;
use Carp;

sub new # ($class, $unit, $tabType, $name)
{
  my $class = shift;
  my $unit = shift;
  my $tabType = shift;
  my $name = shift;

  my $table = $unit->makeTable($tabType, "EM_CALL", $name)
    or confess "Query2 table creation failed: $!";
  my $rt = $table->getRowType();

  my $self = {};
  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{table} = $table;
  $self->{qLabel} = $unit->makeLabel($rt, $name . ".query", undef, sub {
    # This version ignores the row contents, just dumps the table.
    my ($label, $rop, $self) = @_;
    my $rh = $self->{table}->begin();
    for (; !$rh->isNull(); $rh = $rh->next()) {
      $self->{unit}->call(
        $self->{resLabel}->makeRowop("OP_INSERT", $rh->getRow()));
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{resLabel}, "OP_NOP");
  }, $self);
  $self->{resLabel} = $unit->makeDummyLabel($rt, $name . ".response");

  $self->{sendLabel} = &ServerHelpers::makeServerOutLabel($self->{resLabel});

  bless $self, $class;
  return $self;
}

sub getName # ($self)
```

```
{
  my $self = shift;
  return $self->{name};
}

sub getQueryLabel # ($self)
{
  my $self = shift;
  return $self->{qLabel};
}

sub getResponseLabel # ($self)
{
  my $self = shift;
  return $self->{resLabel};
}

sub getSendLabel # ($self)
{
  my $self = shift;
  return $self->{sendLabel};
}

sub getTable # ($self)
{
  my $self = shift;
  return $self->{table};
}

sub getInputLabel # ($self)
{
  my $self = shift;
  return $self->{table}->getInputLabel();
}

sub getOutputLabel # ($self)
{
  my $self = shift;
  return $self->{table}->getOutputLabel();
}

sub getPreLabel # ($self)
{
  my $self = shift;
  return $self->{table}->getPreLabel();
}

# add a factory to the Unit type
package Triceps::Unit;

sub makeTableQuery2 # ($self, $tabType, $name)
{
  return TableQuery2->new(@_);
}
```

The meat of the logic stays the same. The creation of the table and of the client sending label are added around it, as well as a bunch of getter methods to get access to the components.

The output of this example is the same, with the only difference that it expects and sends different label names:

```
window,OP_INSERT,1,AAA,10,10
window,OP_INSERT,3,AAA,20,20
```

```
window.query,OP_INSERT
window,OP_INSERT,5,AAA,30,30
window.query,OP_INSERT
window.response,OP_INSERT,1,AAA,10,10
window.response,OP_INSERT,3,AAA,20,20
window.response,OP_NOP,,,,
window.response,OP_INSERT,3,AAA,20,20
window.response,OP_INSERT,5,AAA,30,30
window.response,OP_NOP,,,,
```

10.5. Template options

Often the arguments of the template constructor become more convenient to organize in the option name-value pairs. It becomes particularly useful when there are many arguments and/or when some of them really are optional. For our little query template this is not the case but it can be written with options nevertheless (a modification of the original version, without the table in it):

```
package Query3;

sub new # ($class, $optionName => $optionValue ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    table => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "Triceps::Table") } ],
  }, @_);

  my $name = $self->{name};

  my $table = $self->{table};
  my $unit = $table->getUnit();
  my $rt = $table->getRowType();

  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{inLabel} = $unit->makeLabel($rt, $name . ".in", undef, sub {
    # This version ignores the row contents, just dumps the table.
    my ($label, $rop, $self) = @_;
    my $rh = $self->{table}->begin();
    for (; !$rh->isNull(); $rh = $rh->next()) {
      $self->{unit}->call(
        $self->{outLabel}->makeRowop("OP_INSERT", $rh->getRow()));
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{outLabel}, "OP_NOP");
  }, $self);
  $self->{outLabel} = $unit->makeDummyLabel($rt, $name . ".out");

  bless $self, $class;
  return $self;
}
```

The getter methods stayed the same, so I've skipped them here. The call has changed:

```
my $query = Query3->new(table => $tWindow, name => "qWindow");
```

The output stayed the same.

The class Triceps::Opt is used to parse the arguments formatted as options. There is actually a similar option parser in CPAN but it didn't do everything I wanted, and considering how tiny it is, it's easier to write a new one from scratch than to extend that one. I also like to avoid the extra dependencies.

The heart of it is the method `Triceps::Opt::parse()`. It's normally called from a class constructor to parse the constructor's options, but can be called from the other functions as well. It does the following:

- Checks that all the options are known.
- Checks that the values are acceptable.
- Copies the values into the instance hash of the calling class.
- Provides the default values for the unspecified options.

If anything goes wrong, it confesses with a reasonable message. The arguments tell the class name for the messages (since, remember, it is normally called from the class constructor), the reference to the object instance hash where to copy the options, the descriptions of the supported options, and the actual key-value pairs.

At the end of it, if all went well, the query's `$self` will have the values at keys “name” and “table”.

The options descriptions go in pairs of option name and an array reference with description. The array contains the default value and the checking function, either of which may be `undef`. The checking function returns if everything went fine or confesses on any errors. To die happily with a proper message, it gets not only the value to check but more, altogether:

- The value to check.
- The name of the option.
- The name of the class, for error messages.
- The object instance (`$self`), just in case.

If you want to do multiple checks, you just make a closure and call all the checks in sequence, passing `@_` to them all, like shown here for the option “table”. If more arguments need to be passed to the checking function, just add them after `@_` (or, if you prefer, before it, if you write your checking function that way).

You can create any checking functions, but a few ready ones are provided:

- `Triceps::Opt::ck_mandatory` checks that the value is defined.
- `Triceps::Opt::ck_ref` checks that the value is a reference to a particular class, or a class derived from it. Just give the class name as the extra argument. Or, to check that the reference is to array or hash, make the argument `"ARRAY"` or `"HASH"`. Or an empty string `""` to check that it's not a reference at all. For the arrays and hashes it can also check the values contained in them for being references to the correct types: give that type as the second extra argument. But it doesn't go deeper than that, just one nesting level. It might be extended later, but for now one nesting level has been enough.
- `Triceps::Opt::ck_refscalar` checks that the value is a reference to a scalar. This is designed to check the arguments which are used to return data back to the caller, and it would accept any previous value in that scalar: an actual scalar value, an `undef` or a reference, since it's about to be overwritten anyway.

The `ck_ref()` and `ck_refscalar()` allow the value to be undefined, so they can safely be used on the truly optional options. When I come up with more of the useful check functions, I'll add them.

Triceps::Opt provides more helper functions to deal with options after they have been parsed. One of them is `handleUnitTypeLabel()` that handles a very specific but frequently occuring case: Depending on the usage, sometimes it's more convenient to give the template the input row type and unit, and later chain its input to another label; and sometimes it's more convenient to give it another ready label and have the template find out the row type and unit from it, and chain its input to that label automatically, like `ServerHelpers::makeServerOutLabel()` was shown doing in Section 10.3: "Simple wrapper templates" (p. 111) . It's possible if the unit, row type and source label are made the optional options.

`Triceps::Opt::handleUnitTypeLabel()` takes care of sorting out what information is available, that enough of it is available, that exactly one of row type or source label options is specified, and fills in the unit and row type values from the source label (specifying the unit option along with the source label is OK as long as the unit is the same). To show it off, I re-wrote the `ServerHelpers::makeServerOutLabel()` as a class with options:

```
package ServerOutput;
use Carp;

# Sending of rows to the server output.
sub new # ($class, $option => $value, ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, undef ],
    unit => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Unit") } ],
    rowType => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::RowType") } ],
    fromLabel => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Label") } ],
  }, @_);

  &Triceps::Opt::handleUnitTypeLabel("$class::new",
    unit => \$self->{unit},
    rowType => \$self->{rowType},
    fromLabel => \$self->{fromLabel}
  );
  my $fromLabel = $self->{fromLabel};

  if (!defined $self->{name}) {
    confess "$class::new: must specify at least one of the options name and fromLabel"
      unless (defined $self->{fromLabel});
    $self->{name} = $fromLabel->getName() . ".serverOut";
  }

  my $lb = $self->{unit}->makeLabel($self->{rowType},
    $self->{name}, undef, sub {
      &main::outCurBuf(join(",",
        $fromLabel? $fromLabel->getName() : $self->{name},
        &Triceps::opcodeString($_[1]->getOpcode()),
        $_[1]->getRow()->toArray()) . "\n");
    }, $self # $self is not used in the function but used for cleaning
  );
  $self->{inLabel} = $lb;
  if (defined $fromLabel) {
    $fromLabel->chain($lb) or confess "$!";
  }

  bless $self, $class;
  return $self;
}

sub getInputLabel() # ($self)
{
  my $self = shift;
  return $self->{inLabel};
}
```

The arguments to `Triceps::Opt::handleUnitTypeLabel()` are the caller function name for the error messages, and the pairs of option name and reference to the option value for the unit, row type and the source label.

The new class also has the optional option “name”. If it's not specified and “fromLabel” is specified, the name is generated by appending a suffix to the name of the source label. The new class can be used in one of two ways, either

```
my $srvout = ServerOutput->new(fromLabel => $query->getOutputLabel());
```

or

```
my $srvout = ServerOutput->new(
  name => "out",
  unit => $uTrades,
  rowType => $tWindow->getRowType(),
);
$query->getOutputLabel()->chain($srvout->getInputLabel())
  or confess "$!";
```

The second form comes handy if you want to create it before creating the query.

The other helper function is `Triceps::Opt::checkMutuallyExclusive()`. It checks that no more than one option from the list is specified. The joins use it to allow multiple ways to specify the join condition. For now I'll show a bit contrived example, rewriting the last example of ServerOutput with it:

```
package ServerOutput2;
use Carp;

# Sending of rows to the server output.
sub new # ($class, $option => $value, ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, undef ],
    unit => [ undef, sub { &Triceps::Opt::ck_mandatory; &Triceps::Opt::ck_ref(@_,
 "Triceps::Unit") } ],
    rowType => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::RowType") } ],
    fromLabel => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Label") } ],
  }, @_);

  my $fromLabel = $self->{fromLabel};
  if (&Triceps::Opt::checkMutuallyExclusive("$class::new", 1,
      rowType => $self->{rowType},
      fromLabel => $self->{fromLabel}
    ) eq "fromLabel"
  ) {
    $self->{rowType} = $fromLabel->getRowType();
  }

  if (!defined $self->{name}) {
    confess "$class::new: must specify at least one of the options name and fromLabel"
      unless (defined $self->{fromLabel});
    $self->{name} = $fromLabel->getName() . ".serverOut";
  }

  my $lb = $self->{unit}->makeLabel($self->{rowType},
    $self->{name}, undef, sub {
      &main::outCurBuf(join(",",
        $fromLabel? $fromLabel->getName() : $self->{name},
        &Triceps::opcodeString($_[1]->getOpcode()),
        $_[1]->getRow()->toArray()) . "\n");
    }, $self # $self is not used in the function but used for cleaning
  );
  $self->{inLabel} = $lb;
  if (defined $fromLabel) {
    $fromLabel->chain($lb) or confess "$!";
  }

  bless $self, $class;
  return $self;
}
```

The arguments of the `Triceps::Opt::checkMutuallyExclusive()` are the caller name for error messages, flag whether one of the mutually exclusive options must be specified, and the pairs of option names and values (this time not references, just values). It returns the name of the only option specified by the user, or `undef` if none were. If more than one option was used, or if none were used and the mandatory flag is set, the function will confess.

The way this version of the code works, the option “unit” must be specified in any case, so the use case with the source label becomes:

```
my $srvout = ServerOutput2->new(
  unit => $uTrades,
  fromLabel => $query->getOutputLabel()
);
```

The use case with the independent creation is the same as with the previous version of the `ServerOutput`.

10.6. Code generation in the templates

Suppose we want to filter the result of the query by the equality to the fields in the query request row. The list of the fields would be given to the query template. The query code would check if these fields are not NULL (and since the simplistic CSV parsing is not good enough to tell between NULL and empty values, not an empty value either), and pass only the rows that match it. Here we go (skipping the methods that are the same as before):

```
package Query4;
use Carp;

sub new # ($class, $optionName => $optionValue ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    table => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "Triceps::Table") } ],
    fields => [ undef, sub { &Triceps::Opt::ck_ref(@_, "ARRAY") } ],
  }, @_);

  my $name = $self->{name};

  my $table = $self->{table};
  my $unit = $table->getUnit();
  my $rt = $table->getRowType();

  my $fields = $self->{fields};
  if (defined $fields) {
    my %rtdef = $rt->getdef();
    foreach my $f (@$fields) {
      my $t = $rtdef{$f};
      confess "$class::new: unknown field '$f', the row type is:\n"
          . $rt->print() . " "
```

```
        unless defined $t;
    }
  }

  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{inLabel} = $unit->makeLabel($rt, $name . ".in", undef, sub {
    my ($label, $rop, $self) = @_;
    my $query = $rop->getRow();
    my $cmp = $self->{compare};
    my $rh = $self->{table}->begin();
    ITER: for (; !$rh->isNull(); $rh = $rh->next()) {
      if (defined $self->{fields}) {
        my $data = $rh->getRow();
        my %rtdef = $self->{table}->getRowType()->getdef();
        foreach my $f (@{$self->{fields}}) {
          my $v = $query->get($f);
          # Since the simplified CSV parsing in the mainLoop() provides
          # no easy way to send NULLs, consider any empty or 0 value
          # in the query row equivalent to NULLs.
          if ($v
          && (&Triceps::Fields::isStringType($rtdef{$f})
            ? $query->get($f) ne $data->get($f)
            : $query->get($f) != $data->get($f)
            )
          ) {
            next ITER;
          }
        }
      }
      $self->{unit}->call(
        $self->{outLabel}->makeRowop("OP_INSERT", $rh->getRow()));
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{outLabel}, "OP_NOP");
  }, $self);
  $self->{outLabel} = $unit->makeDummyLabel($rt, $name . ".out");

  bless $self, $class;
  return $self;
}
```

Used as:

```
my $query = Query4->new(table => $tWindow, name => "qWindow",
  fields => ["symbol", "price"]);
```

The field names get checked up front for correctness. And then at run time the code iterates through them and does the checking. Since the comparisons have to be done differently for the string and numeric values, `Triceps::Fields::isStringType()` is used to check the type of the fields. Triceps::Fields is a collection of functions that help dealing with fields in the templates. Another similar function is `Triceps::Fields::isArrayType()`

If the option “fields” is not specified, it would work the same as before and produce the same result. For the filtering by symbol and price, a sample output is:

```
tWindow,OP_INSERT,1,AAA,10,10
tWindow,OP_INSERT,3,AAA,20,20
tWindow,OP_INSERT,4,BBB,20,20
qWindow,OP_INSERT
tWindow,OP_INSERT,5,AAA,30,30
qWindow,OP_INSERT,5,AAA,0,0
qWindow,OP_INSERT,0,,20,0
```

```
qWindow.out,OP_INSERT,1,AAA,10,10
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,4,BBB,20,20
qWindow.out,OP_NOP,,,,
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,5,AAA,30,30
qWindow.out,OP_NOP,,,,
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,4,BBB,20,20
qWindow.out,OP_NOP,,,,
```

The table data now has one more row of data added to it, with the symbol “BBB”. The first query has no values to filter in it, so it just dumps the whole table as before. The second query filters by the symbol “AAA”. The field for price is 0, so it gets treated as empty and excluded from the comparison. The fields for id and size are not in the fields option, so they get ignored even if the value of id is 5. The third query filters by the price equal to 20. The symbol field is empty in the query, so it does not participate in the filtering.

Looking at the query execution code, now there is a lot more going on in it. And quite a bit of it is static, that could be computed at the time the query object is created. The next version does that, building and compiling the comparator function in advance:

```
package Query5;
use Carp;

sub new # ($class, $optionName => $optionValue ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    table => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "Triceps::Table") } ],
    fields => [ undef, sub { &Triceps::Opt::ck_ref(@_, "ARRAY") } ],
    saveCodeTo => [ undef, \&Triceps::Opt::ck_refscalar ],
  }, @_);

  my $name = $self->{name};

  my $table = $self->{table};
  my $unit = $table->getUnit();
  my $rt = $table->getRowType();

  my $fields = $self->{fields};
  if (defined $fields) {
    my %rtdef = $rt->getdef();

    # Generate the code of the comparison function by the fields.
    # Since the simplified CSV parsing in the mainLoop() provides
    # no easy way to send NULLs, consider any empty or 0 value
    # in the query row equivalent to NULLs.
    my $gencmp = '
      sub # ($query, $data)
      {
        use strict;
        my ($query, $data) = @_;';
    foreach my $f (@$fields) {
      my $t = $rtdef{$f};
      confess "$class::new: unknown field '$f', the row type is:\n"
          . $rt->print() . " "
        unless defined $t;
      $gencmp .= '
        my $v = $query->get("' . quotemeta($f) . '");
        if ($v) {';
      if (&Triceps::Fields::isStringType($t)) {
        $gencmp .= '
          return 0 if ($v ne $data->get("' . quotemeta($f) . '"));';
      } else {
        $gencmp .= '
          return 0 if ($v != $data->get("' . quotemeta($f) . '"));';
      }
      $gencmp .= '
        }';
    }
    $gencmp .= '
        return 1; # all succeeded
      }';

    ${$self->{saveCodeTo}} = $gencmp if (defined($self->{saveCodeTo}));
    $self->{compare} = eval $gencmp;
    confess("Internal error: $class failed to compile the comparator:\n$@\nfunction text:
\n$gencmp ")
      if $@;
  }

  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{inLabel} = $unit->makeLabel($rt, $name . ".in", undef, sub {
    my ($label, $rop, $self) = @_;
    my $query = $rop->getRow();
    my $cmp = $self->{compare};
    my $rh = $self->{table}->begin();
    for (; !$rh->isNull(); $rh = $rh->next()) {
      if (!defined $cmp || &$cmp($query, $rh->getRow())) {
        $self->{unit}->call(
          $self->{outLabel}->makeRowop("OP_INSERT", $rh->getRow()));
      }
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{outLabel}, "OP_NOP");
  }, $self);
  $self->{outLabel} = $unit->makeDummyLabel($rt, $name . ".out");

  bless $self, $class;
  return $self;
}
```

The code of the anonymous comparison function gets generated in `$gencmp` and then compiled by using `eval`. `eval` returns the pointer to the compiled function which is then used at run time. The generation uses all the same logic to decide on the string or numeric comparisons, and also effectively unrolls the loop. When generating the string constants in functions from the user-supplied values, it's important to enquote them with `quotemeta()`. Even when we're talking about the field names, they still could have some funny characters in them. The option “saveCodeTo” can be used to get the source code of the comparator, it gets saved at the reference after it gets generated.

If the filter field option is not used, the comparator remains undefined.

The use of this version is the same as of the previous one, but to show the source code of the comparator, I've added its printout:

```
my $cmpcode;
my $query = Query5->new(table => $tWindow, name => "qWindow",
  fields => ["symbol", "price"], saveCodeTo => \$cmpcode );
```

```
# as a demonstration
print("Code:\n$cmpcode\n");
```

This produces the result:

```
Code:

      sub # ($query, $data)
      {
        use strict;
        my ($query, $data) = @_;
        my $v = $query->get("symbol");
        if ($v) {
          return 0 if ($v ne $data->get("symbol"));
        }
        my $v = $query->get("price");
        if ($v) {
          return 0 if ($v != $data->get("price"));
        }
        return 1; # all succeeded
      }
tWindow,OP_INSERT,1,AAA,10,10
tWindow,OP_INSERT,3,AAA,20,20
tWindow,OP_INSERT,4,BBB,20,20
qWindow,OP_INSERT
tWindow,OP_INSERT,5,AAA,30,30
qWindow,OP_INSERT,5,AAA,0,0
qWindow,OP_INSERT,0,,20,0
qWindow.out,OP_INSERT,1,AAA,10,10
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,4,BBB,20,20
qWindow.out,OP_NOP,,,,
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,5,AAA,30,30
qWindow.out,OP_NOP,,,,
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,4,BBB,20,20
qWindow.out,OP_NOP,,,,
```

Besides the code printout, the result is the same as last time.

Now, why list the fields in an option? Why not just take them all? After all, if the user doesn't want filtering on some field, he can always simply not set it in the query row. If the efficiency is a concern, with possibly hundreds of fields in the row with only few of them used for filtering, we can do better: we can generate and compile the comparison function after we see the query row. Here goes the next version that does all this:

```
package Query6;
use Carp;

sub new # ($class, $optionName => $optionValue ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    table => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "Triceps::Table") } ],
  }, @_);

  my $name = $self->{name};
  my $table = $self->{table};
  my $unit = $table->getUnit();
  my $rt = $table->getRowType();

  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{inLabel} = $unit->makeLabel($rt, $name . ".in", undef, sub {
    my ($label, $rop, $self) = @_;
    my $query = $rop->getRow();
    my $cmp = $self->genComparison($query);
    my $rh = $self->{table}->begin();
    for (; !$rh->isNull(); $rh = $rh->next()) {
      if (&$cmp($query, $rh->getRow())) {
        $self->{unit}->call(
          $self->{outLabel}->makeRowop("OP_INSERT", $rh->getRow()));
      }
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{outLabel}, "OP_NOP");
  }, $self);
  $self->{outLabel} = $unit->makeDummyLabel($rt, $name . ".out");

  bless $self, $class;
  return $self;
}

# Generate the comparison function on the fly from the fields in the
# query row.
# Since the simplified CSV parsing in the mainLoop() provides
# no easy way to send NULLs, consider any empty or 0 value
# in the query row equivalent to NULLs.
sub genComparison # ($self, $query)
{
  my $self = shift;
  my $query = shift;

  my %qhash = $query->toHash();
  my %rtdef = $self->{table}->getRowType()->getdef();
  my ($f, $v);

  my $gencmp = '
      sub # ($query, $data)
      {
        use strict;';

  while (($f, $v) = each %qhash) {
    next unless($v);
    my $t = $rtdef{$f};

    if (&Triceps::Fields::isStringType($t)) {
      $gencmp .= '
        return 0 if ($_[0]->get("' . quotemeta($f) . '")
          ne $_[1]->get("' . quotemeta($f) . '"));';
    } else {
      $gencmp .= '
        return 0 if ($_[0]->get("' . quotemeta($f) . '")
          != $_[1]->get("' . quotemeta($f) . '"));';
    }
  }
  $gencmp .= '
        return 1; # all succeeded
      }';

  my $compare = eval $gencmp;
  confess("Internal error: Query '" . $self->{name}
      . "' failed to compile the comparator:\n$@\nfunction text:\n$gencmp ")
    if $@;

  # for debugging
  &main::outCurBuf("Compiled comparator:\n$gencmp\n");

  return $compare;
}
```

Thie option “fields” is gone, and the code generation has moved into the method `genComparison()`, that gets called for each query. I've inserted the sending back of the comparison source code at the end of it, to make it easier to understand. Obviously, if this code were used in production, this would have to be commented out, and maybe some better option added for debugging. An example of the output is:

```
tWindow,OP_INSERT,1,AAA,10,10
tWindow,OP_INSERT,3,AAA,20,20
tWindow,OP_INSERT,4,BBB,20,20
qWindow,OP_INSERT
tWindow,OP_INSERT,5,AAA,30,30
qWindow,OP_INSERT,5,AAA,0,0
qWindow,OP_INSERT,0,,20,0
Compiled comparator:

      sub # ($query, $data)
      {
        use strict;
        return 1; # all succeeded
      }
qWindow.out,OP_INSERT,1,AAA,10,10
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,4,BBB,20,20
qWindow.out,OP_NOP,,,,
Compiled comparator:

      sub # ($query, $data)
      {
        use strict;
        return 0 if ($_[0]->get("symbol")
          ne $_[1]->get("symbol"));
        return 0 if ($_[0]->get("id")
          != $_[1]->get("id"));
        return 1; # all succeeded
      }
qWindow.out,OP_INSERT,5,AAA,30,30
qWindow.out,OP_NOP,,,,
Compiled comparator:

      sub # ($query, $data)
      {
        use strict;
        return 0 if ($_[0]->get("price")
          != $_[1]->get("price"));
        return 1; # all succeeded
      }
qWindow.out,OP_INSERT,3,AAA,20,20
qWindow.out,OP_INSERT,4,BBB,20,20
qWindow.out,OP_NOP,,,,
```

The first query contains no filter fields, so the function compiles to the constant 1. The second query has the fields id and symbol not empty, so the filtering goes by them. The third query has only the price field, and it is used for filtering.

The code generation on the fly is a powerful tool and is used throughout Triceps.

10.7. Result projection in the templates

The other functionality provided by the Triceps::Fields is the filtering of the fields in the result row type, also known as “projection”. You can select which fields you want and which you don't want, and rename the fields.

To show how it's done, I took the Query3 example from Section 10.5: “Template options” (p. 117) and added the result field filtering to it. I've also changed the format in which it returns the results to `printP()`, to show the field names and make the effects of the field renaming visible.

```
package Query7;

sub new # ($class, $optionName => $optionValue ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    table => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "Triceps::Table") } ],
    resultFields => [ undef, sub { &Triceps::Opt::ck_ref(@_, "ARRAY", ""); } ],
  }, @_);

  my $name = $self->{name};

  my $table = $self->{table};
  my $unit = $table->getUnit();
  my $rtIn = $table->getRowType();
  my $rtOut = $rtIn;

  if (defined $self->{resultFields}) {
    my @inFields = $rtIn->getFieldNames();
    my @pairs =  &Triceps::Fields::filterToPairs($class, \@inFields, $self-
>{resultFields});
    ($rtOut, $self->{projectFunc}) = &Triceps::Fields::makeTranslation(
      rowTypes => [ $rtIn ],
      filterPairs => [ \@pairs ],
    );
  } else {
    $self->{projectFunc} = sub {
      return $_[0];
    }
  }

  $self->{unit} = $unit;
  $self->{name} = $name;
  $self->{inLabel} = $unit->makeLabel($rtIn, $name . ".in", undef, sub {
    # This version ignores the row contents, just dumps the table.
    my ($label, $rop, $self) = @_;
    my $rh = $self->{table}->begin();
    for (; !$rh->isNull(); $rh = $rh->next()) {
      $self->{unit}->call(
        $self->{outLabel}->makeRowop("OP_INSERT",
          &{$self->{projectFunc}}($rh->getRow())));
    }
    # The end is signaled by OP_NOP with empty fields.
    $self->{unit}->makeArrayCall($self->{outLabel}, "OP_NOP");
  }, $self);
  $self->{outLabel} = $unit->makeDummyLabel($rtOut, $name . ".out");

  bless $self, $class;
  return $self;
}

sub getInputLabel # ($self)
{
  my $self = shift;
  return $self->{inLabel};
}

sub getOutputLabel # ($self)
{
  my $self = shift;
  return $self->{outLabel};
}

sub getName # ($self)
{
  my $self = shift;
  return $self->{name};
}

package main;

my $uTrades = Triceps::Unit->new("uTrades");
my $tWindow = $uTrades->makeTable($ttWindow, "EM_CALL", "tWindow")
  or confess "$!";
my $query = Query7->new(table => $tWindow, name => "qWindow",
  resultFields => [ '!id', 'size/lot_$&', '.*' ],
);
# print in the tokenized format
my $srvout = $uTrades->makeLabel($query->getOutputLabel()->getType(),
  $query->getOutputLabel()->getName() . ".serverOut", undef, sub {
    &main::outCurBuf($_[1]->printP() . "\n");
  });
$query->getOutputLabel()->chain($srvout) or confess "$!";

my %dispatch;
$dispatch{$tWindow->getName()} = $tWindow->getInputLabel();
$dispatch{$query->getName()} = $query->getInputLabel();
$dispatch{"exit"} = &ServerHelpers::makeExitLabel($uTrades, "exit");

run(\%dispatch);
```

The query now has the new option “resultFields” that defines the projection. That option accepts a reference to an array of pattern strings. If present, it gives the patterns of the fields to let through. The patterns may be either the explicit field names or regular expressions implicitly anchored at both front and back. There is also a bit of extra modification possible:

! *pattern*
: Skip the fields matching the pattern.

pattern / *substitution*
: Pass the matching fields and rename them according to the substitution.

So in this example `[ '!id', 'size/lot_$&', '.*' ]` means: skip the field “id”, rename the field “size” by prepending “lot_” to it, and pass through the rest of the fields. In the renaming pattern, $& is the reference to the whole original field name. If you use the parenthesised groups, they are referred to as `$1`, `$2` and so on. But if you use any of those, don't forget to put the pattern into single quotes to prevent the unwanted expansion in the double quotes before the projection gets a chance to see it.

For an example of why the parenthesised groups can be useful, suppose that the row type has multiple account-related elements that all start with “acct”: `acctsrc`, `acctinternal`, `acctexternal`. Suppose we want to insert an underscore after “acct”. This can be achieved with the pattern `'acct(.*)/acct_$1'`. As usual in the Perl regexps, the parenthesised groups are numbered left to right, starting with `$1`.

If a specification element refers to a literal field, like here “id” and “size”, the projection checks that the field is actually present in the original row type, catching the typos. For the general regular expressions it doesn't check whether the pattern matched anything. It's not difficult to check but that would preclude the reuse of the same patterns on the varying row types, and I'm not sure yet, what is more important.

The way this whole thing works is that each field gets tested against each pattern in order. The first pattern that matches determines what happens to this field. If none of the patterns matches, the field gets ignored. An important consequence about the skipping patterns is that they don't automatically pass through the non-matching fields. You need to add an explicit positive pattern at the end of the list to pass the fields through. `'.*'` serves this purpose in the example.

A consequence is that the order of the fields can't be changed by the projection. They are tested in the order they appear in the original row type, and are inserted into the projected row type in the same order.

Another important point is that the field names in the result must not duplicate. It would be an error. Be careful with the substitution syntax to avoid creating the duplicate names.

A run example from this version, with the same input as before:

```
tWindow,OP_INSERT,1,AAA,10,10
tWindow,OP_INSERT,3,AAA,20,20
qWindow,OP_INSERT
tWindow,OP_INSERT,5,AAA,30,30
qWindow,OP_INSERT
qWindow.out OP_INSERT symbol="AAA" price="10" lot_size="10"
qWindow.out OP_INSERT symbol="AAA" price="20" lot_size="20"
qWindow.out OP_NOP
qWindow.out OP_INSERT symbol="AAA" price="20" lot_size="20"
qWindow.out OP_INSERT symbol="AAA" price="30" lot_size="30"
qWindow.out OP_NOP
```

The rows returned are the same, but projected and printed in the `printP()` format.

Inside the template the projection works in three steps:

- `Triceps::Fields::filterToPairs()` does the projection of the field names and returns its result as an array of names. The names in the array go in pairs: the old name and the new name in each pair. The fields that got skipped do not get included in the list. In this example the array would be `( "symbol", "symbol", "price", "price", "size", "lot_size" )`.

- `Triceps::Fields::makeTranslation()` then takes this array along with the original row type and produces the result row type and a function reference that does the projection by converting an original row into the projected one.

- The template execution then calls this projection function for the result rows.

The split of work between `filterToPairs()` and `makeTranslation()` has been done partially historically and partially because sometimes you may want to just get the pair names array and then use them on your own instead of calling `makeTranslation()`. There is one more function that you may find useful if you do the handling on your own: `filter()`. It takes the same arguments and does the same thing as `filterToPairs()` but returns the result in a different format. It's still an array of strings but it contains only the names of the translated field names instead of the pairs, in the order matching the order of the original fields. For the fields that have been skipped it contains an `undef`. For this example it would return `( undef, "symbol", "price", "lot_size" )`.

The calls are:

```
@fields = &Triceps::Fields::filter(
  $caller, \@inFields, \@translation);
@pairs = &Triceps::Fields::filterToPairs(
  $caller, \@inFields, \@translation);
($rowType, $projectFunc) = &Triceps::Fields::makeTranslation(
  $optName => $optValue, ...);
```

All of them confess on errors, and the argument `$caller` is used for building the error messages. The options of `makeTranslations()` are:

“rowTypes” is a reference to an array of original row types. “filterPairs” is a reference to an array of filter pair arrays. Both of these options are mandatory. And that's right, `makeTranslations()` can accept and merge more than one original row type, with a separate projection specification for each of them. It's not quite as flexible as I'd want it to be, not allowing to reorder and mix the fields from different originals (now the fields go in sequence: from the first original, from the second original, and so on), but it's a decent start. When you combine multiple original row types, you need to be particularly careful with avoiding the duplicate field names in the result.

The option “saveCodeTo” also allows to save the source code of the generated function, same as in the Query5 example in Section 10.6: “Code generation in the templates” (p. 121) .

The general call form of `makeTranslations()` is:

```
($rowType, $projectFunc) = &Triceps::Fields::makeTranslation(
  rowTypes => [ $rt1, $rt2, ..., $rtN ],
  filterPairs => [ \@pairs1, \@pairs2, ..., \@pairsN ],
  saveCodeTo => \$codeVar,
);
```

One of the result type or projection function referece could have also been returned to a place pointed to by an option, like “saveCodeTo”, but since Perl supports returning multiple values from a function, that looks simpler and cleaner.

The projection function is then called:

```
$row = &$projectFunc($origRow1, $origRow2, ..., $origRowN);
```

Naturally, `makeTranslations()` is a template itself. Let's look at its source code, it shows a new trick.

```
package Triceps::Fields;
use Carp;

use strict;

sub makeTranslation # (optName => optValue, ...)
{
  my $opts = {}; # the parsed options
  my $myname = "Triceps::Fields::makeTranslation";

  &Triceps::Opt::parse("Triceps::Fields", $opts, {
      rowTypes => [ undef, sub { &Triceps::Opt::ck_mandatory(@_);
 &Triceps::Opt::ck_ref(@_, "ARRAY", "Triceps::RowType") } ],
      filterPairs => [ undef, sub { &Triceps::Opt::ck_mandatory(@_);
 &Triceps::Opt::ck_ref(@_, "ARRAY", "ARRAY") } ],
      saveCodeTo => [ undef, sub { &Triceps::Opt::ck_refscalar(@_) } ],
    }, @_);

  # reset the saved source code
  ${$opts->{saveCodeTo}} = undef if (defined($opts->{saveCodeTo}));

  my $rts = $opts->{rowTypes};
  my $fps = $opts->{filterPairs};

  confess "$myname: the arrays of row types and filter pairs must be of the same size, got
 " . ($#{$rts}+1) . " and " . ($#{$fps}+1) . " elements"
    unless ($#{$rts} == $#{$fps});

  my $gencode = '
    sub { # (@rows)
      use strict;
      use Carp;
      confess "template internal error in ' . $myname  . ': result translation expected
 ' . ($#{$rts}+1) . ' row args, received " . ($#_+1)
        unless ($#_ == ' . $#{$rts} . ');
      # $result_rt comes at compile time from Triceps::Fields::makeTranslation
      return $result_rt->makeRowArray(';

  my @rowdef; # of the result row type
  for (my $i = 0; $i <= $#{$rts}; $i++) {
    my %origdef = $rts->[$i]->getdef();
    my @fp = @{$fps->[$i]}; # copy the array, because it will be shifted
    while ($#fp >= 0) {
      my $from = shift @fp;
      my $to = shift @fp;
      my $type = $origdef{$from};
      confess "$myname: unknown original field '$from' in the original row type $i:\n" .
 $rts->[$i]->print() . " "
        unless (defined $type);
      push(@rowdef, $to, $type);
      $gencode .= '
        $_[' . $i . ']->get("' . quotemeta($from) . '"),';
    }
  }

  $gencode .= '
      );
    }';

  my $result_rt = Triceps::RowType->new(@rowdef)
    or confess "$myname: Invalid result row type specification: $! ";

  ${$opts->{saveCodeTo}} = $gencode if (defined($opts->{saveCodeTo}));

  # compile the translation function
  my $func = eval $gencode
    or confess "$myname: error in compilation of the function:\n  $@\nfunction text:\n
$gencode ";

  return ($result_rt, $func);
}
```

By now almost all the parts of the implementation should look familiar to you. It builds the result row definition and the projection function code in parallel by iterating through the originals. An interesting trick is done with passing the result row type into the projection function. The function needs it to create the result rows. But it can't be easily placed into the function source code. So the closure property of the projection function is used: whatever outside “my” variables occur in the function at the time when it's compiled, will have their values compiled hardcoded into the function. So the “my” variable `$result_rt` is set with the result row type, and then the projection function gets compiled. The projection function refers to `$result_rt`, which gets picked up from the parent scope and hardcoded in the closure.

Chapter 11. Aggregation

11.1. The ubiquitous VWAP

Every CEP supplier loves an example of VWAP calculation: it's small, it's about that quintessential CEP activity: aggregation, and it sounds like something from the real world.

A quick sidebar: what is the VWAP? It's the Value-Weighted Average Price: the average price for the shares traded during some period of time, usually a day. If you take the price of every share traded during the day and calculate the average, you get the VWAP. What is the value-weighted part? The shares don't usually get sold one by one. They're sold in the variable-sized lots. If you think in the terms of lots and not individual shares, you have to weigh the trade prices (not to be confused with costs) for the lots proportional to the number of shares in them.

I've been using VWAP for trying out the different approaches to the aggregation. There are multiple ways to do it, from fully manual, to the aggregator infrastructure with manual computation of the aggregations, to the simple aggregation functions. The cutest version of VWAP so far is implemented as a user-defined aggregation function for the SimpleAggregator. Here is how it goes:

```
# VWAP function definition
my $myAggFunctions = {
  myvwap => {
    vars => { sum => 0, count => 0, size => 0, price => 0 },
    step => '($%size, $%price) = @$%argiter; '
      . 'if (defined $%size && defined $%price) '
        . '{$%count += $%size; $%sum += $%size * $%price;}',
    result => '($%count == 0? undef : $%sum / $%count)',
  },
};

my $uTrades = Triceps::Unit->new("uTrades");

# the input data
my $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
) or confess "$!";

my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("fifo", Triceps::IndexType->newFifo())
  )
or confess "$!";

# the aggregation result
my $rtVwap;
my $compText; # for debugging

Triceps::SimpleAggregator::make(
  tabType => $ttWindow,
  name => "aggrVwap",
  idxPath => [ "bySymbol", "fifo" ],
  result => [
    symbol => "string", "last", sub {$_[0]->get("symbol");},
    id => "int32", "last", sub {$_[0]->get("id");},
    volume => "float64", "sum", sub {$_[0]->get("size");},
    vwap => "float64", "myvwap", sub { [$_[0]->get("size"), $_[0]->get("price")];},
  ],
  functions => $myAggFunctions,
  saveRowTypeTo => \$rtVwap,
  saveComputeTo => \$compText,
);

$ttWindow->initialize() or confess "$!";
my $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

# label to print the result of aggregation
my $lbPrint = $uTrades->makeLabel($rtVwap, "lbPrint",
  undef, sub { # (label, rowop)
    print($_[1]->printP(), "\n");
  }) or confess "$!";
$tWindow->getAggregatorLabel("aggrVwap")->chain($lbPrint)
  or confess "$!";

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a string opcode
  $uTrades->makeArrayCall($tWindow->getInputLabel(), @data);
  $uTrades->drainFrame(); # just in case, for completeness
}
```

The aggregators get defined as parts of the table type. `Triceps::SimpleAggregator::make()` is a kind of a template that adds an aggregator definition to the table type that is specified in the option “tabType”. An aggeragtor doesn't live in a vacuum, it always works as a part of the table type. As the table gets modified, the aggregator also re-computes its aggregation results. The fine distinction is that the aggregator is a part of the table type, and is common for all the tables of this type. But the table stores its aggregation state, and when an aggregator runs on a table, it uses and modifies that state.

The name of the aggregator is how you can find its result later in the table: each aggregator has an output label created for it, that can be found with `$table->getAggregatorLabel()`. The option “idxPath” defines both the grouping of the rows for this aggregator and their order in the group. The index type at the path determines the order and its parent defines the groups. In this case the grouping happens by symbol, and the rows in the groups go in the FIFO order. This means that the aggregation function `last` will be selecting the row that has been inserted last, in the FIFO order.

The option “result” defines both the row type of the result and the rules for its computation. Each field is defined there with four elements: name, type, aggregation function name, and the function reference to select the value to be aggregated from the row. Triceps provides a bunch of pre-defined aggregation functions like `first`, `last`, `sum`, `count`, `avg` and so on. But VWAP is not one of them (well, maybe now it should be, but then this example would be less interesting). Not to worry, the user can add custom aggregation functions, and that's what this example does.

The option “functions” contains the definitions of such user-defined aggregation functions. Here it defines the function `myvwap`. It defines the state variables that will be used to keep the intermediate values for a group, a step computation, and the result computation. Whenever the group changes, the aggregator will reset the state variables to the default values and iterate through the new contents of the group. It will perform the step computation for each row and collect the data in the intermediate variables. After the iteration it will perform the result computation and produce the final value.

The VWAP computation in a weird one, taking two fields as arguments. These two fields get packed into an array reference by

```
sub { [$_[0]->get("size"), $_[0]->get("price")];}
```

and then the step computation unpacks and handles them. In the aggregator computations the syntax `$%name` refers to the intermediate variables and also to a few pre-defined ones. `$%argiter` is the value extracted from the current row during the iteration.

And that's pretty much it: send the rows to the table, the iterator state gets updated to match the table contents, computes the results and sends them. For example:

```
OP_INSERT,11,abc,123,100
tWindow.aggrVwap OP_INSERT symbol="abc" id="11" volume="100"
    vwap="123"
OP_INSERT,12,abc,125,300
tWindow.aggrVwap OP_DELETE symbol="abc" id="11" volume="100"
    vwap="123"
tWindow.aggrVwap OP_INSERT symbol="abc" id="12" volume="400"
    vwap="124.5"
OP_INSERT,13,def,200,100
tWindow.aggrVwap OP_INSERT symbol="def" id="13" volume="100"
    vwap="200"
OP_INSERT,14,fgh,1000,100
tWindow.aggrVwap OP_INSERT symbol="fgh" id="14" volume="100"
    vwap="1000"
OP_INSERT,15,abc,128,300
tWindow.aggrVwap OP_DELETE symbol="abc" id="12" volume="400"
    vwap="124.5"
tWindow.aggrVwap OP_INSERT symbol="abc" id="15" volume="700"
    vwap="126"
OP_INSERT,16,fgh,1100,25
tWindow.aggrVwap OP_DELETE symbol="fgh" id="14" volume="100"
    vwap="1000"
tWindow.aggrVwap OP_INSERT symbol="fgh" id="16" volume="125"
    vwap="1020"
OP_INSERT,17,def,202,100
tWindow.aggrVwap OP_DELETE symbol="def" id="13" volume="100"
    vwap="200"
tWindow.aggrVwap OP_INSERT symbol="def" id="17" volume="200"
    vwap="201"
OP_INSERT,18,def,192,1000
tWindow.aggrVwap OP_DELETE symbol="def" id="17" volume="200"
    vwap="201"
tWindow.aggrVwap OP_INSERT symbol="def" id="18" volume="1200"
    vwap="193.5"
```

When a group gets modified, the aggregator first sends a DELETE of the old contents, then an INSERT of the new contents. But when the first row gets inserted in a group, there is nothing to delete, and only INSERT is sent. And the opposite, when the last row is deleted from a group, only the DELETE is sent.

After this highlight, let's look at the aggregators from the bottom up.

11.2. Manual aggregation

The table exanmple in Section 9.7: “Secondary indexes” (p. 84) prints the aggregated information (the average price of two records). This can be fairly easily changed to put the information into the rows and send them on as labels. The function `printAverage()` has morphed into `computeAverage()`, while the rest of the example stayed the same and is omitted:

```
our $rtAvgPrice = Triceps::RowType->new(
  symbol => "string", # symbol traded
  id => "int32", # last trade's id
  price => "float64", # avg price of the last 2 trades
) or confess "$!";

# place to send the average: could be a dummy label, but to keep the
# code smaller also print the rows here, instead of in a separate label
our $lbAverage = $uTrades->makeLabel($rtAvgPrice, "lbAverage",
  undef, sub { # (label, rowop)
    print($_[1]->printP(), "\n");
  }) or confess "$!";

# Send the average price of the symbol in the last modified row
sub computeAverage # (row)
{
  return unless defined $rLastMod;
  my $rhFirst = $tWindow->findIdx($itSymbol, $rLastMod) or confess "$!";
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2) or confess "$!";
  print("Contents:\n");
  my $avg = 0;
  my ($sum, $count);
  my $rhLast;
  for (my $rhi = $rhFirst;
      !$rhi->same($rhEnd); $rhi = $rhi->nextIdx($itLast2)) {
    print("  ", $rhi->getRow()->printP(), "\n");
    $rhLast = $rhi;
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  if ($count) {
    $avg = $sum/$count;
    $uTrades->call($lbAverage->makeRowop(&Triceps::OP_INSERT,
      $rtAvgPrice->makeRowHash(
        symbol => $rhLast->getRow()->get("symbol"),
        id => $rhLast->getRow()->get("id"),
        price => $avg
      )
    ));
  }
}

while(<STDIN>) {
  chomp;
  my @data = split(/,/);
  $uTrades->makeArrayCall($tWindow->getInputLabel(), @data);
  &computeAverage();
  undef $rLastMod; # clear for the next iteration
  $uTrades->drainFrame(); # just in case, for completeness
}
```

For the demonstration, the aggregated rows sent to `$lbAverage` get printed. The rows being aggregated are printed during the iteration too, indented after “Contents:”. And here is a sample run's result, with the input records shown in bold:

```
OP_INSERT,1,AAA,10,10
Contents:
  id="1" symbol="AAA" price="10" size="10"
lbAverage OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
Contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
lbAverage OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
Contents:
  id="3" symbol="AAA" price="20" size="20"
  id="5" symbol="AAA" price="30" size="30"
lbAverage OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
Contents:
  id="5" symbol="AAA" price="30" size="30"
lbAverage OP_INSERT symbol="AAA" id="5" price="30"
OP_DELETE,5
Contents:
```

There are a couple of things to notice about it: it produces only the INSERT rowops, no DELETEs, and when the last record of the group is removed, that event produces nothing.

The first item is mildly problematic because the processing downstream from here might not be able to handle the updates properly without the DELETE rowops. It can be worked around fairly easily by connecting another table to store the aggregation results, with the same primary key as the aggregation key. That table would automatically transform the repeated INSERTs on the same key to a DELETE-INSERT sequence.

The second item is actually pretty bad because it means that the last record deleted gets stuck in the aggregation results. The Coral8 solution for this situation is to send a row with all non-key fields set to NULL, to reset them (interestingly, it's a relatively recent addition, that bug took Coral8 years to notice). But with the opcodes available, we can as well send a DELETE rowop with the key fields filled, the helper table will fill in the rest of the fields, and produce a clean DELETE.

All this can be done by the following changes. Add the table, remember its input label in `$lbAvgPriceHelper`. It will be used to send the aggregated rows instead of `$tAvgPrice`. Then still use `$tAvgPrice` to print the records coming out, but now connect it after the helper table. And in `computeAverage()` change the destination label and add the case for when the group becomes empty (`$count == 0`). The rest of the example stays the same.

```
our $ttAvgPrice = Triceps::TableType->new($rtAvgPrice)
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
  )
  or confess "$!";
$ttAvgPrice->initialize() or confess "$!";
our $tAvgPrice = $uTrades->makeTable($ttAvgPrice,
  &Triceps::EM_CALL, "tAvgPrice") or confess "$!";
our $lbAvgPriceHelper = $tAvgPrice->getInputLabel() or confess "$!";

# place to send the average: could be a dummy label, but to keep the
# code smaller also print the rows here, instead of in a separate label
our $lbAverage = makePrintLabel("lbAverage", $tAvgPrice->getOutputLabel());

# Send the average price of the symbol in the last modified row
sub computeAverage2 # (row)
{
  return unless defined $rLastMod;
  my $rhFirst = $tWindow->findIdx($itSymbol, $rLastMod) or confess "$!";
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2) or confess "$!";
  print("Contents:\n");
  my $avg = 0;
  my ($sum, $count);
  my $rhLast;
  for (my $rhi = $rhFirst;
      !$rhi->same($rhEnd); $rhi = $rhi->nextIdx($itLast2)) {
    print("  ", $rhi->getRow()->printP(), "\n");
    $rhLast = $rhi;
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  if ($count) {
    $avg = $sum/$count;
    $uTrades->makeHashCall($lbAvgPriceHelper, &Triceps::OP_INSERT,
      symbol => $rhLast->getRow()->get("symbol"),
      id => $rhLast->getRow()->get("id"),
      price => $avg
    );
  } else {
    $uTrades->makeHashCall($lbAvgPriceHelper, &Triceps::OP_DELETE,
      symbol => $rLastMod->get("symbol"),
    );
  }
}
```

The change is straightforward. The label `$lbAverage` now reverts to just printing the rowops going through it, so it can be created with the template `makePrintLabel()` described in Section 10.3: “Simple wrapper templates” (p. 111) .

Then the output for the same input becomes:

```
OP_INSERT,1,AAA,10,10
Contents:
  id="1" symbol="AAA" price="10" size="10"
tAvgPrice.out OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
Contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
tAvgPrice.out OP_DELETE symbol="AAA" id="1" price="10"
tAvgPrice.out OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
Contents:
  id="3" symbol="AAA" price="20" size="20"
  id="5" symbol="AAA" price="30" size="30"
tAvgPrice.out OP_DELETE symbol="AAA" id="3" price="15"
tAvgPrice.out OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
Contents:
  id="5" symbol="AAA" price="30" size="30"
tAvgPrice.out OP_DELETE symbol="AAA" id="5" price="25"
tAvgPrice.out OP_INSERT symbol="AAA" id="5" price="30"
OP_DELETE,5
Contents:
tAvgPrice.out OP_DELETE symbol="AAA" id="5" price="30"
```

All fixed, the proper DELETEs are coming out. The last line shows the empty group contents in the table but the DELETE row is still coming out.

Why should we worry so much about the DELETEs? Because without them, relying on just INSERTs for updates, it's easy to create bugs. The last example still has an issue with handling the row replacement by INSERTs. Can you spot it from reading the code?

Here is run example that highlights the issue (as usual, the input lines are in bold):

```
OP_INSERT,1,AAA,10,10
Contents:
  id="1" symbol="AAA" price="10" size="10"
tAvgPrice.out OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
Contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
tAvgPrice.out OP_DELETE symbol="AAA" id="1" price="10"
tAvgPrice.out OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
```

```
Contents:
  id="3" symbol="AAA" price="20" size="20"
  id="5" symbol="AAA" price="30" size="30"
tAvgPrice.out OP_DELETE symbol="AAA" id="3" price="15"
tAvgPrice.out OP_INSERT symbol="AAA" id="5" price="25"
OP_INSERT,5,BBB,30,30
Contents:
  id="5" symbol="BBB" price="30" size="30"
tAvgPrice.out OP_INSERT symbol="BBB" id="5" price="30"
OP_INSERT,7,AAA,40,40
Contents:
  id="3" symbol="AAA" price="20" size="20"
  id="7" symbol="AAA" price="40" size="40"
tAvgPrice.out OP_DELETE symbol="AAA" id="5" price="25"
tAvgPrice.out OP_INSERT symbol="AAA" id="7" price="30"
```

The row with id=5 has been replaced to change the symbol from “AAA” to “BBB”. This act changes both the groups of “AAA” and of “BBB”, removing the row from the first one and inserting it into the second one. Yet only the output for “BBB” came out. The printout of the next row with id=7 and symbol=“AAA” shows that the row with id=5 has been indeed removed from the group “AAA”. It even corrects the result. But until that row came in, the average for the symbol “AAA” remained unchanged and incorrect.

There are multiple ways to fix this issue but first it had to be noticed. Which requires a lot of attention to detail. It's much better to avoid these bugs in the first place by sending the clean and nice input.

11.3. Introducing the proper aggregation

Since the manual aggregation is error-prone, Triceps can manage it for you and do it right. The only thing you need to do is do the actual iteration and computation. Here is the rewrite of the same example with a Triceps aggregator:

```
my $uTrades = Triceps::Unit->new("uTrades") or confess "$!";

# the input data
my $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
) or confess "$!";

# the aggregation result
my $rtAvgPrice = Triceps::RowType->new(
  symbol => "string", # symbol traded
  id => "int32", # last trade's id
  price => "float64", # avg price of the last 2 trades
) or confess "$!";

# aggregation handler: recalculate the average each time the easy way
sub computeAverage1 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);

  my $sum = 0;
  my $count = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull();
      $rhi = $context->next($rhi)) {
```

```
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->last()->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $context->send($opcode, $res);
}

my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last2",
      Triceps::IndexType->newFifo(limit => 2)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", undef, \&computeAverage1)
      )
    )
  )
or confess "$!";
$ttWindow->initialize() or confess "$!";
my $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

# label to print the result of aggregation
my $lbAverage = makePrintLabel("lbAverage",
  $tWindow->getAggregatorLabel("aggrAvgPrice"));

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a string opcode
  $uTrades->makeArrayCall($tWindow->getInputLabel(), @data);
  $uTrades->drainFrame(); # just in case, for completeness
}
```

What has changed in this code? The things got rearranged a bit.The aggregator is now defined as a part of the table type, so the aggregation result row type and its computation function had to be moved up.

The AggregatorType object holds the information about the aggregator. In the table type, the aggregator type gets attached to an index type with `setAggregator()`. In this case, to the FIFO index type. The parent of that index type determines the aggregation groups, grouping happening by its combined key fields (that is, all the key fields of all the indexes in the path starting from the root). For aggregation the working or non-working method `getKey()` doesn't matter, so any of the Hashed, Ordered and Sorted index types can be used. The index type where the aggregator type is attached determines the order of the rows in the groups. If you use FIFO, the rows will be in the order of arrival. If you use Ordered or Sorted, the rows will be in the sort order. If you use Hashed, the rows will be in some random order, which is not particularly useful.

At present an index type may have no more than one aggregator type attached to it. There is no particular reason for that, other than that it was slightly easier to implement, and that I can't think yet of a real-word situation where multiple aggregators on the same index would be needed. If this situation will ever occur, this support can be added. However a table type may have multiple aggregator types in it, on different indexes. You can save a reference to an aggregator type in a variable and reuse it in the different table types too (though not multiple times in the same table, since that would cause a naming conflict).

The aggregator type is created with the arguments of

- result row type,
- aggregator name,
- group initialization Perl function (which may be `undef`, as in this example),
- group computation Perl function,
- the optional arguments for the functions.

Note that there is a difference in naming between the aggregator types and index types: an aggregator type knows its name, while an index type does not. An index type is given a name only in its hierarchy inside the table type, but it does not know its name.

When a table is created, it finds all the aggregator types in it, and creates an output label for each of them. The names of the aggregator types are used as suffixes to the table name. In this example the aggregator will have its output label named “tWindow.aggrAvgPrice”. This puts all the aggregator types in the table into the same namespace, so make sure to give them different names in the same table type. Also avoid the names “in”, “out” and “pre” because these are already taken by the table's own labels. The aggregator labels in the table can be found with

```
$aggLabel = $table->getAggregatorLabel("aggName") or confess "$!";
```

The aggregator types are theoretically multithreaded, but for all I can tell, they will not integrate with the Perl multithreading well, due to the way the Perl objects (the execution methods!) are tied to each thread's separate interpreter. In the future expect that the table types with aggregators could not be shared between the threads. But then again, maybe they could be copied between the threads and that would work just as well.

After the logic is moved into a managed aggregator, the main loop becomes simpler.

The computation function gets a lot more arguments than it used to. The most interesting and most basic ones are `$context`, `$opcode`, and `$rh`. The rest are useful in the more complex cases only.

The aggregator type is exactly that: a type. It doesn't know, on which table or index, or even index type it will be used. And indeed, it might be used on multiple tables and index types. But to do the iteration on the rows, the computation function needs to get this information somehow. And it does, in the form of aggregator context. The manual aggregation used the last table output row to find, on which exact group to iterate. The managed aggregator gets the last modified row handle as the argument `$rh`. But our simple aggregator doesn't even need to consult `$rh` because the context takes care of finding the group too: it knows the exact group and exact index that needs to be aggregated (look at the index tree drawings in Section 9.10: “The index tree” (p. 93) for the difference between an index type and an index).

All the aggregator context methods use the new error handling, confessing on errors.

The context provides its own `begin()` and `next()` methods. They are actually slightly more efficient than the usual table iteration methods because they take advantage of that exact known index. The most important part, they work differently.

```
$rhi = $context->next($rhi);
```

returns a NULL row handle when it reaches the end of the group. Do not, I repeat, **DO NOT** use the `$rhi->next()` in the aggregators, or you'll get some very wrong results.

The context also has a bit more of its own magic.

```
$rh = $context->last();
```

returns the last row handle in the group. This comes very handy because in most of the cases you want the data from the last row to fill the fields that haven't been aggregated as such. This is like the SQL function `LAST()`. Using the fields from the argument `$rh`, unless they are the key fields for this group, is generally not a good idea because it adds an extra dependency on the order of modifications to the table. The `FIRST()` or `LAST()` (i.e. the context's `begin()` or `last()`) are much better and not any more expensive.

```
$size = $context->groupSize();
```

returns the number of rows in the group. It's your value of `COUNT(*)` in SQL terms, and if that's all you need, you don't need to iterate.

```
$context->send($opcode, $row);
```

constructs a result rowop and sends it to the aggregator's output label. Remember, the aggregator type as such knows nothing about this label, so the path through the context is the only path. Note also that it takes a row and not a rowop, because a label is needed to construct the rowop in the first place.

```
$rt = $context->resultType();
```

provides the result row type needed to construct the result row. There also are a couple of convenience methods that combine the row construction and sending, that can be used instead:

```
$context->makeHashSend ($opcode, $fieldName => $fieldValue, ...);
$context->makeArraySend($opcode, @fieldValues);
```

The final thing about the aggregator context: it works only inside the aggregator computation function. Once the function returns, all its methods start returning `undef`. So there is no point in trying to save it for later in a global variable or such, don't do that.

As you can see, `computeAverage()` has the same logic as before, only now it uses the aggregation context. And I've removed the debugging printout of the rows in the group.

The last unexplained piece is the opcode handling and that comparison to `OP_NOP`. Basically, the table calls the aggregator computation every time something changes in its index. It describes the reason for the call in the argument `$aggop` (“aggregation operation”). Depending on how clever an aggregator wants to be, it may do something useful on all of these occasions, or only on some of them. The simple aggregator that doesn't try any smart optimizations but just goes and iterates through the rows every time only needs to react in some of the cases. To make its life easier, Triceps pre-computes the opcode that should be used for the result and puts it into the argument `$opcode`. So to ignore the non-interesting calls, the simple aggregator computation can just return if it sees the opcode `OP_NOP`.

Why does it also check for the group size being 0? Again, Triceps provides flexibility in the aggregators. Among other things, it allows to implement the logic like Coral8, when on deletion of the last row in the group the aggregator would send a row with all non-key fields set to NULL (it can take the key fields from the argument `$rh`). So for this specific purpose the computation function gets called with all rows deleted from the group, and `$opcode` set to `OP_INSERT`. And, by the way, a true Coral8-styled aggregator would ignore all the calls where the `$opcode` is not `OP_INSERT`. But the normal aggregators need to avoid doing this kind of crap, so they have to ignore the calls where `$context->groupSize()==0`.

And here is an example of the output from that code (as usual, the input lines are in bold):

```
OP_INSERT,1,AAA,10,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="10"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="15"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="30"
OP_DELETE,5
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="30"
```

As you can see, it's exactly the same as from the manual aggregation example with the helper table, minus the debugging printout of the group contents. However here it's done without the helper table: instead the aggregation function is called before and after each update.

This presents a memory vs CPU compromise: a helper table uses more memory but requires less CPU for the aggregation computations (presumably, the insertion of the row into the table is less computationally intensive than the iteration through the original records).

The managed aggregators can be made to work with a helper table too: just chain a helper table to the aggregator's label, and in the aggregator computation add

```
return if ($opcode == &Triceps::OP_DELETE
  && $context->groupSize() != 1);
```

This would skip all the DELETEs except for the last one, before the group collapses.

There is also a way to optimize this logic right inside the aggregator: remember the last INSERT row sent, and on DELETE just resend the same row, as will be shown in Section 11.5: “Optimized DELETEs” (p. 147). This remembered last state can also be used for the other interesting optimizations that will be shown in Section 11.6: “Additive aggregation” (p. 149) .

Which approach is better, depends on the particular case. If you need to store the results of aggregation in a table for the future look-ups anyway, then that table is no extra overhead. That's what the Aleri system does internally: since each element in its model keeps a primary-indexed table (“materialized view”) of the result, that table is used whenever possible to generate the DELETEs without involving any logic. Or the extra optimization inside the aggregator can seriously improve the performance on the large groups. Sometimes you may want both.

Now let's look at the run with the same input that went wrong with the manual aggregation:

```
OP_INSERT,1,AAA,10,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="10"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="15"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_INSERT,5,BBB,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="20"
tWindow.aggrAvgPrice OP_INSERT symbol="BBB" id="5" price="30"
OP_INSERT,7,AAA,40,40
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="20"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="7" price="30"
```

Here it goes right. Triceps recognizes that the second INSERT with id=5 moves the row to another group. So it performs the aggregation logic for both groups. First for the group where the row gets removed, it updates the aggregator result with a DELETE and INSERT (note that id became 3, since it's now the last row left in that group). Then for the group where the row gets added, and since there was nothing in that group before, it generates only an INSERT.

11.4. Tricks with aggregation on a sliding window

Now it all works as it should, but there is still some room for improvement, related to the way the sliding window limits are handled.

Let's look again at the sample aggregation output with row deletion, copied here for convenience:

```
OP_INSERT,1,AAA,10,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="10"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="15"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="30"
OP_DELETE,5
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="30"
```

When the row with id=3 is deleted, the average price reverts to 30, which is the price of the trade with id=5, not the average of trades with id 1 and 5.

This is because the table is actually a sliding window, with the FIFO index having a limit of 2 rows

```
my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last2",
      Triceps::IndexType->newFifo(limit => 2)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", undef, \&computeAverage1)
      )
    )
  )
or confess "$!";
```

When the row with id=5 was inserted, it pushed out the row with id=1. Deleting the record with id=3 does not put that row with id=1 back. You can see the group contents in an even earlier printout with the manual aggregation, also copied here for convenience:

```
OP_INSERT,1,AAA,10,10
Contents:
  id="1" symbol="AAA" price="10" size="10"
lbAverage OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
Contents:
  id="1" symbol="AAA" price="10" size="10"
  id="3" symbol="AAA" price="20" size="20"
lbAverage OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
Contents:
  id="3" symbol="AAA" price="20" size="20"
  id="5" symbol="AAA" price="30" size="30"
lbAverage OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
Contents:
  id="5" symbol="AAA" price="30" size="30"
lbAverage OP_INSERT symbol="AAA" id="5" price="30"
OP_DELETE,5
Contents:
```

Like the toothpaste, once out of the tube, it's not easy to put back. But for this particular kind of toothpaste there is a trick: keep more rows in the group just in case but use only the last few for the actual aggregation. To allow an occasional deletion of a single row, we can keep 3 rows instead of 2.

So, change the table definition:

```
...
      Triceps::IndexType->newFifo(limit => 3)
...
```

and modify the aggregator function to use only the last 2 rows from the group, even if more are available:

```
sub computeAverage2 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);

  my $skip = $context->groupSize()-2;
  my $sum = 0;
  my $count = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull();
      $rhi = $context->next($rhi)) {
    if ($skip > 0) {
      $skip--;
      next;
    }
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->last()->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $context->send($opcode, $res);
}
```

The output from this version becomes:

```
OP_INSERT,1,AAA,10,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="10"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="15"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="20"
OP_DELETE,5
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="20"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
```

Now after `OP_DELETE,3` the average price becomes 20, the average of 10 and 30, because the row with id=1 comes into play again. Can you repeat that in the SQLy languages?

This version stores one extra row and thus can handle only one deletion (until the deleted row's spot gets pushed out of the window naturally, then it can handle another). It can not handle the arbitrary modifications properly. If you insert another row with id=3 for the same symbol “AAA”, the new version will be placed again at the end of the window. If it was the last row anyway, that is fine. But if it was not the last, as in this example, that would be an incorrect order that will produce incorrect results.

But just change the table type definition to aggregate on a sorted index instead of FIFO and it becomes able to handle the updates while keeping the rows in the order of their ids:

```
my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("orderById",
      Triceps::SimpleOrderedIndex->new(id => "ASC",)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", undef, \&computeAverage3)
      )
    )
    ->addSubIndex("last3",
      Triceps::IndexType->newFifo(limit => 3))
  )
or confess "$!";
```

The FIFO index is still there, in parallel, but it doesn't determine the order of rows for aggregation any more. Here is a sample of this version's work:

```
OP_INSERT,1,AAA,10,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="10"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,5,AAA,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="15"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="20"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="20"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_INSERT,7,AAA,40,40
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="7" price="35"
```

When the row with id=3 gets deleted, the average reverts to the rows 1 and 5. When the row 3 gets inserted back, the average works on rows 3 and 5 again. Then when the row 7 is inserted, the aggregation moves up to the rows 5 and 7.

The row expiration is still controlled by the FIFO index. So after the row 3 is inserted back, the order of rows in the FIFO becomes

```
1, 5, 3
```

Then when the row 7 is inserted, it advances to

```
5, 3, 7
```

At this point, until the row 3 gets naturally popped out of the FIFO, it's best not to have other deletions nor updates, or the group contents may become incorrect.

The FIFO and Ordered index types work in parallel on the same group, and the Ordered index always keeps the right order:

```
1, 3, 5
```

```
3, 5, 7
```

At long as the records with the two highest ids are in the group at all, the Ordered index will keep them in the right position at the end.

In this case we could even make a bit of optimization: turn the sorting order around, and have the Ordered index arrange the rows in the descending order. Then instead of skipping the rows until the last two, just take the first two rows of the reverse order. They'll be iterated in the opposite direction but for the averaging it doesn't matter. And instead of the last row take the first row of the opposite order. This is a simple modification and is left as an exercise for the reader.

Thinking further, the sensitivity to the ordering comes largely from the FIFO index. If the replacement policy could be done directly on the Ordered index, it would become easier. Would be a good thing to add in the future. Also, if you keep all the day's trades anyway, you might not need to have a replacement policy at all: just pick the last 2 records for the aggregation. There is currently no way to iterate back from the end (another thing to add in the future) but the same trick with the opposite order would work.

For a new subject, this table type indexes by id twice: once as a primary index, another time as a nested one. Are both of them really necessary or would just the nested one be good enough? That depends on your input data. If you get the DELETEs like `OP_DELETE,3` with all the other fields as NULL, then a separate primary index is definitely needed. But if the DELETEs come exactly as the same records that were inserted, only with a different opcode, like `OP_DELETE,3,AAA,20,20` then the primary index can be skipped because the nested sorted index will be able to find the rows correctly and handle them. The bottom line is, the fully correct DELETE records are good.

11.5. Optimized DELETEs

I've already mentioned that the DELETEs coming out of an aggregator do not have to be recalculated every time. Instead the rows can be remembered from the insert time, and simply re-sent with the new opcode. That allows to trade the CPU time for the extra memory. Of course, this works best when there are many rows per aggregation group, then more CPU time is saved on not iterating through them. How many is “many”? It depends on the particular cases. You'd have to measure. Anyway, here is how it's done:

```
sub computeAverage4 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);
  if ($opcode == &Triceps::OP_DELETE) {
    $context->send($opcode, $$state);
    return;
  }

  my $sum = 0;
  my $count = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull();
      $rhi = $context->next($rhi)) {
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->last()->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  ${$state} = $res;
  $context->send($opcode, $res);
}

sub initRememberLast #  (@args)
{
  my $refvar;
  return \$refvar;
}

my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last2",
      Triceps::IndexType->newFifo(limit => 2)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", \&initRememberLast, \&computeAverage4)
      )
    )
  )
or confess "$!";
```

The rest of the example stays the same, so it's not shown. Even in the part that is shown, very little has changed.

The aggregator type now has an initialization function. (This function is **not** of the same kind as for the sorted index!) This function gets called every time a new aggregation group gets created, before the first row is inserted into it. It initializes the aggregator group's Perl state by creating and returning the state value (the state is per aggregator type, so if there are two parallel index types, each with an aggregator, each aggregator will have its own group state).

The state is stored in the group as a single Perl variable. So it usually is a reference to a more complex object. In this case the value returned is a reference to a variable that would contain a Row reference. (Ironically, the simplest case looks a bit more confusing than if it were a reference to an array or hash). Returning a reference to a `my` variable is a way to create a reference to an anonymous value: each time `my` executes, it creates a new value. Which is then kept in a reference after the initialization function returns. The next time the function executes, `my` would create another new value.

The computation function has that state passed as an argument and now makes use of it. It has two small additions. Before sending a new result row, that row gets remembered in the state reference. And then before doing any computation the function checks, whether the required opcode is DELETE, and if so then simply resends the last result with the new opcode. Remember, the rows are not copied but reference-counted, so this is fairly cheap.

The extra level of referencing is used because simply assigning to `$state` would only change the local variable and not the value kept in the group.

However if you change the argument of the function directly, that would change the value kept in the group (similar to changing the loop variable in a *foreach* loop). So you can save a bit of overhead by eliminating the extra indirection. The modified version will be:

```
sub computeAverage5 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);
  if ($opcode == &Triceps::OP_DELETE) {
    $context->send($opcode, $state);
    return;
  }
  my $sum = 0;
  my $count = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull();
      $rhi = $context->next($rhi)) {
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->last()->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $_[5] = $res;
  $context->send($opcode, $res);
}

sub initRememberLast5 #  (@args)
{
  return undef;
}
```

Even though the initialization function returns `undef`, it still must be present. If it's not present, the state argument of the comparison function will contain a special hardcoded and unmodifiable `undef` constant, and nothing could be remembered.

And here is an example of its work:

```
OP_INSERT,1,AAA,10,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="10"
OP_INSERT,2,BBB,100,100
tWindow.aggrAvgPrice OP_INSERT symbol="BBB" id="2" price="100"
OP_INSERT,3,AAA,20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="10"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="15"
OP_INSERT,4,BBB,200,200
tWindow.aggrAvgPrice OP_DELETE symbol="BBB" id="2" price="100"
tWindow.aggrAvgPrice OP_INSERT symbol="BBB" id="4" price="150"
OP_INSERT,5,AAA,30,30
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="15"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="25"
OP_DELETE,3
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="25"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="5" price="30"
OP_DELETE,5
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="5" price="30"
```

Since the rows are grouped by the symbol, the symbols “AAA” and “BBB” will have separate aggregation states.

11.6. Additive aggregation

In some cases the aggregation values don't have to be calculated by going through all the rows from scratch every time. If you do a sum of a field, you can as well add the value of the field when a row is inserted and subtract when a row is deleted. Not surprisingly, this is called an “additive aggregation”.

The averaging can also be done as an additive aggregation: it amounts to a sum divided by a count. The sum can obviously be done additively. The count is potentially additive too, but even better, we have the shortcut of `$context->group-`

Size(). Well, at least for the same definition of count that has been used previously in the non-additive example. The SQL definition of count (and of average) includes only the non-NULL values, but in the next example we will go with the Perl approach where a NULL is taken to have the same meaning as 0. The proper SQL count could not use that shortcut but would still be additive.

Triceps provides a way to implement the additive aggregation too. It calls the aggregation computation function for each changed row, giving it an opportunity to react. The argument $aggop indicates, what has happened. Here is the same example from Section 11.3: “Introducing the proper aggregation” (p. 139) rewritten in an additive way:

```
# aggregation handler: recalculate the average additively
sub computeAverage7 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  my $rowchg;

  if ($aggop == &Triceps::AO_BEFORE_MOD) {
    $context->send($opcode, $state->{lastrow});
    return;
  } elsif ($aggop == &Triceps::AO_AFTER_DELETE) {
    $rowchg = -1;
  } elsif ($aggop == &Triceps::AO_AFTER_INSERT) {
    $rowchg = 1;
  } else { # AO_COLLAPSE, also has opcode OP_DELETE
    return
  }

  $state->{price_sum} += $rowchg * $rh->getRow()->get("price");

  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);

  my $rLast = $context->last()->getRow() or confess "$!";
  my $count = $context->groupSize();
  my $avg = $state->{price_sum}/$count;
  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $state->{lastrow} = $res;

  $context->send($opcode, $res);
}

sub initAverage7 #  (@args)
{
  return { lastrow => undef, price_sum => 0 };
}
```

The tricks of keeping an extra row from Section 11.4: “Tricks with aggregation on a sliding window” (p. 143)could not be used with the additive aggregation. An additive aggregation relies on Triceps to tell it, which rows are deleted and which inserted, so it can not do any extra skipping easily. The index for the aggregation has to be defined with the correct limits. If we want an average of the last 2 rows, we set the limit to 2:

```
my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last2",
      Triceps::IndexType->newFifo(limit => 2)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", \&initAverage7, \&computeAverage7)
      )
    )
  )
or confess "$!";
```

The aggregation state has grown: now it includes not only the last sent row but also the sum of the price, which is used for the aggregation, kept together in a hash. The last sent row doesn't really have to be kept, and I'll show another example without it, but for now let's look at how things are done when it is kept.

The argument `$aggop` describes, why the computation is being called. Note that Triceps doesn't know if the aggregation is additive or not. It does the calls the same in every case. Just in the previous examples we weren't interested in this information and didn't look at it. `$aggop` contains one of the constant values:

- `&Triceps::AO_BEFORE_MOD`: the group is about to be modified, need to send a DELETE of the old aggregated row. The argument `$opcode` will always be `OP_DELETE`.

- `&Triceps::AO_AFTER_DELETE`: the group has been modified by deleting a row from it. The argument `$rh` will refer to the row handle being deleted. The `$opcode` may be either `OP_NOP` or `OP_INSERT`. A single operation on a table may affect multiple rows: an insert may trigger the replacement policy in the indexes and cause one or more rows to be deleted. If there are multiple rows deleted or inserted in a group, the additive aggregator needs to know about all of them to keep its state correct but does not need (and even must not) send a new result until the last one of them has been processed. The call for the last modification will have the opcode of `OP_INSERT`. The preceding intermediate ones will have the opcode of `OP_NOP`. An important point, even though a row is being deleted from the group, the aggregator opcode is `OP_INSERT`, because it inserts the new aggregator state!

- `&Triceps::AO_AFTER_INSERT`: the group has been modified by inserting a row into it. Same as for `AO_AFTER_DELETE`, `$rh` will refer to the row handle being inserted, and `$opcode` will be `OP_NOP` or `OP_INSERT`.

- `&Triceps::AO_COLLAPSE`: called after the last row is deleted from the group, just before the whole group is collapsed and deleted. This allows the aggregator to destroy its state properly. For most of the aggregators there is nothing special to be done. The only case when you want to do something is if your state causes some circular references. Perl doesn't free the circular references until the whole interpreter exits, and so you'd have to break the circle to let them be freed immediately. The aggregator should not produce any results on this call. The `$opcode` will be `OP_NOP`.

The computation reacts accordingly: for the before-modification it re-sends the old result with the new opcode, for the collapse it does nothing, and for after-modification it calculates the sign, whether the value from `$rh` needs to be added or subtracted from the sum. I'm actually thinking, maybe this sign should be passed as a separate argument too, and then both the aggregation operation constants `AO_AFTER_*` can be merged into one. We'll see, maybe it will be changed in the future.

Then the addition/subtraction is done and the state updated.

After that, if the row does not need to be sent (opcode is `OP_NOP` or group size is 0), the function can as well return here without constructing the new row.

If the row needs to be produced, continue with the same logic as the non-additive aggregator, only without iteration through the group. The id field in the result is produced by essentially the SQL `LAST()` operator. `LAST()` and `FIRST()` are not additive, they refer to the values in the last or first row in the group's order, and simply can not be calculated from looking at which rows are being inserted and deleted without knowing their order in the group. But they are fast as they are, and do not require iteration. The same goes for the row count (as long as we don't care about excluding NULLs, violating the SQL semantics). And for averaging there is the last step to do after the additive part is done: divide the sum by the count.

All these non-additive steps are done in this last section, then the result row is constructed, remembered and sent.

Not all the aggregation operations can be expressed in an additive way. It may even vary by the data. For `MAX()`, the insertion of a row can be always done additively, just comparing the new value with the remembered maximum, and replacing it if the new value is greater. The deletion can also compare the deleted value with the remembered maximum. If the deleted value is less, then the maximum is unchanged. But if the deleted value is equal to the maximum, `MAX()` has to iterate through all the values and find the new maximum.

There is also an issue with the floating point precision in the additive aggregation. It's not such a big issue if the rows are only added and never deleted from the group, but can get much worse with the deletion. Let me show it with a sample run of the additive code:

```
OP_INSERT,1,AAA,1,10
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="1" price="1"
OP_INSERT,2,AAA,1e20,20
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="1" price="1"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="2" price="5e+19"
OP_INSERT,3,AAA,2,10
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="2" price="5e+19"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="3" price="5e+19"
OP_INSERT,4,AAA,3,10
tWindow.aggrAvgPrice OP_DELETE symbol="AAA" id="3" price="5e+19"
tWindow.aggrAvgPrice OP_INSERT symbol="AAA" id="4" price="1.5"
```

Why is the last result 1.5 while it had to be (2+3)/2 = 2.5? Because adding together 1e20 and 2 had pushed the 2 beyond the precision of floating-point number. 1e20+2 = 1e20. So when the row with 1e20 was deleted from the group and subtracted form the sum, that left 0. Which got then averaged with 3, producing 1.5.

Of course, with the real stock prices there won't be that much variation. But the subtler errors will still accumulate over time, and you have to expect them and plan accordingly.

Switching to a different subject, the additive aggregation contains enough information in its state to generate the result rows quickly without an iteration. This means that keeping the saved result row for DELETEs doesn't give a whole lot of advantage and adds at least a little memory overhead. We can change the code and avoid keeping it:

```
sub computeAverage8 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  my $rowchg;

  if ($aggop == &Triceps::AO_COLLAPSE) {
    return
  } elsif ($aggop == &Triceps::AO_AFTER_DELETE) {
    $state->{price_sum} -= $rh->getRow()->get("price");
  } elsif ($aggop == &Triceps::AO_AFTER_INSERT) {
    $state->{price_sum} += $rh->getRow()->get("price");
  }
  # on AO_BEFORE_MOD do nothing

  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);

  my $rLast = $context->last()->getRow() or confess "$!";
  my $count = $context->groupSize();

  $context->makeHashSend($opcode,
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $state->{price_sum}/$count,
  );
}

sub initAverage8 #  (@args)
```

```
{
  return { price_sum => 0 };
}
```

On AO_BEFORE_MOD it doesn't do any change to the additive state but then produces the result row from that state as usual, using the supplied $opcode value of OP_DELETE. The other change in this example is that the sum gets directly added or subtracted in AO_AFTER_* instead of computing the sign first. It's all pretty much self-explanatory.

11.7. Computation function arguments

Let's look up close at what calls are done to the aggregation computation function. Just make a “computation” that prints the call arguments:

```
sub computeAverage9 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;

  print(&Triceps::aggOpString($aggop), " ", &Triceps::opcodeString($opcode), " ",
 $context->groupSize(), " ", (!$rh->isNull()? $rh->getRow()->printP(): "NULL"), "\n");
}
```

It prints the aggregation operation, the result opcode, row count in the group, and the argument row (or “NULL”). The aggregation is done as before, on the same FIFO index with the size limit of 2.

To show the order of aggregator calls relative to the table label calls, I've added the labels that print the updates form the table:

```
my $lbPre = makePrintLabel("lbPre", $tWindow->getPreLabel());
my $lbOut = makePrintLabel("lbOut", $tWindow->getOutputLabel());
```

To make keeping track of the printout easier, I broke up the sequence into multiple fragments, with a description after each fragment:

```
OP_INSERT,1,AAA,10,10
tWindow.pre OP_INSERT id="1" symbol="AAA" price="10" size="10"
tWindow.out OP_INSERT id="1" symbol="AAA" price="10" size="10"
AO_AFTER_INSERT OP_INSERT 1 id="1" symbol="AAA" price="10" size="10"
OP_INSERT,2,BBB,100,100
tWindow.pre OP_INSERT id="2" symbol="BBB" price="100" size="100"
tWindow.out OP_INSERT id="2" symbol="BBB" price="100" size="100"
AO_AFTER_INSERT OP_INSERT 1 id="2" symbol="BBB" price="100" size="100"
```

The INSERT of the first row in each group causes only one call. There is no previous value to delete, only a new one to insert. The call happens after the row has been inserted into the group.

```
OP_INSERT,3,AAA,20,20
AO_BEFORE_MOD OP_DELETE 1 NULL
tWindow.pre OP_INSERT id="3" symbol="AAA" price="20" size="20"
tWindow.out OP_INSERT id="3" symbol="AAA" price="20" size="20"
AO_AFTER_INSERT OP_INSERT 2 id="3" symbol="AAA" price="20" size="20"
```

Adding the second record in a group means that the aggregation result for this group is modified. So first the aggregator is called to delete the old result, then the new row gets inserted, and the aggregator is called the second time to produce its new result.

```
OP_INSERT,5,AAA,30,30
AO_BEFORE_MOD OP_DELETE 2 NULL
tWindow.pre OP_DELETE id="1" symbol="AAA" price="10" size="10"
tWindow.out OP_DELETE id="1" symbol="AAA" price="10" size="10"
tWindow.pre OP_INSERT id="5" symbol="AAA" price="30" size="30"
tWindow.out OP_INSERT id="5" symbol="AAA" price="30" size="30"
AO_AFTER_DELETE OP_NOP 2 id="1" symbol="AAA" price="10" size="10"
AO_AFTER_INSERT OP_INSERT 2 id="5" symbol="AAA" price="30" size="30"
```

The insertion of the third row in a group triggers the replacement policy in the FIFO index. The replacement policy causes the row with id=1 to be deleted before the row with id=5 is inserted. For the aggregator result it's still a single delete-insert pair: First, before modification, the old aggregation result is deleted. Then the contents of the group gets modified with both the delete and insert. And then the aggregator gets told, what has been modified. The deletion of the row with id=1 is not the last step, so that call gets the opcode of `OP_NOP`. Note that the group size with it is 2, not 1. That's because the aggregator gets notified only after all the modifications are already done. So the additive part of the computation must never read the group size or do any kind of iteration through the group, because that would often cause an incorrect result: it has no way to tell, what other modifications have been already done to the group. The last `AO_AFTER_INSERT` gets the opcode of `OP_INSERT` which tells the computation to send the new result of the aggregation. When the opcode is `OP_INSERT`, reading the group size and the other group information becomes safe, because by this time all the modifications are guaranteed to be done, and the additive notifications have caught up with all the changes.

```
OP_INSERT,3,BBB,20,20
AO_BEFORE_MOD OP_DELETE 2 NULL
AO_BEFORE_MOD OP_DELETE 1 NULL
tWindow.pre OP_DELETE id="3" symbol="AAA" price="20" size="20"
tWindow.out OP_DELETE id="3" symbol="AAA" price="20" size="20"
tWindow.pre OP_INSERT id="3" symbol="BBB" price="20" size="20"
tWindow.out OP_INSERT id="3" symbol="BBB" price="20" size="20"
AO_AFTER_DELETE OP_INSERT 1 id="3" symbol="AAA" price="20" size="20"
AO_AFTER_INSERT OP_INSERT 2 id="3" symbol="BBB" price="20" size="20"
```

This insert is of a “dirty” kind, the one that replaces the row using the replacement policy of the hashed primary index, without deleting its old state first. It also moves the row from one aggregation group to another. So the table logic calls `AO_BEFORE_MOD` for each of the modified groups, then modifies the contents of the groups, then tells both groups about the modifications. In this case both calls with `AO_AFTER_*` have the opcode of `OP_INSERT` because each of them is the last and only change to a separate aggregation group.

```
OP_DELETE,5
AO_BEFORE_MOD OP_DELETE 1 NULL
tWindow.pre OP_DELETE id="5" symbol="AAA" price="30" size="30"
tWindow.out OP_DELETE id="5" symbol="AAA" price="30" size="30"
AO_AFTER_DELETE OP_INSERT 0 id="5" symbol="AAA" price="30" size="30"
AO_COLLAPSE OP_NOP 0 NULL
```

This operation removes the last row in a group. It starts as usual with deleting the old state. The next `AO_AFTER_DELETE` with `OP_INSERT` is intended for the Coral8-style aggregators that produce only the rows with the INSERT opcodes, never DELETEs, to let them insert the NULL (or zero) values in all the non-key fields. For the normal aggregators the work is all done after `OP_DELETE`. That's why all the shown examples were checking for `$context->groupSize() == 0` and returning if so. The group size will be zero in absolutely no other case than after the deletion of the last row. Finally `AO_COLLAPSE` allows to clean up the aggregator's group state if it needs any cleaning. It has the opcode `OP_NOP` because no rows need to be sent.

To recap, the high-level order of the table operation processing is:

1. Execute the replacement policies on all the indexes to find all the rows that need to be deleted first.

2. If any of the index policies forbid the modification, return 0.

3. Call all the aggregators with `AO_BEFORE_MOD` on all the affected rows.

4. Send these aggregator results.

5. For each affected row:

   a. Call the "pre" label (if it has any labels chained to it).

   b. Modify the row in the table.

   c. Call the "out" label.

6. Call all the aggregators with `AO_AFTER_*`, on all the affected rows.

7. Send these aggregator results.

11.8. Using multiple indexes

I've mentioned before that the floating numbers are tricky to handle. Even without additive aggregation the result depends on the rounding. Which in turn depends on the order in which the operations are done. Let's look at a version of the aggregation code that highlights this issue.

```
sub computeAverage10 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode != &Triceps::OP_INSERT);

  my $sum = 0;
  my $count = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull();
      $rhi = $context->next($rhi)) {
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->last()->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $context->send($opcode, $res);
}

my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last4",
      Triceps::IndexType->newFifo(limit => 4)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", undef, \&computeAverage10)
      )
    )
  )
or confess "$!";
$ttWindow->initialize() or confess "$!";
my $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

# label to print the result of aggregation
```

```
my $lbAverage = $uTrades->makeLabel($rtAvgPrice, "lbAverage",
  undef, sub { # (label, rowop)
    printf("%.17g\n", $_[1]->getRow()->get("price"));
  }) or confess "$!";
$tWindow->getAggregatorLabel("aggrAvgPrice")->chain($lbAverage)
  or confess "$!";
```

The differences from the previously shown basic aggregation are:

- the FIFO limit has been increased to 4;
- the only result value printed by the `$lbAverage` handler is the price, and it's printed with a higher precision to make the difference visible;
- the aggregator computation only does the inserts, to reduce the clutter in the results and highlight the issue.

And here is an example of how the order of computation matters:

```
OP_INSERT,1,AAA,1,10
1
OP_INSERT,2,AAA,1,10
1
OP_INSERT,3,AAA,1,10
1
OP_INSERT,4,AAA,1e16,10
2500000000000001
OP_INSERT,5,BBB,1e16,10
10000000000000000
OP_INSERT,6,BBB,1,10
5000000000000000
OP_INSERT,7,BBB,1,10
3333333333333333.5
OP_INSERT,8,BBB,1,10
2500000000000000
```

Of course, the real prices won't vary so wildly. But the other values could. This example is specially stacked to demonstrate the point. The final results for “AAA” and “BBB” should be the same but aren't. Why? The precision of the 64-bit floating-point numbers is such that adding 1 to 1e16 makes this 1 fall beyond the precision, and the result is still 1e16. On the other hand, adding 3 to 1e16 makes at least a part of it stick. 1 still falls off but the other 2 of 3 sticks on. Next look at the data sets: if you add 1e16+1+1+1, that's adding 1e16+1 repeated three times, and the result is still the same unchanged 1e16. But if you add 1+1+1+1e16, that's adding 3+1e16, and now the result is different and more correct. When the averages get computed from these different values by dividing the sums by 4, the results are also different.

Overall the rule of thumb for adding the floating point numbers is this: add them up in the order from the smallest to the largest. (What if the numbers can be negative too? I don't know, that goes beyond my knowledge of floating point calculations. My guess is that you still arrange them in the ascending order, only by the absolute value.) So let's do it in the aggregator.

```
our $idxByPrice;

# aggregation handler: sum in proper order
sub computeAverage11 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  our $idxByPrice;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode != &Triceps::OP_INSERT);

  my $sum = 0;
  my $count = 0;
  my $end = $context->endIdx($idxByPrice);
  for (my $rhi = $context->beginIdx($idxByPrice); !$rhi->same($end);
      $rhi = $rhi->nextIdx($idxByPrice)) {
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->last()->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $context->send($opcode, $res);
}

my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last4",
      Triceps::IndexType->newFifo(limit => 4)
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", undef, \&computeAverage11)
      )
    )
    ->addSubIndex("byPrice",
      Triceps::SimpleOrderedIndex->new(price => "ASC",)
      ->addSubIndex("multi", Triceps::IndexType->newFifo())
    )
  )
or confess "$!";
$ttWindow->initialize() or confess "$!";
my $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

$idxByPrice = $ttWindow->findIndexPath("bySymbol", "byPrice");
```

Here another index type is added, ordered by price. It has to be non-leaf, with a FIFO index type nested in it, to allow for multiple rows having the same price in them. That would work out more efficiently if the ordered index could have a multimap mode, but that is not supported yet.

When the compute function does its iteration, it now goes by that index. The aggregator can't be simply moved to that new index type, because it still needs to get the last trade id in the order in which the rows are inserted into the group. Instead it has to work with two index types: the one on which the aggregator is defined, and the additional one. The calls for iteration on an additional index are different. `$context->beginIdx()` is similar to `$context->begin()` but the end condition and the next step are done differently. When `$rhi->nextIdx()` reaches the end of the group, it returns not a NULL row handle but a handle value that has to be found in advance with `$context->endIdx()`. Perhaps the consistency in this department can be improved in the future.

And finally, the reference to that additional index type has to make it somehow into the compute function. It can't be given as an argument because it's not known yet at the time when the aggregator is constructed (and no, reordering the index types won't help because the index types are copied when connected to their parents, and we need the exact index type that ends up in the assembled table type). So a global variable `$idxByPrice` is used. The index type reference is found and placed there, and later when the compute function runs, it takes the reference from the global variable.

The printout from this version on the same input is:

```
OP_INSERT,1,AAA,1,10
1
OP_INSERT,2,AAA,1,10
1
OP_INSERT,3,AAA,1,10
1
OP_INSERT,4,AAA,1e16,10
2500000000000001
OP_INSERT,5,BBB,1e16,10
10000000000000000
OP_INSERT,6,BBB,1,10
5000000000000000
OP_INSERT,7,BBB,1,10
3333333333333334
OP_INSERT,8,BBB,1,10
2500000000000001
```

Now no matter what the order of the row arrival, the prices get added up in the same order from the smallest to the largest and produce the same correct (inasmuch the floating point precision allows) result.

Which index type is used to put the aggregator on, doesn't matter a whole lot. The computation can be turned around, with the ordered index used as the main one, and the last value from the FIFO index obtained with `$context->lastIdx()`:

```
our $idxByOrder;

# aggregation handler: sum in proper order
sub computeAverage12 # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  our $idxByOrder;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode != &Triceps::OP_INSERT);

  my $sum = 0;
  my $count = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull();
      $rhi = $context->next($rhi)) {
    $count++;
    $sum += $rhi->getRow()->get("price");
  }
  my $rLast = $context->lastIdx($idxByOrder)->getRow() or confess "$!";
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  ) or confess "$!";
  $context->send($opcode, $res);
}

my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last4",
      Triceps::IndexType->newFifo(limit => 4)
    )
    ->addSubIndex("byPrice",
      Triceps::SimpleOrderedIndex->new(price => "ASC",)
      ->addSubIndex("multi", Triceps::IndexType->newFifo())
      ->setAggregator(Triceps::AggregatorType->new(
        $rtAvgPrice, "aggrAvgPrice", undef, \&computeAverage12)
      )
    )
  )
or confess "$!";
$ttWindow->initialize() or confess "$!";
my $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

$idxByOrder = $ttWindow->findIndexPath("bySymbol", "last4");
```

The last important note: when aggregating with multiple indexes, always use the sibling index types forming the same group or their nested sub-indexes (since the actual order is defined by the first leaf sub-index anyway). But don't use the random unrelated index types. If you do, the context would return some unexpected values for those, and you may end up with endless loops.

11.9. SimpleAggregator

Even though the writing the aggregation computation functions manually gives the flexibility, it's too much work for the simple cases. The SimpleAggregator template takes care of most of that work and allows you to specify the aggregation in a way similar to SQL. It has been already shown on the VWAP example, ans here is the trade aggregation example from Section 11.3: “Introducing the proper aggregation” (p. 139) rewritten with SimpleAggregator:

```
my $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("byId",
    Triceps::IndexType->newHashed(key => [ "id" ])
  )
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
    ->addSubIndex("last2",
      Triceps::IndexType->newFifo(limit => 2)
    )
  )
or confess "$!";

# the aggregation result
my $rtAvgPrice;
my $compText; # for debugging

Triceps::SimpleAggregator::make(
  tabType => $ttWindow,
  name => "aggrAvgPrice",
  idxPath => [ "bySymbol", "last2" ],
  result => [
    symbol => "string", "last", sub {$_[0]->get("symbol");},
    id => "int32", "last", sub {$_[0]->get("id");},
    price => "float64", "avg", sub {$_[0]->get("price");},
  ],
  saveRowTypeTo => \$rtAvgPrice,
  saveComputeTo => \$compText,
);

$ttWindow->initialize() or confess "$!";
my $tWindow = $uTrades->makeTable($ttWindow,
  &Triceps::EM_CALL, "tWindow") or confess "$!";

# label to print the result of aggregation
my $lbAverage = makePrintLabel("lbAverage",
  $tWindow->getAggregatorLabel("aggrAvgPrice"));
```

The main loop and the printing is the same as before. The result produced is also exactly the same as before.

But the aggregator is created with `Triceps::SimpleAggregator::make()`. Its arguments are in the option format: the option name-value pairs, in any order.

```
$tabType = Triceps::SimpleAggregator::make($optName => $optValue, ...);
```

It returns back the table type that it received as an option on success or confesses on errors. But most of the time there is not a whole lot of use to that return value, and it gets simply ignored. Most of the "options" are actually mandatory. The aggregator type is connected to the table type with the options:

tabType
: Table type to put the aggregator on. It must be un-initialized yet.

idxPath
: A reference to an array of index names, forming the path to the index where the aggregator type will be set.

name
: The aggregator type name.

The result row type and computation is defined with the option "result": each group of four values in that array defines one result field:

- The field name.
- The field type.
- The aggregation function name used to compute the field. There is no way to combine multiple aggregation functions or even an aggregation function and any arithmetics in a field computation. The workaround is to compute each function in a separate field, and then send the result rows to a computational label that would arithmetically combine these fields into one.
- A closure that extracts the aggregation function argument from the row (well, it can be any function reference, doesn't have to be an anonymous closure). That closure gets the row as the argument `$_[0]` and returns the extracted value to run the aggregation on.

The field name is by convention separated from its definition fields by `=>`. Remember, it's just a convention, for Perl a `=>` is just as good as a comma.

`SimpleAggregator::make()` automatically generates the result row type and aggregation function, creates an aggregator type from them, and sets it on the index type. The information about the aggregation result can be found by traversing through the index type tree, or by constructing a table and getting the row type from the aggregator result label. However it's often easier to save it during construction, and the option (this time an optional one!) "saveRowTypeTo" allows to do this. Give it a reference to a variable, and the row type will be placed into that variable.

Most of the time the things would just work. However if they don't and something dies in the aggregator, you will need the source code of the compute function to make sense of these errors. The option `saveComputeTo` gives a variable to save that source code for future perusal and other entertainment. Here is the compute function that gets produced by the example above:

```
sub {
  use strict;
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  return if ($context->groupSize()==0 || $opcode == &Triceps::OP_NOP);
  my $v2_count = 0;
  my $v2_sum = 0;
  my $npos = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull(); $rhi = $context->next($rhi)) {
    my $row = $rhi->getRow();
    # field price=avg
    my $a2 = $args[2]($row);
    { if (defined $a2) { $v2_sum += $a2; $v2_count++; }; }
    $npos++;
  }
  my $rowLast = $context->last()->getRow();
  my $l0 = $args[0]($rowLast);
  my $l1 = $args[1]($rowLast);
  $context->makeArraySend($opcode,
    ($l0), # symbol
    ($l1), # id
    (($v2_count == 0? undef : $v2_sum / $v2_count)), # price
  );
}
```

At the moment the compute function is quite straightforward and just does the aggregation from scratch every time. It doesn't support the additive aggregation nor the DELETE optimization. It's only smart enough to skip the iteration if all the result consists of only aggregation functions `first`, `last` and `count_star`. It receives the closures for the argument extraction as arguments in `@args`, SimpleAggregator arranges these arguments when it creates the aggregator.

The aggregation functions available at the moment are:

`first`
: Value from the first row in the group.

`last`
: Value from the last row in the group.

`count_star`
: Number of rows in the group, like SQL `COUNT(*)`. Since there is no argument for this function, use `undef` instead of the argument closure.

`sum`
: Sum of the values.

`max`
: The maximal value.

`min`
: The minimal value.

`avg`
: The average of all the non-NULL values.

`avg_perl`
: The average of all values, with NULL values treated in Perl fashion as zeroes. So, technically when the example above used `avg`, it works the same as the previous versions only for the non-NULL fields. To be really the same, it should have used `avg_perl`.

`nth_simple`
: The Nth value from the start of the group. This is a tricky function because it needs two arguments: the value of N and the field selector. Multiple direct arguments will be supported in the future but right now it works through a workaround: the argument closure must return not just the extracted field but a reference to array with two values, the N and the field. For example, `sub { [1, $_[0]->get("id")];}`. The N is counted starting from 0, so the value of 1 will return the second record. This function works in a fairly simple-minded and inefficient way at the moment.

As usual in Triceps and Perl, the case of the aggregation function name matters. The names have to be used in lowercase as shown. There will be more functions to come, and you can even already add your own, as has been shown in Section 11.1: “The ubiquitous VWAP” (p. 133) .

The user-defined aggregation functions are defined with the option “functions”. Let's take another look at the code from the VWAP example:

```
# VWAP function definition
my $myAggFunctions = {
  myvwap => {
    vars => { sum => 0, count => 0, size => 0, price => 0 },
    step => '($%size, $%price) = @$%argiter; '
      . 'if (defined $%size && defined $%price) '
        . '{$%count += $%size; $%sum += $%size * $%price;}',
    result => '($%count == 0? undef : $%sum / $%count)',
  },
};

...

Triceps::SimpleAggregator::make(
  functions => $myAggFunctions,
);
```

The definition of the functions is a reference to a hash, keyed by the function name. Each function definition in order is a hash of options, keyed by the option name. When the SimpleAggregator builds the common computation function, it assembles the code by tying together the code fragments from these options: Whenever the group changes, the aggregator will reset the function state variables to the default values and iterate through the new contents of the group. It will perform the step computation for each row and collect the data in the intermediate variables. After the iteration it will perform the result computation of all the functions and produce the final value.

The expected format of the values of these options varies with the option. The option “result” is mandatory, the rest can be skipped if not needed. The supported options are:

`argcount`

Integer. Defines the number of arguments of the function, which may currently be 0 or 1, with 1 being the default. If this option is 0, SimpleAggregator will check that the argument closure is `undef`. If the aggregation function needs more arguments than one, they have to be packed into an array or hash, and then its reference used as a single argument. The standard function `nth_simple` and the VWAP function provide the examples of how to do this.

`vars`

Reference to a hash. Defines the variables used to keep the context of this function during the iteration (the hash keys are the variable names) and their initial values (specified as the values in the hash).

`step`

String. The code fragment to compute a single step of iteration. It can refer to the variables defined in `vars` and to a few of the pre-defined values using the syntax `$%name` (which has been chosen because it's illegal in the normal Perl variable syntax). When SimpleAggregator generates the code, it creates the actual scope variables for everything defined in `vars`, then substitutes them for the `$%` syntax in the string and inserts the result into its group iteration code.

If this option is not defined, SimpleAggregator assumes that this function doesn't need it. If no functions in the aggregation define the `step`, the iteration does not get included into the generated code altogether.

The defined special values are:

- `$%argiter` - The function's argument extracted from the current row.
- `$%niter` - The number of the current row in the group, starting from 0.
- `$%groupsize` - The size of the group (`$context->groupSize()`).

`result`

String. The code fragment to compute the result of the function. This option is mandatory. Works in the same way as `step`, only gets executed once per call of the computation function, and the defined special values are different:

- `$%argfirst` - The function's argument extracted from the first row.
- `$%arglast` - The function's argument extracted from the last row.
- `$%groupsize` - The size of the group (`$context->groupSize()`).

I can think of many ways the SimpleAggregator can be improved, but for now they have been pushed into the future to keep it simple.

11.10. The guts of SimpleAggregator

The implementation of the SimpleAggregator has turned out to be surprisingly small. Not quite tiny but still small. I've liked it so much that I've even saved the original small version in the file `xSimpleAggregator.t`. As more features will be added, the “official” version of the SimpleAggregator will grow (and already did) but that example file will stay small and simple.

It's a nice example of yet another kind of template that I want to present. I'm going to go through it, interlacing the code with the commentary.

```
package MySimpleAggregator;
use Carp;

use strict;

our $FUNCTIONS = {
  first => {
    result => '$%argfirst',
  },
  last => {
    result => '$%arglast',
  },
  count_star => {
    argcount => 0,
    result => '$%groupsize',
  },
  count => {
    vars => { count => 0 },
    step => '$%count++ if (defined $%argiter);',
    result => '$%count',
  },
  sum => {
    vars => { sum => 0 },
    step => '$%sum += $%argiter;',
    result => '$%sum',
  },
  max => {
    vars => { max => 'undef' },
    step => '$%max = $%argiter if (!defined $%max || $%argiter > $%max);',
    result => '$%max',
  },
  min => {
    vars => { min => 'undef' },
    step => '$%min = $%argiter if (!defined $%min || $%argiter < $%min);',
    result => '$%min',
  },
  avg => {
    vars => { sum => 0, count => 0 },
    step => 'if (defined $%argiter) { $%sum += $%argiter; $%count++; }',
    result => '($%count == 0? undef : $%sum / $%count)',
  },
  avg_perl => { # Perl-like treat the NULLs as 0s
    vars => { sum => 0 },
    step => '$%sum += $%argiter;',
    result => '$%sum / $%groupsize',
  },
  nth_simple => { # inefficient, need proper multi-args for better efficiency
    vars => { n => 'undef', tmp => 'undef', val => 'undef' },
    step => '($%n, $%tmp) = @$%argiter; if ($%n == $%niter) { $%val = $%tmp; }',
    result => '$%val',
  },
};
```

The package name of this saved simple version is MySimpleAggregator, to avoid confusion with the “official” SimpleAggregator class. First goes the definition of the aggregation functions. They are defined in exactly the same way as the vwap function has been shown before. They are fairly straightforward. You can use them as the starting point for adding your own.

```
sub make # (optName => optValue, ...)
{
  my $opts = {}; # the parsed options
  my $myname = "MySimpleAggregator::make";

  &Triceps::Opt::parse("MySimpleAggregator", $opts, {
      tabType => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "Triceps::TableType") } ],
      name => [ undef, \&Triceps::Opt::ck_mandatory ],
      idxPath => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "ARRAY", "") } ],
      result => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "ARRAY") } ],
      saveRowTypeTo => [ undef, sub { &Triceps::Opt::ck_refscalar(@_) } ],
      saveInitTo => [ undef, sub { &Triceps::Opt::ck_refscalar(@_) } ],
      saveComputeTo => [ undef, sub { &Triceps::Opt::ck_refscalar(@_) } ],
    }, @_);
```

The options get parsed. Since it's not a proper object constructor but a factory, it uses the hash `$opts` instead of `$self` to save the processed copy of the options. This early version doesn't have an option for the user-supplied aggregation function definitions.

```
  # reset the saved source code
  ${$opts->{saveInitTo}} = undef if (defined($opts->{saveInitTo}));
  ${$opts->{saveComputeTo}} = undef if (defined($opts->{saveComputeTo}));
  ${$opts->{saveRowTypeTo}} = undef if (defined($opts->{saveRowTypeTo}));
```

The generated source code will not be placed into the `save*` references until the table type gets initialized, so for the meantime they get filled with `undef`s.

```
  # find the index type, on which to build the aggregator
  my $idx = $opts->{tabType}->findIndexPath(@{$opts->{idxPath}});
  confess "$myname: the index type is already initialized, can not add an aggregator on
 it"
    if ($idx->isInitialized());
```

Since the SimpleAggregator uses an existing table with existing index, it doesn't require the aggregation key: it just takes an index that forms the group, and whatever key that leads to this index becomes the aggregation key.

```
  # check the result definition and build the result row type and code snippets for the
 computation
  my $rtRes;
  my $needIter = 0; # flag: some of the functions require iteration
  my $needfirst = 0; # the result needs the first row of the group
  my $needlast = 0; # the result needs the last row of the group
  my $codeInit = ''; # code for function initialization
  my $codeStep = ''; # code for iteration
  my $codeResult = ''; # code to compute the intermediate values for the result
  my $codeBuild = ''; # code to build the result row
  my @compArgs; # the field functions are passed as args to the computation
  {
    my $grpstep = 4; # definition grouped by 4 items per result field
    my @resopt = @{$opts->{result}};
    my @rtdefRes; # field definition for the result
    my $id = 0; # numeric id of the field

    while ($#resopt >= 0) {
      confess "$myname: the values in the result definition must go in groups of 4"
        unless ($#resopt >= 3);
      my $fld = shift @resopt;
      my $type = shift @resopt;
      my $func = shift @resopt;
      my $funcarg = shift @resopt;

      confess("$myname: the result field name must be a string, got a " . ref($fld) . " ")
        unless (ref($fld) eq '');
      confess("$myname: the result field type must be a string, got a " . ref($type) . "
 for field '$fld'")
        unless (ref($type) eq '');
      confess("$myname: the result field function must be a string, got a " . ref($func) .
 " for field '$fld'")
        unless (ref($func) eq '');
```

This starts the loop that goes over the result fields and builds the code to create them. The code will be built in multiple snippets that will eventually be combined to produce the compute function. Since the arguments go in groups of 4, it becomes fairly easy to miss one element somewhere, and then everything gets real confusing. So the code attempts to check the types of the arguments, in hopes of catching these off-by-ones as early as possible. The variable `$id` will be used to produce the unique prefixes for the function's variables.

```
     my $funcDef = $FUNCTIONS->{$func}
       or confess("$myname: function '" . $func . "' is unknown");

     my $argCount = $funcDef->{argcount};
     $argCount = 1 # 1 is the default value
       unless defined($argCount);
     confess("$myname: in field '$fld' function '$func' requires an argument computation
that must be a Perl sub reference")
       unless ($argCount == 0 || ref $funcarg eq 'CODE');
     confess("$myname: in field '$fld' function '$func' requires no argument, use undef
as a placeholder")
       unless ($argCount != 0 || !defined $funcarg);

     push(@rtdefRes, $fld, $type);

     push(@compArgs, $funcarg)
       if (defined $funcarg);
```

The function definition for a field gets pulled out by name, and the arguments of the field are checked for correctness. The types of the fields get collected for the row definition, and the aggregation argument computation closures (or, technically, functions) get also collected, to pass later as the arguments of the compute function.

```
     # add to the code snippets
```

```
      ### initialization
      my $vars = $funcDef->{vars};
      if (defined $vars) {
        foreach my $v (keys %$vars) {
          # the variable names are given a unique prefix;
          # the initialization values are constants, no substitutions
          $codeInit .= "  my \$v${id}_${v} = " . $vars->{$v} . ";\n";
        }
      } else {
        $vars = { }; # a dummy
      }
```

The initialization fragment gets processed if defined. The unique names for variables are generated from the `$id` and the variable name in the definition, so that there would be no interference between the result fields. And the initialization snippets are collected in `$codeInit`. The initialization values are not enquoted because they are expected to be strings suitable for such use. That's why the undefined values in the function defnitions are not `undef` but `'undef'`. If you'd want to initialize a variable as a string `"x"`, you'd use it as `'"x"'`. For the numbers it doesn't really matter, the numbers just get converted to strings as needed, so the zeroes are simply `0`s without quoting.

Another possibility would be to have the actual values as-is in the hash and then either put these values into the argument array passed to the computation function or use the closure trick from `Triceps::Fields::makeTranslation()` described in Section 10.7: “Result projection in the templates” (p. 128) .

```
      ### iteration
      my $step = $funcDef->{step};
      if (defined $step) {
        $needIter = 1;
        $codeStep .= "    # field $fld=$func\n";
        if (defined $funcarg) {
          # compute the function argument from the current row
          $codeStep .= "    my \$a${id} = \$args[" . $#compArgs ."](\$row);\n";
        }
        # substitute the variables in $step
        $step =~ s/\$\%(\w+)/&replaceStep($1, $func, $vars, $id, $argCount)/ge;
        $codeStep .= "    { $step; }\n";
      }
```

Then the iteration fragment gets processed. The logic remembers in `$needIter` if any of the functions involved needs iteration. Before the iteration snippet gets collected, it has the `$%` names substitutted, and placed into a block, just in case if it wants to define some local variables. An extra “;” is added just in case, it doesn't hurt and helps if it was forgotten in the function definition.

```
      ### result building
      my $result = $funcDef->{result};
      confess "MySimpleAggregator: internal error in definition of aggregation function
 '$func', missing result computation"
        unless (defined $result);
      # substitute the variables in $result
      if ($result =~ /\$\%argfirst/) {
        $needfirst = 1;
        $codeResult .= "  my \$f${id} = \$args[" . $#compArgs ."](\$rowFirst);\n";
      }
      if ($result =~ /\$\%arglast/) {
        $needlast = 1;
        $codeResult .= "  my \$l${id} = \$args[" . $#compArgs ."](\$rowLast);\n";
      }
      $result =~ s/\$\%(\w+)/&replaceResult($1, $func, $vars, $id, $argCount)/ge;
      $codeBuild .= "    ($result), # $fld\n";

      $id++;
    }
    $rtRes = Triceps::RowType->new(@rtdefRes)
      or confess "$myname: invalid result row type definition: $!";
  }
  ${$opts->{saveRowTypeTo}} = $rtRes if (defined($opts->{saveRowTypeTo}));
```

In the same way the result computation is created, and remembers if any function wanted the fields from the first or last row. And eventually after all the functions have been processed, the result row type is created. If it was asked to save, it gets saved.

```
  # build the computation function
  my $compText = "sub {\n";
  $compText .= "  use strict;\n";
  $compText .= "  my (\$table, \$context, \$aggop, \$opcode, \$rh, \$state, \@args) = \@_;
\n";
  $compText .= "  return if (\$context->groupSize()==0 || \$opcode == &Triceps::OP_NOP);
\n";
  $compText .= $codeInit;
  if ($needIter) {
    $compText .= "  my \$npos = 0;\n";
    $compText .= "  for (my \$rhi = \$context->begin(); !\$rhi->isNull(); \$rhi = \
$context->next(\$rhi)) {\n";
    $compText .= "    my \$row = \$rhi->getRow();\n";
    $compText .= $codeStep;
    $compText .= "    \$npos++;\n";
    $compText .= "  }\n";
  }
  if ($needfirst) {
    $compText .= "  my \$rowFirst = \$context->begin()->getRow();\n";
  }
  if ($needlast) {
    $compText .= "  my \$rowLast = \$context->last()->getRow();\n";
  }
  $compText .= $codeResult;
  $compText .= "  \$context->makeArraySend(\$opcode,\n";
  $compText .= $codeBuild;
  $compText .= "  );\n";
  $compText .= "}\n";

  ${$opts->{saveComputeTo}} = $compText if (defined($opts->{saveComputeTo}));
```

The compute function gets assembled from the collected fragments. The optional parts get included only if some of the functions needed them.

```
  # compile the computation function
  my $compFun = eval $compText
    or confess "$myname: error in compilation of the aggregation computation:\n  $@
\nfunction text:\n$compText ";

  # build and add the aggregator
  my $agg = Triceps::AggregatorType->new($rtRes, $opts->{name}, undef, $compFun,
 @compArgs)
    or confess "$myname: internal error: failed to build an aggregator type: $! ";

  $idx->setAggregator($agg)
    or confess "$myname: failed to set the aggregator in the index type: $! ";

  return $opts->{tabType};
}
```

Then the compute function is compiled. In case if the compilation fails, the error message will include both the compilation error and the text of the auto-generated function. Otherwise there would be no way to know, what exactly went wrong.

Well, since no user code is included into the auto-generated function, it should never fail. Except if there is some bad code in the aggregation function definitions. The compiled function and collected closures are then used to create the aggregator, which should also never fail.

The functions that translate the `$%variable` names are built after the same pattern but have the different built-in variables:

```
sub replaceStep # ($varname, $func, $vars, $id, $argCount)
{
  my ($varname, $func, $vars, $id, $argCount) = @_;

  if ($varname eq 'argiter') {
    confess "MySimpleAggregator: internal error in definition of aggregation function
 '$func', step computation refers to 'argiter' but the function declares no arguments"
      unless ($argCount > 0);
    return "\$a${id}";
  } elsif ($varname eq 'niter') {
    return "\$npos";
  } elsif ($varname eq 'groupsize') {
    return "\$context->groupSize()";
  } elsif (exists $vars->{$varname}) {
    return "\$v${id}_${varname}";
  } else {
    confess "MySimpleAggregator: internal error in definition of aggregation function
 '$func', step computation refers to an unknown variable '$varname'"
  }
}

sub replaceResult # ($varname, $func, $vars, $id, $argCount)
{
  my ($varname, $func, $vars, $id, $argCount) = @_;

  if ($varname eq 'argfirst') {
    confess "MySimpleAggregator: internal error in definition of aggregation function
 '$func', result computation refers to '$varname' but the function declares no arguments"
      unless ($argCount > 0);
    return "\$f${id}";
  } elsif ($varname eq 'arglast') {
    confess "MySimpleAggregator: internal error in definition of aggregation function
 '$func', result computation refers to '$varname' but the function declares no arguments"
      unless ($argCount > 0);
    return "\$l${id}";
  } elsif ($varname eq 'groupsize') {
    return "\$context->groupSize()";
  } elsif (exists $vars->{$varname}) {
    return "\$v${id}_${varname}";
  } else {
    confess "MySimpleAggregator: internal error in definition of aggregation function
 '$func', result computation refers to an unknown variable '$varname'"
  }
}
```

They check for the references to the undefined variables and confess if any are found. That's it, the whole aggregator generation.

Now let's look back at the printout of a generated computation function that has been shown above.. The aggregation results were:

```
  result => [
    symbol => "string", "last", sub {$_[0]->get("symbol");},
    id => "int32", "last", sub {$_[0]->get("id");},
    price => "float64", "avg", sub {$_[0]->get("price");},
  ],
```

Which produced the function:

```
sub {
  use strict;
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  return if ($context->groupSize()==0 || $opcode == &Triceps::OP_NOP);
  my $v2_count = 0;
  my $v2_sum = 0;
  my $npos = 0;
  for (my $rhi = $context->begin(); !$rhi->isNull(); $rhi = $context->next($rhi)) {
    my $row = $rhi->getRow();
    # field price=avg
    my $a2 = $args[2]($row);
    { if (defined $a2) { $v2_sum += $a2; $v2_count++; }; }
    $npos++;
  }
  my $rowLast = $context->last()->getRow();
  my $l0 = $args[0]($rowLast);
  my $l1 = $args[1]($rowLast);
  $context->makeArraySend($opcode,
    ($l0), # symbol
    ($l1), # id
    (($v2_count == 0? undef : $v2_sum / $v2_count)), # price
  );
}
```

The fields get assigned the ids 0, 1 and 2. `avg` for the `price` field is the only function here that requires the iteration, and its variables are defined with the prefix `$v2_`. In the loop the function argument closure is called from `$args[2]`, and its result is stored in `$a2` (again, 2 here is the id of this field). Then a copy of the step computation for `avg` is copied in a block, with the variables substituted. `$%argiter` becomes `$a2`, `$%sum` becomes `$v2_sum`, `$%count` becomes `$v2_count`. Then the loop ends.

The functions make use of the last row, so `$rowLast` is computed. The values for the `$%arglast` fields 0 and 1 are calculated in `$l0` and `$l1`. Then the result row is created and sent from an array of substituted result snippets from all the fields. That's how it all works together.

Chapter 12. Joins

12.1. Joins variety

The joins are quite important for the relational data processing, and come in many varieties. And the CEP systems have their own specifics. Basically, in CEP you want the joins to be processed fast. The CEP systems deal with the changing model state, and have to process these changes incrementally.

A small change should be handled fast. It has to use the indexes to find and update all the related result rows. Even though you can make it just go sequentially through all the rows and find the relevant ones, like in a common database, that's not what you normally want. When something like this happens, the usual reaction is “wtf is my model suddenly so slow?” following by an annoyingly long investigation into the reasons of the slowness, and then rewriting the model to make it work faster. It's better to just prevent the slowness in the first place and make sure that the joins always use an index. And since you don't have to deal much with the ad-hoc queries when you write a CEP model, you can provide all the needed indexes in advance very easily.

A particularly interesting kind of joins in this regard is the equi-joins: ones that join the rows by the equality of the fields in them. They allow a very efficient index look-up. Because of this, they are popular in the CEP world. Some systems, like Aleri, support only the equi-joins to start with. The other systems are much more efficient on the equi-joins than on the other kinds of joins. At the moment Triceps follows the fashion of having the advanced support only for the equi-joins. Even though the Sorted/Ordered indexes in Triceps should allow the range-based comparisons to be efficient too, at the moment there are no table methods for the look-up of ranges, they are left for the future work. Of course, nothing stops you from copying an equi-join template and modifying it to work by a dumb iteration. Just it would be slow, and I didn't see much point in it.

There also are three common patterns of the join usage.

In the first pattern the rows sort of go by and get enriched by looking up some information from a table and tacking it onto these rows. Sometimes not even tacking it on but maybe just filtering the data: passing through some of the rows and throwing away the rest, or directing the rows into the different kinds of processing, based on the looked-up data. For a reference, in the Coral8 CCL this situation is called “stream-to-window joins”. In Triceps there are no streams and no windows, so I just call them the “lookup joins”.

In the second pattern multiple stateful tables are joined together. Whenever any of the tables changes, the join result also changes, and the updates get propagated through. This can be done through lookups, but in reality it turns out that defining manually the lookups for the every possible table change becomes tedious pretty quickly. This has to be addressed by the automation.

In the third pattern the same table gets joined recursively, essentially traversing a representation of a tree stored in that table. This actually doesn't work well with the classic SQL unless the recursion depth is strictly limited. There are SQL extensions for the recursive self-joins in the modern databases but I haven't seen them in the CEP systems yet. Anyway, the procedural approach tends to work for this situation much better than the SQLy one, so the templates tend to be of not much help. I'll show a templated and a manual example of this kind for comparison.

12.2. Hello, joins!

As usual, let me show a couple of little teasers before starting the long bottom-up discussion. We'll eventually get by the long way to the same examples, so here I'll show only some very short code snippets and basic explanations.

```
our $join = Triceps::LookupJoin->new(
  name => "join",
  leftFromLabel => $lbTrans,
  rightTable => $tAccounts,
  rightIdxPath => ["lookupSrcExt"],
  leftFields => [ "!acct.*", ".*" ],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
);
```

This is a lookup join that gets the incoming rows with transactions data from the label $lbTrans, finds the account translation in the table $tAccounts, and translates the external account representation to internal one on its output. The join condition is an equivalent of the SQLy

```
on
  lbTrans.acctSrc = tAccounts.source
  and lbTrans.acctXtrId = tAccounts.external
```

The condition looks up the rows in $tAccounts using the index "lookupSrcExt" that must have the key fields source and external.

The result fields will contain all the fields from $lbTrans except those starting with "acct" plus the field internal from $tAccounts that becomes renamed to acct.

Next goes a table join:

```
our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightIdxPath => [ "primary" ],
  type => "inner",
);
```

It joins the tables $tPosition and $tToUsd, with the inner join logic. The table $tPosition uses its index "currencyLookup", and $tToUsd uses its index "primary". The join condition is determined by pairing the key fields of the indexes.

12.3. The lookup join, done manually

First let's look at a lookup done manually. It would also establish the baseline for the further joins.

For the background of the model, let's consider the trade information coming in from multiple sources. Each source system has its own designation of the accounts on which the trades happen but ultimately they are the same accounts. So there is a table that contains the translation from the account designations of various external systems to our system's own internal account identifier. This gets described with the row types:

```
our $rtInTrans = Triceps::RowType->new( # a transaction received
  id => "int32", # the transaction id
  acctSrc => "string", # external system that sent us a transaction
  acctXtrId => "string", # its name of the account of the transaction
  amount => "int32", # the amount of transaction (int is easier to check)
) or confess "$!";

our $rtAccounts = Triceps::RowType->new( # account translation map
  source => "string", # external system that sent us a transaction
  external => "string", # its name of the account in the transaction
  internal => "int32", # our internal account id
) or confess "$!";
```

Other than those basics, the rest of information is only minimal, to keep the examples smaller. Even the trade ids are expected to be global and not per the source systems (which is not realistic but saves another little bit of work).

The accounts table can be indexed in multiple ways for multiple purposes, say:

```
our $ttAccounts = Triceps::TableType->new($rtAccounts)
  ->addSubIndex("lookupSrcExt", # quick look-up by source and external id
    Triceps::IndexType->newHashed(key => [ "source", "external" ])
  )
  ->addSubIndex("iterateSrc", # for iteration in order grouped by source
    Triceps::IndexType->newHashed(key => [ "source" ])
    ->addSubIndex("iterateSrcExt",
      Triceps::IndexType->newHashed(key => [ "external" ])
    )
  )
  ->addSubIndex("lookupIntGroup", # quick look-up by internal id (to multiple externals)
    Triceps::IndexType->newHashed(key => [ "internal" ])
    ->addSubIndex("lookupInt", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttAccounts->initialize() or confess "$!";
```

For our purpose of joining, the first, primary key is the way to go. Using the primary key also has the advantage of making sure that there is no more than one row for each key value.

The first manual lookup example will just do the filtering: find, whether there is a match in the translation table, and if so then pass the row through. The example goes as follows:

```
our $uJoin = Triceps::Unit->new("uJoin");

our $tAccounts = $uJoin->makeTable($ttAccounts,
  "EM_CALL", "tAccounts") or confess "$!";

my $lbFilterResult = $uJoin->makeDummyLabel($rtInTrans, "lbFilterResult");
my $lbFilter = $uJoin->makeLabel($rtInTrans, "lbFilter", undef, sub {
  my ($label, $rowop) = @_;
  my $row = $rowop->getRow();
  my $rh = $tAccounts->findBy(
    source => $row->get("acctSrc"),
    external => $row->get("acctXtrId"),
  );
  if (!$rh->isNull()) {
    $uJoin->call($lbFilterResult->adopt($rowop));
  }
}) or confess "$!";

# label to print the changes to the detailed stats
makePrintLabel("lbPrint", $lbFilterResult);

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "acct") {
    $uJoin->makeArrayCall($tAccounts->getInputLabel(), @data);
  } elsif ($type eq "trans") {
    $uJoin->makeArrayCall($lbFilter, @data);
  }
  $uJoin->drainFrame(); # just in case, for completeness
}
```

The `findBy()` is where the join actually happens: the lookup of the data in a table by values from another row. Very similar to what the basic window example in Section 9.1: “Hello, tables!” (p. 73) was doing before. It's `findBy()`, without the need for `findByIdx()`, because in this case the index type used in the accounts table is its first leaf index, to which `findBy()` defaults. After that the fact of successful or unsuccessful lookup is used to pass the original row through or throw it away. If the found row were used to pick some fields from it and stick them into the result, that would be a more complete join, more like what you often expect to see.

And here is an example of the input processing:

```
acct,OP_INSERT,source1,999,1
acct,OP_INSERT,source1,2011,2
acct,OP_INSERT,source2,ABCD,1
trans,OP_INSERT,1,source1,999,100
lbFilterResult OP_INSERT id="1" acctSrc="source1" acctXtrId="999"
    amount="100"
trans,OP_INSERT,2,source2,ABCD,200
lbFilterResult OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200"
trans,OP_INSERT,3,source2,QWERTY,200
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
lbFilterResult OP_DELETE id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200"
acct,OP_DELETE,source1,999,1
```

It starts with populating the accounts table. Then the transactions that find the match pass, and those who don't find don't pass. If more of the account translations get added later, the transactions for them start passing but as you can see, the result might be slightly unexpected: you may get a DELETE that had no matching previous INSERT, as happened for the row with id=3. This happens because the lookup join keeps no history on its left side and can't react properly to the changes to the table on the right. Because of this, the lookup joins work best when the reference table gets pre-populated in advance and then stays stable.

12.4. The LookupJoin template

When a join has to produce the new rows, with the data from both the incoming row and the ones looked up in the reference table, this can also be done manually but may be more convenient to do with the LookupJoin template. The translation of account to the internal ids can be done like this:

```
our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => ["lookupSrcExt"],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  isLeft => 1,
); # would confess by itself on an error

# label to print the changes to the detailed stats
makePrintLabel("lbPrint", $join->getOutputLabel());

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "acct") {
    $uJoin->makeArrayCall($tAccounts->getInputLabel(), @data);
  } elsif ($type eq "trans") {
    $uJoin->makeArrayCall($join->getInputLabel(), @data);
  }
  $uJoin->drainFrame(); # just in case, for completeness
}
```

The join gets defined in the option name-value format. The options "unit" and "name" are as usual.

The incoming rows are always on the left side, the table on the right. LookupJoin can do either the inner join or the left outer join (since it does not react to the changes of the right table and has no access to the past data from the left side, the full and right outer joins are not available). In this case the option "isLeft => 1" selects the left outer join. The left outer join also happens to be the default if this option is not used.

The left side is described by the option "leftRowType", and causes the join's input label of this row type to be created. The input label can be found with `$join->getInputLabel()`.

The right side is a table, specified in the option "rightTable". The lookups in the table are done using a combination of an index and the field pairing. The option "by" provides the field pairing. It contains the pairs of field names, one from the left, and one from the right, for the equal fields. They can be separated by "`,`" too, but "`=>`" feels more idiomatic to me. These fields from the left are translated to the right and are used for lookup through the index. The index is specified with the path in the option "rightIdxPath". If this option is missing, LookupJoin will just try to find the first top-level Hashed index. Either way, the index must be a Hashed index.

There is no particular reason for it not being a Sorted/Ordered index, other that the `getKey()` call does not work for these indexes yet, and that's what the LookupJoin uses to check that the right-side index key matches the join key in "by". The order of the fields in the option "by" and in the index may vary but the set of the fields must be the same.

The index may be either a leaf (as in this example) or non-leaf. If it's a leaf, it could look up no more than one row per key, and LookupJoin uses this internally for a little optimization. Otherwise LookupJoin is capable of producing multiple result rows for one input row.

Finally, there is the result row. It is built out of the two original rows by picking the fields according to the options "leftFields" and "rightFields". If either option is missing, that means "take all the fields". The format of these options is from `Triceps::Fields::filterToPairs()` that has been described in Section 10.7: "Result projection in the templates" (p. 128) . So in this example `[ "internal/acct" ]` means: pass the field `internal` but rename it to `acct`.

Remember that the field names in the result must not duplicate. It would be an error. So if the duplications happen, use the substitution syntax to rename some of the fields.

A fairly common usage in joins is to just give the unique prefixes to the left-side and right-side fields. This can be achieved with:

```
  leftFields => [ '.*/left_$&' ],
  rightFields => [ '.*/right_$&' ],
```

The $& in the substitution gets replaced with the whole matched field name.

The option "fieldsLeftFirst" determines, which side will go first in the result. By default it's set to 1 (as in this example), and the left side goes first. If set to 0, the right side would go first.

This setup for the result row types is somewhat clumsy but it's a reasonable first attempt.

Now, having gone through the description, an example of how it works:

```
acct,OP_INSERT,source1,999,1
acct,OP_INSERT,source1,2011,2
acct,OP_INSERT,source2,ABCD,1
trans,OP_INSERT,1,source1,999,100
join.out OP_INSERT id="1" acctSrc="source1" acctXtrId="999"
    amount="100" acct="1"
trans,OP_INSERT,2,source2,ABCD,200
join.out OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200" acct="1"
trans,OP_INSERT,3,source2,QWERTY,200
join.out OP_INSERT id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200"
```

```
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
join.out OP_DELETE id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200" acct="2"
acct,OP_DELETE,source1,999,1
```

Same as before, first the accounts table gets populated, then the transactions are sent. If an account is not found, this left outer join still passes through the original fields from the left side. Adding an account later doesn't help the rowops that already went through but the new rowops will see it. The same goes for deleting an account, it doesn't affect the past rowops either.

The left-side data can also be specified in another way: the option “leftFromLabel” provides a label which in turn provides both the input row type and the unit. You can still specify the unit option as well but it must match the one in the label. This is driven internally by `Triceps::Opt::handleUnitTypeLabel()`, described in Section 10.5: “Template options” (p. 117) , so it follows the same rules. The join still has its own input label but it gets automatically chained to the one in the option. For an example of such a join:

```
our $lbTrans = $uJoin->makeDummyLabel($rtInTrans, "lbTrans");

our $join = Triceps::LookupJoin->new(
  name => "join",
  leftFromLabel => $lbTrans,
  rightTable => $tAccounts,
  rightIdxPath => ["lookupSrcExt"],
  leftFields => [ "id", "amount" ],
  fieldsLeftFirst => 0,
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  isLeft => 0,
); # would confess by itself on an error

# label to print the changes to the detailed stats
makePrintLabel("lbPrint", $join->getOutputLabel());

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "acct") {
    $uJoin->makeArrayCall($tAccounts->getInputLabel(), @data);
  } elsif ($type eq "trans") {
    $uJoin->makeArrayCall($lbTrans, @data);
  }
  $uJoin->drainFrame(); # just in case, for completeness
}
```

The other options demonstrate the possibilities described in the last post. This time it's an inner join, the result has the right-side fields going first, and the left-side fields are filtered in the result by an explicit list of fields to pass.

Another way to achieve the same filtering of the left-side fields would be by throwing away everything starting with “acct” and passing through the rest:

```
  leftFields => [ "!acct.*", ".*" ],
```

And here is an example of a run:

```
acct,OP_INSERT,source1,999,1
acct,OP_INSERT,source1,2011,2
acct,OP_INSERT,source2,ABCD,1
trans,OP_INSERT,1,source1,999,100
join.out OP_INSERT acct="1" id="1" amount="100"
```

```
trans,OP_INSERT,2,source2,ABCD,200
join.out OP_INSERT acct="1" id="2" amount="200"
trans,OP_INSERT,3,source2,QWERTY,200
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
join.out OP_DELETE acct="2" id="3" amount="200"
acct,OP_DELETE,source1,999,1
```

The input data is the same as the last time, but the result is different. Since it's an inner join, the rows that don't find a match don't pass through. And of course the fields are ordered and subsetted differently in the result.

The next example loses all connection with reality, it just serves to demonstrate another ability of LookupJoin: matching multiple rows on the right side for an incoming row. The situation itself is obviously useful and normal, just it's not what normally happens with the account id translation, and I was too lazy to invent another realistically-looking example.

```
our $ttAccounts2 = Triceps::TableType->new($rtAccounts)
  ->addSubIndex("iterateSrc", # for iteration in order grouped by source
    Triceps::IndexType->newHashed(key => [ "source" ])
    ->addSubIndex("lookupSrcExt",
      Triceps::IndexType->newHashed(key => [ "external" ])
      ->addSubIndex("grouping", Triceps::IndexType->newFifo())
    )
  )
or confess "$!";
$ttAccounts2->initialize() or confess "$!";

our $tAccounts = $uJoin->makeTable($ttAccounts2,
  "EM_CALL", "tAccounts") or confess "$!";

our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => [ "iterateSrc", "lookupSrcExt" ],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
); # would confess by itself on an error
```

The main loop is unchanged from the first LookupJoin example, so I wont' copy it here. Just for something different, the join index here is nested, and its path consists of two elements. It's not a leaf index either, with one FIFO level under it. And when the “isLeft” is not specified explicitly, it defaults to 1, making it a left join.

The example of a run uses a slightly different input, highlighting the ability to match multiple rows:

```
acct,OP_INSERT,source1,999,1
acct,OP_INSERT,source1,2011,2
acct,OP_INSERT,source2,ABCD,1
acct,OP_INSERT,source2,ABCD,10
acct,OP_INSERT,source2,ABCD,100
trans,OP_INSERT,1,source1,999,100
join.out OP_INSERT id="1" acctSrc="source1" acctXtrId="999"
    amount="100" acct="1"
trans,OP_INSERT,2,source2,ABCD,200
join.out OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200" acct="1"
join.out OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200" acct="10"
join.out OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200" acct="100"
trans,OP_INSERT,3,source2,QWERTY,200
```

```
join.out OP_INSERT id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200"
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
join.out OP_DELETE id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200" acct="2"
acct,OP_DELETE,source1,999,1
```

When a row matches multiple rows in the table, it gets multiplied. The join function iterates through the whole matching row group, and for each found row creates a result row and calls the output label with it.

Now, what if you don't want to get multiple rows back even if they are found? Of course, the best way is to just use a leaf index. But once in a while you get into situations with the denormalized data in the lookup table. You might know in advance that for each row in an index group a certain field would be the same. Or you might not care, what exact value you get as long as it's from the right group. But you might really not want the input rows to multiply when they go through the join. LookupJoin has a solution:

```
our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => [ "iterateSrc", "lookupSrcExt" ],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  limitOne => 1,
); # would confess by itself on an error
```

The option “limitOne” changes the processing logic to pick only the first matching row. It also optimizes the join function. If “limitOne” is not specified explicitly, the join constructor deduces it magically by looking at whether the join index is a leaf or not. Actually, for a leaf index it would always override “limitOne” to 1, even if you explicitly set it to 0.

With the limit, the same input produces a different output:

```
acct,OP_INSERT,source1,999,1
acct,OP_INSERT,source1,2011,2
acct,OP_INSERT,source2,ABCD,1
acct,OP_INSERT,source2,ABCD,10
acct,OP_INSERT,source2,ABCD,100
trans,OP_INSERT,1,source1,999,100
join.out OP_INSERT id="1" acctSrc="source1" acctXtrId="999"
    amount="100" acct="1"
trans,OP_INSERT,2,source2,ABCD,200
join.out OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200" acct="1"
trans,OP_INSERT,3,source2,QWERTY,200
join.out OP_INSERT id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200"
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
join.out OP_DELETE id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200" acct="2"
acct,OP_DELETE,source1,999,1
```

Now it just picks the first matching row instead of multiplying the rows.

12.5. Manual iteration with LookupJoin

Sometimes you might want to just get the list of the resulting rows from LookupJoin and iterate over them by yourself, rather than have it call the labels. To be honest, this looked kind of important when I wrote LookupJoin first, but by now I don't see a whole lot of use in it. By now, if you want to do a manual iteration, calling `findBy()` and then iterating looks like a more useful option. But at the time there was no `findBy()`, and this feature came to exist. Here is an example:

```
our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => ["lookupSrcExt"],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  automatic => 0,
); # would confess by itself on an error

# label to print the changes to the detailed stats
my $lbPrint = makePrintLabel("lbPrint", $join->getOutputLabel());

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "acct") {
    $uJoin->makeArrayCall($tAccounts->getInputLabel(), @data);
  } elsif ($type eq "trans") {
    my $op = shift @data; # drop the opcode field
    my $trans = $rtInTrans->makeRowArray(@data) or confess "$!";
    my @rows = $join->lookup($trans);
    foreach my $r (@rows) {
      $uJoin->call($lbPrint->makeRowop($op, $r)) or confess "$!";
    }
  }
  $uJoin->drainFrame(); # just in case, for completeness
}
```

It copies the first LookupJoin example, only now with a manual iteration. Once the option “automatic” is set to 0 for the join, the method `$join->lookup()` becomes available to perform the lookup and return the result rows in an array (the data sent to the input label keeps working as usual, sending the result rows to the output label). This involves the extra overhead of keeping all the result rows (and there might be lots of them) in an array, so by default the join is compiled in an automatic-only mode.

Since `lookup()` returns rows, not rowops, and knows nothing about the opcodes, those had to be handled separately around the lookup.

The result is the same as for the first example, only the name of the result label differs:

```
acct,OP_INSERT,source1,999,1
acct,OP_INSERT,source1,2011,2
acct,OP_INSERT,source2,ABCD,1
trans,OP_INSERT,1,source1,999,100
lbPrint OP_INSERT id="1" acctSrc="source1" acctXtrId="999"
    amount="100" acct="1"
trans,OP_INSERT,2,source2,ABCD,200
lbPrint OP_INSERT id="2" acctSrc="source2" acctXtrId="ABCD"
    amount="200" acct="1"
trans,OP_INSERT,3,source2,QWERTY,200
lbPrint OP_INSERT id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200"
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
lbPrint OP_DELETE id="3" acctSrc="source2" acctXtrId="QWERTY"
    amount="200" acct="2"
acct,OP_DELETE,source1,999,1
```

The print label is still connected to the output label of the LookupJoin, but it's done purely for the convenience of its creation. Since no rowops get sent to the LookupJoin's input, none get to its output, and none get from there to the output label. Instead the main loop creates and sends the rowops directly to the output label when it iterates through the lookup results. Because of this the label name in the output is the name of the output label.

12.6. The key fields of LookupJoin

The key fields are the ones that participate in the join condition. I use these terms interchangeably because by the definition of LookupJoin, these fields must be the key fields in the join index in the right-side table. LookupJoin has a few more facilities for their handling that haven't been shown yet.

First, the join condition can be specified as the `Triceps::Fields::filterToPairs()` patterns in the option “byLeft”. The options “by” and “byLeft” are mutually exclusive and one of them must be present. The condition

```
by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
```

can be also specified as:

```
byLeft => [ "acctSrc/source", "acctXtrId/external" ],
```

The option name “byLeft” says that the pattern specification is for the fields on the left side (there is no symmetric “byRight”). The substitutions produce the matching field names for the right side. Unlike the result pattern, here the fields that do not find a match do not get included in the key. It's as if an implicit `"!.*"` gets added at the end. In fact, `"!.*"` really does get added implicitly at the end.

Of course, for the example above either option doesn't make much difference. It starts making the difference when the key fields follow a pattern. For example, if the key fields on both sides have the names `acctSrc` and `acctXtrId`, the specification with the “byLeft” becomes a little simpler:

```
byLeft => [ "acctSrc", "acctXtrId" ],
```

Even more so if the key is long, common on both sides, and all the fields have a common prefix. Such as:

```
k_AccountSystem
k_AccountId
k_InstrumentSystem
k_InstrumentId
k_TransactionDate
k_SettlementDate
```

Then the join condition can be specified simply as:

```
byLeft => [ "k_.*" ],
```

If say the settlement date doesn't matter for a particular join, it can be excluded:

```
byLeft => [ "!k_SettlementDate", "k_.*" ],
```

If the right side represents a swap of securities, it might have two parts to it, each describing its half with its key:

```
BorrowAccountSystem
BorrowAccountId
BorrowInstrumentSystem
BorrowInstrumentId
BorrowTransactionDate
BorrowSettlementDate
LoanAccountSystem
LoanAccountId
LoanInstrumentSystem
LoanInstrumentId
LoanTransactionDate
LoanSettlementDate
```

Then the join of the one-sided rows with the borrow part condition can be done using:

```
byLeft => [ 'k_(.*)/Borrow$1' ],
```

The key patterns make the long keys easier to drag around.

Second, key fields of LookupJoin don't have to be of the same type on the left and on the right side. Since the key building for lookup is done through Perl, the key values get automatically converted as needed.

A caveat is that the conversion might be not exactly direct. If a string gets converted to a number, then any string values that do not look like numbers will be converted to 0. A conversion between a string and a floating-point number, in either direction, is likely to lose precision. A conversion between int64 and int32 may cause the upper bits to be truncated. So what gets looked up may be not what you expect.

I'm not sure yet if I should add the requirement for the types being exactly the same. The automatic conversions seem to be convenient, just use them with care. I suppose, when the joins will get eventually implemented in the C++ code, this freedom would go away because it's much easier and more efficient in C++ to copy the field values as-is than to convert them.

The only thing currently checked is whether a field is represented in Perl as a scalar or an array, and that must match on the left and on the right. Note that the array `uint8[]` gets represented in Perl as a scalar string, so an `uint8[]` field can be matched with other scalars but not with the other arrays.

Third, the key fields have the problem of duplication. The LookupJoin is by definition an equi-join, it joins together the rows that have the same values in a set of key fields. If all the fields from both sides are to be included in the result, they key values will be present in it twice, once from the left side, once from the right side. This is not what is usually wanted, and the good practice is to let these fields through from one side and filter out from the other side.

Letting these fields through on the left side is usually the better choice. For the inner joins it doesn't really matter but for the left outer joins it works much better than the with letting through the fields from the right side. The reason is that when the join doesn't find the match on the right side, all the right-side fields will be NULL. If you pass through the key fields only from the right side, they will contain NULL, and this is probably not what you want.

However if for some reason, be it the order of the fields or the better field types on the right side, you really want to pass the key fields only from the right side, you can. LookupJoin provides a special magic act enabled by the option

```
  fieldsMirrorKey => 1
```

Then if the row is not found on the right side, a special right-side row will be created with the key fields copied from the left side, and it will be used to produce the result row. With “fieldsMirrorKey” you are guaranteed to always have the key values present on the right side.

12.7. A peek inside LookupJoin

I won't be describing in the details the internals of LookupJoin. They seem a bit too big and complicated. Partially it's because the code is of an older origin, and not using all the newer calls. Partially it's because when I wrote it, I've tried to optimize by translating the rows to an array format instead of referring to the fields by names, and that made the code more tricky. Partially, the code has grown more complex due to all the added options. And partially the functionality just is a little tricky by itself.

But, for debugging purposes, the LookupJoin constructor can return the auto-generated code of the joiner function. It's done with the option “saveJoinerTo”:

```
  saveJoinerTo => \$code,
```

This will cause the auto-generated code to be placed into the variable `$code`. I've collected a few such examples in this section. They provide a glimpse into the internal workings of the joiner. It's definitely a quite advanced topic, but it's helpful if you want to know, what is really going on in there.

The joiner code from the example

```
our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => ["lookupSrcExt"],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  isLeft => 1,
); # would confess by itself on an error
```

that was shown first in the Section 12.4: “The LookupJoin template” (p. 174) is this:

```
sub # ($inLabel, $rowop, $self)
{
  my ($inLabel, $rowop, $self) = @_;
  #print STDERR "DEBUGX LookupJoin " . $self->{name} . " in: ", $rowop->printP(), "\n";

  my $opcode = $rowop->getOpcode(); # pass the opcode
  my $row = $rowop->getRow();

  my @leftdata = $row->toArray();

  my $resRowType = $self->{resultRowType};
  my $resLabel = $self->{outputLabel};

  my $lookuprow = $self->{rightRowType}->makeRowHash(
    source => $leftdata[1],
    external => $leftdata[2],
    );

  #print STDERR "DEBUGX " . $self->{name} . " lookup: ", $lookuprow->printP(), "\n";
  my $rh = $self->{rightTable}->findIdx($self->{rightIdxType}, $lookuprow);
  Carp::confess("$!") unless defined $rh;

  my @rightdata; # fields from the right side, defaults to all-undef, if no data found
  my @result; # the result rows will be collected here

  if (!$rh->isNull()) {
    #print STDERR "DEBUGX " . $self->{name} . " found data: " . $rh->getRow()->printP() .
 "\n";
    @rightdata = $rh->getRow()->toArray();
  }

    my @resdata = ($leftdata[0],
    $leftdata[1],
    $leftdata[2],
    $leftdata[3],
    $rightdata[2],
    );
    my $resrowop = $resLabel->makeRowop($opcode, $resRowType->makeRowArray(@resdata));
    #print STDERR "DEBUGX " . $self->{name} . " +out: ", $resrowop->printP(), "\n";
    Carp::confess("$!") unless defined $resrowop;
    Carp::confess("$!")
      unless $resLabel->getUnit()->call($resrowop);

}
```

From the example with the manual iteration:

```
our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => ["lookupSrcExt"],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  automatic => 0,
); # would confess by itself on an error
```

comes this code:

```
sub  # ($self, $row)
{
  my ($self, $row) = @_;

  #print STDERR "DEBUGX LookupJoin " . $self->{name} . " in: ", $row->printP(), "\n";

  my @leftdata = $row->toArray();

  my $lookuprow = $self->{rightRowType}->makeRowHash(
    source => $leftdata[1],
    external => $leftdata[2],
    );

  #print STDERR "DEBUGX " . $self->{name} . " lookup: ", $lookuprow->printP(), "\n";
  my $rh = $self->{rightTable}->findIdx($self->{rightIdxType}, $lookuprow);
  Carp::confess("$!") unless defined $rh;

  my @rightdata; # fields from the right side, defaults to all-undef, if no data found
  my @result; # the result rows will be collected here

  if (!$rh->isNull()) {
    #print STDERR "DEBUGX " . $self->{name} . " found data: " . $rh->getRow()->printP() .
 "\n";
    @rightdata = $rh->getRow()->toArray();
  }

    my @resdata = ($leftdata[0],
    $leftdata[1],
    $leftdata[2],
    $leftdata[3],
    $rightdata[2],
    );
    push @result, $self->{resultRowType}->makeRowArray(@resdata);
    #print STDERR "DEBUGX " . $self->{name} . " +out: ", $result[$#result]->printP(),
 "\n";
  return @result;
}
```

It takes different arguments because now it's not an input label handler but a common function that gets called from both the label handler and the `lookup()` method. And it collects the rows in an array to be returned instead of immediately passing them on.

From the example with multiple rows matching on the right side

```
our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => [ "iterateSrc", "lookupSrcExt" ],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
); # would confess by itself on an error
```

comes this code:

```
sub # ($inLabel, $rowop, $self)
{
  my ($inLabel, $rowop, $self) = @_;
  #print STDERR "DEBUGX LookupJoin " . $self->{name} . " in: ", $rowop->printP(), "\n";

  my $opcode = $rowop->getOpcode(); # pass the opcode
  my $row = $rowop->getRow();

  my @leftdata = $row->toArray();

  my $resRowType = $self->{resultRowType};
  my $resLabel = $self->{outputLabel};

  my $lookuprow = $self->{rightRowType}->makeRowHash(
    source => $leftdata[1],
    external => $leftdata[2],
    );

  #print STDERR "DEBUGX " . $self->{name} . " lookup: ", $lookuprow->printP(), "\n";
  my $rh = $self->{rightTable}->findIdx($self->{rightIdxType}, $lookuprow);
  Carp::confess("$!") unless defined $rh;

  my @rightdata; # fields from the right side, defaults to all-undef, if no data found
  my @result; # the result rows will be collected here

  if ($rh->isNull()) {
    #print STDERR "DEBUGX " . $self->{name} . " found NULL\n";

    my @resdata = ($leftdata[0],
    $leftdata[1],
    $leftdata[2],
    $leftdata[3],
    $rightdata[2],
    );
    my $resrowop = $resLabel->makeRowop($opcode, $resRowType->makeRowArray(@resdata));
    #print STDERR "DEBUGX " . $self->{name} . " +out: ", $resrowop->printP(), "\n";
    Carp::confess("$!") unless defined $resrowop;
    Carp::confess("$!")
      unless $resLabel->getUnit()->call($resrowop);

  } else {
    #print STDERR "DEBUGX " . $self->{name} . " found data: " . $rh->getRow()->printP() .
 "\n";
    my $endrh = $self->{rightTable}->nextGroupIdx($self->{iterIdxType}, $rh);
    for (; !$rh->same($endrh); $rh = $self->{rightTable}->nextIdx($self->{rightIdxType},
 $rh)) {
      @rightdata = $rh->getRow()->toArray();
    my @resdata = ($leftdata[0],
    $leftdata[1],
    $leftdata[2],
    $leftdata[3],
    $rightdata[2],
    );
    my $resrowop = $resLabel->makeRowop($opcode, $resRowType->makeRowArray(@resdata));
    #print STDERR "DEBUGX " . $self->{name} . " +out: ", $resrowop->printP(), "\n";
    Carp::confess("$!") unless defined $resrowop;
    Carp::confess("$!")
```

```
      unless $resLabel->getUnit()->call($resrowop);

    }
  }
}
```

It's more complicated in two ways: If a match is found, it has to iterate through the whole matching group. And if the match is not found, it still has to produce a result row for the left join with a separate code fragment.

12.8. JoinTwo joins two tables

Fundamentally, joining the two tables is kind of like the two symmetrical copies of LookupJoin, each of them reacting to the changes in one table and doing look-ups in another table. For all I can tell, the CEP systems with the insert-only stream model tend to start with the assumption that the LookupJoin (or whetever they call it) is good enough. Then it turns out that manually writing the join twice where it can be done once is a pain. So the table-to-table join gets added. Then the interesting nuances crop up, since a correct table-to-table join has more to it than just two stream-to-table joins. Then it turns out that it would be real convenient to propagate the deletes through the join, and that gets added as a special feature behind the scenes.

In Triceps, JoinTwo is the template for joining the tables. And actually it is translated under the hood to two LookupJoins, but it has more on top of them.

In a common database a join query causes a join plan to be created: on what table to iterate, and in which to look up next. A CEP system deals with the changing data, and a join has to react to the data changes on each of its input tables. It must have multiple plans, one for starting from each of the tables. And essentially a LookupJoin embodies such a plan, and JoinTwo makes two of them.

Why only two? Because it's the minimal usable number. The join logic is tricky, so it's better to work out the kinks on something simpler first. And it still can be scaled to many tables by joining them in stages. It's not quite as efficient as a direct join of multiple tables, because the result of each stage has to be put into a table, but it does the job.

I'll be doing the demonstrations of the table joins on an application example from the area of stock lending. Think of a large multinational broker that wants to keep track of its lending activities. It has many customers to whom the stock can be loaned or from whom it can be borrowed. This information comes as the records of positions, of how many shares are loaned or borrowed for each customer, and at what contractual price. And since the clients are from all around the world, the prices may be in different currencies. A simplified and much shortened version of the position information may look like this:

```
our $rtPosition = Triceps::RowType->new( # a customer account position
  date => "int32", # as of which date, in format YYYYMMDD
  customer => "string", # customer account id
  symbol => "string", # stock symbol
  quantity => "float64", # number of shares
  price => "float64", # share price in local currency
  currency => "string", # currency code of the price
) or confess "$!";
```

Then we want to aggregate these data in different ways, getting the broker-wide summaries by the symbol, by customer etc. The aggregation is updated as the business day goes on. At the end of the business day the state of the day freezes, and the new day's initial data is loaded. That's why the business date is part of the schema. If you wonder, the next day's initial data is usually the same as at the end of the previous day, except where some contractual conditions change. The detailed position data is thrown away after a few days, or even right at the end of the day, but the aggregation results from the end of the day are kept for a longer history.

There is a problem with summing up the monetary values: they come in different currencies and can not be added up directly. If we want to get this kind of summaries, we have to translate all of them to a single reference currency. That's what the sample joins will be doing: finding the translation rates to the US dollars. The currency rates come in the translation schema:

```
our $rtToUsd = Triceps::RowType->new( # a currency conversion to USD
  date => "int32", # as of which date, in format YYYYMMDD
  currency => "string", # currency code
  toUsd => "float64", # multiplier to convert this currency to USD
) or confess "$!";
```

Since the currency rates change all the time, to make sense of a previous day's position, the previous day's rates need to be kept around, and so the rates are also marked with a date.

Having the mood set, here is the first example of a model with an inner join:

```
# exchange rates, to convert all currencies to USD
our $ttToUsd = Triceps::TableType->new($rtToUsd)
  ->addSubIndex("primary",
    Triceps::IndexType->newHashed(key => [ "date", "currency" ])
  )
  ->addSubIndex("byDate", # for cleaning by date
    Triceps::SimpleOrderedIndex->new(date => "ASC")
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttToUsd->initialize() or confess "$!";

# the positions in the original currency
our $ttPosition = Triceps::TableType->new($rtPosition)
  ->addSubIndex("primary",
    Triceps::IndexType->newHashed(key => [ "date", "customer", "symbol" ])
  )
  ->addSubIndex("currencyLookup", # for joining with currency conversion
    Triceps::IndexType->newHashed(key => [ "date", "currency" ])
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
  ->addSubIndex("byDate", # for cleaning by date
    Triceps::SimpleOrderedIndex->new(date => "ASC")
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttPosition->initialize() or confess "$!";

our $uJoin = Triceps::Unit->new("uJoin");

our $tToUsd = $uJoin->makeTable($ttToUsd,
  "EM_CALL", "tToUsd") or confess "$!";
our $tPosition = $uJoin->makeTable($ttPosition,
  "EM_CALL", "tPosition") or confess "$!";

our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightIdxPath => [ "primary" ],
  type => "inner",
); # would confess by itself on an error

# label to print the changes to the detailed stats
makePrintLabel("lbPrint", $join->getOutputLabel());

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "cur") {
    $uJoin->makeArrayCall($tToUsd->getInputLabel(), @data);
  } elsif ($type eq "pos") {
    $uJoin->makeArrayCall($tPosition->getInputLabel(), @data);
  }
  $uJoin->drainFrame(); # just in case, for completeness
}
```

The example just does the joining, leaving the aggregation to the imagination of the reader. The result of a JoinTwo is not stored in a table. It is a stream of ephemeral updates, same as for LookupJoin. If you want to keep them, you can put them into a table yourself (and maybe do the aggregation in the same table).

Both the joined tables must provide a Hashed index for the efficient joining. The index may be leaf (selecting one row per key) or non-leaf (containing multiple rows per key) but it must be there. This makes sure that the joins are always efficient and you don't have to hunt for why your model is suddenly so slow.

The indexes also provide the default way of finding the join condition: the key fields in the indexes are paired up together, in the order they go in the index specifications. Once again, the fields are paired not by name but by order. If the indexes are nested, the outer indexes precede in the order. For example, the `$ttToUsd` could have the same index done in a nested way and it would work just as well:

```
  ->addSubIndex("byDate",
    Triceps::IndexType->newHashed(key => [ "date" ])
    ->addSubIndex("primary",
      Triceps::IndexType->newHashed(key => [ "currency" ])
    )
  )
```

Same as with LookupJoin, currently only the Hashed indexes are supported, and must go through all the path. The outer index “byDate” here can not be a Sorted/Ordered index, that would be an error and the join will refuse to accept it.

But if the order of key fields in the `$ttToUsd` index were changed to be different from `$ttPosition`, like this

```
  ->addSubIndex("primary",
    Triceps::IndexType->newHashed(key => [ "currency", "date" ])
  )
```

then it would be a mess. The wrong fields would be matched up in the join condition, which would become `(tPosition.date == tToUsd.currency && tPosition.currency == tToUsd.date)`, and everything would go horribly wrong.

Incidentally, this situation in this particular case would be caught because JoinTwo is much less lenient than LookupJoin as the key field types go. It requires the types of the matching fields to be exactly the same. Partially, for the reasons of catching the wrong order, partially for the sake of the result consistency. JoinTwo does the look-ups in both directions. And think about what happens if a string field and an int32 field get matched up, and then the non-numeric strings turn up in the string field, containing things like “abc” and “qwerty”. Those strings on the left side will match the rows with numeric 0 on the right side. But then if the row with 0 on the right side changes, it would look for the string “0” on the left, which would not find either “abc” or “qwerty”. The state of the join will become a mess. So no automatic key type conversions here.

By the way, even though JoinTwo doesn't refuse to have the float64 key fields, using them is a bad idea. The floating-point values are subject to non-obvious rounding. And if you have two floating-point values that print the same, this doesn't mean that they are internally the same down to the last bit (because the printing involves the conversion to decimal that involves rounding). The joining requires that the values are exactly equal. Because of this the joining on a floating-point field is rife with unpleasant surprises. Better don't do it. A possible solution is to round values by converting them to integers (scaled by multiplying by a fixed factor to get essentially a fixed-point value). You can even convert them back from fixed-point to floating-point and still join on these floating-point values, because the same values would always be produced from integers in exactly the same way, and will be exactly the same.

If you wonder, there are ways to specify the key fields pairing explicitly, with the option “by” or “byLeft”, same as for the LookupJoin. But that's all they can do: change the pairing of the key fields. You can't specify any other fields. And since they are optional for JoinTwo, they get usually skipped.

More of the JoinTwo options closely parallel those in LookupJoin. Obviously, “name”, “rightTable” and “rightIdxPath” are the same, with the added symmetrical “leftTable” and “leftIdxPath”. There is no “unit” option though, the unit is always taken from the tables (which must belong to the same unit). The option to save the source code of the generated joiner code has been split in two: “leftSaveJoinerTo” and “rightSaveJoinerTo”. Since JoinTwo has to react to the updates from both sides, is has to have two handlers. And since internally it uses two LookupJoin for this purpose, these happen to be the joiner functions of the left and right LookupJoin.

The option “type” selects the inner join mode. The inner join is the default, and would have been used even if this option was not specified.

The options controlling the result are also the same as in LookupJoin: “leftFields”, “rightFields”, “fieldsLeftFirst”. The results in this example include all the fields from both sides by default. JoinTwo is smart and knows how to exclude the duplicate key fields automatically.

The joins are currently not equipped to actually compute the translated prices directly. They can only look up the information for it, and the computation can be done later, before or during the aggregation.

That's enough explanations for now, let's look at the result. The input rows are shown as usual in bold, and to make keeping track easier, I broke up the output into short snippets with commentary after each one.

```
cur,OP_INSERT,20120310,USD,1
cur,OP_INSERT,20120310,GBP,2
cur,OP_INSERT,20120310,EUR,1.5
```

Inserting the reference currencies produces no result, since it's an inner join and they have no matching positions yet.

```
pos,OP_INSERT,20120310,one,AAA,100,15,USD
join.leftLookup.out OP_INSERT date="20120310" customer="one"
    symbol="AAA" quantity="100" price="15" currency="USD" toUsd="1"
pos,OP_INSERT,20120310,two,AAA,100,8,GBP
join.leftLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2"
```

Now the positions arrive and find the matching translations to USD. The label names on the output are an interesting artifact of all the chained labels receiving the original rowop that refers to the first label in the chain. Which happens to be the output label of a LookupJoin inside JoinTwo. It works conveniently for the demonstrational purposes, since the name of that LookupJoin shows whether the row that triggered the result came from the left or right side of the JoinTwo.

```
pos,OP_INSERT,20120310,three,AAA,100,300,RUR
```

This position is out of luck: no translation for its currency. The inner join is actually not a good choice here. If a row does not pass through because of the lack of translation, it gets excluded even from the aggregations that do not require the translation, such as those that total up the quantity of a particular symbol across all the customers. A left outer join would have been suited better.

```
pos,OP_INSERT,20120310,three,BBB,200,80,GBP
join.leftLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2"
```

Another position arrives, same as before.

```
cur,OP_INSERT,20120310,RUR,0.04
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
    toUsd="0.04"
```

The translation for RUR finally comes in. The position in RUR can now find its match and propagate through.

```
cur,OP_DELETE,20120310,GBP,2
join.rightLookup.out OP_DELETE date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2"
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2"
```

```
cur,OP_INSERT,20120310,GBP,2.2
join.rightLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2.2"
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2.2"
```

An exchange rate update for GBP arrives. It amounts to “delete the old translation and then insert a new one”. Each of these operations updates the state of the join: the disappearing translation causes all the GBP positions to be deleted from the result, and the new translation inserts them back, with the new value of toUsd. Which is the correct behavior: to make an up date to the result positions, they have to be deleted and then inserted witn the new values.

```
pos,OP_DELETE,20120310,one,AAA,100,15,USD
join.leftLookup.out OP_DELETE date="20120310" customer="one"
    symbol="AAA" quantity="100" price="15" currency="USD" toUsd="1"
pos,OP_INSERT,20120310,one,AAA,200,16,USD
join.leftLookup.out OP_INSERT date="20120310" customer="one"
    symbol="AAA" quantity="200" price="16" currency="USD" toUsd="1"
```

A position update arrives. Again, it's a delete-and-insert, and propagates through the join as such.

That's the end of the first example. The commentary said that the left outer join would have been better for the logic, so let's make one for the left outer join. All we need to change is the join type option:

```
our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightIdxPath => [ "primary" ],
  type => "left",
); # would confess by itself on an error
```

Now the positions would pass through even if the currency translation is not available. The same input now produces a different result:

```
cur,OP_INSERT,20120310,USD,1
cur,OP_INSERT,20120310,GBP,2
cur,OP_INSERT,20120310,EUR,1.5
pos,OP_INSERT,20120310,one,AAA,100,15,USD
join.leftLookup.out OP_INSERT date="20120310" customer="one"
    symbol="AAA" quantity="100" price="15" currency="USD" toUsd="1"
pos,OP_INSERT,20120310,two,AAA,100,8,GBP
join.leftLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2"
```

So far things are going the same as for the inner join.

```
pos,OP_INSERT,20120310,three,AAA,100,300,RUR
join.leftLookup.out OP_INSERT date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
```

The first difference: even though there is no translation for RUR, the row still passes through (with the field `toUsd` being NULL).

```
pos,OP_INSERT,20120310,three,BBB,200,80,GBP
join.leftLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2"
```

This is also unchanged.

```
cur,OP_INSERT,20120310,RUR,0.04
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
```

```
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
    toUsd="0.04"
```

The second difference: since this row from the left side has already passed through, just sending another INSERT for it would make the data inconsistent. The original result without the translation must be deleted first, and then a new one, with translation, inserted. JoinTwo is smart enough to figure it out all by itself.

```
cur,OP_DELETE,20120310,GBP,2
join.rightLookup.out OP_DELETE date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2"
join.rightLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP"
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2"
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP"
```

The same logic works for the deletes, only backwards: when the translation for GBP is deleted, the result rows that used it change to the lose the translation.

```
cur,OP_INSERT,20120310,GBP,2.2
join.rightLookup.out OP_DELETE date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP"
join.rightLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2.2"
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP"
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2.2"
```

And again, when the new translation for GBP comes in, the DELETE-INSERT sequence is done for each of the rows. As you can see, the update of the GBP translation in the last two snippets worked in not the most efficient way. Fundamentally, if we knew that a DELETE of GBP will be immediately followed by an INSERT, we could skip inserting and then deleting the rows with the NULL in `toUsd`. But we don't know, and in Triceps there is no way to know it.

If you really, really want to avoid the propagation of these intermediate changes, insert after the join a Collapse template described in Section 14.2: “Collapsed updates” (p. 227), and flush it only after the whole update has been processed. There will be more overhead in the Collapse itself, but all the logic below it will skip the intermediate changes. If this logic below is heavy-weight, that might be an overall win. A caveat though: a Collapse requires that the data has a primary key, a JoinTwo doesn't require its result (nor its inputs) to have a primary key. Because of this, the collapse might not work right with every possible join, you'd have to limit yourself to the joins that produce the data with a primary key.

```
pos,OP_DELETE,20120310,one,AAA,100,15,USD
join.leftLookup.out OP_DELETE date="20120310" customer="one"
    symbol="AAA" quantity="100" price="15" currency="USD" toUsd="1"
pos,OP_INSERT,20120310,one,AAA,200,16,USD
join.leftLookup.out OP_INSERT date="20120310" customer="one"
    symbol="AAA" quantity="200" price="16" currency="USD" toUsd="1"
```

And the rest is again the same as with an inner join.

JoinTwo can do a right outer join too, just use the type “right”. It works in exactly the same way as the left outer join, just with a different table. So much the same that it's not even worth a separate example.

Now, the full outer join. The full outer joins usually get used with a variation of the “fork-join” topology described in the Section 14.1: “The dreaded diamond” (p. 223). In it the processing of a row can be forked into multiple parallel paths, each path doing an optional part of the comuptation and either providing a result row or not, eventually with all the parts merged back together into one row. The full outer join is a convenient way to do this merge: the paths that didn't produce the result get quietly ignored, and the results that were produced get merged back into a single row. The row in such situations is usually identified by a primary key, so the partial results can find each other. This scheme makes the most sense when the paths are executed in the parallel threads, or when the processing on some paths may get delayed and then continued later. If the processing is single-threaded and fast, Triceps provides a more convenient procedural way of getting the same result: just call every path in order and merge the results from them procedurally, and you won't have to keep the intermediate results in their tables forever, nor delete them manually.

Even though that use is typical, it has only the 1:1 record matching and does not highlight all the abilities of the JoinTwo. So, let's come up with another example that does.

The positions-and-currencies do not lend itself easily to a full outer join but we'll make them do. Suppose that you want to get the total count of positions (per symbol, or altogether), or maybe the total value, for every currency. Including those for which we have the exchange rates but no positions, for them the count should simply be 0 (or maybe NULL). And those for which there are positions but no exchange rate translations. This is a job for a full outer join, followed by an aggregation. The join has the type "outer" and looks like this:

```
our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightIdxPath => [ "primary" ],
  type => "outer",
); # would confess by itself on an error
```

As before, the aggregation part will be left to the imagination of the reader. This join has the many-to-one (M:1) row matching, since there might be multiple positions on the left matching one currency rate translation on the right. This will create interesting effects in the output, let's look at it:

```
cur,OP_INSERT,20120310,GBP,2
join.rightLookup.out OP_INSERT date="20120310" currency="GBP"
    toUsd="2"
```

The first translation gets through, even though there is no position for it yet.

```
pos,OP_INSERT,20120310,two,AAA,100,8,GBP
join.leftLookup.out OP_DELETE date="20120310" currency="GBP" toUsd="2"
join.leftLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2"
```

The first position for an existing translation comes in. Now the GBP row has a match, so the unmatched row gets deleted and a matched one gets inserted instead.

```
pos,OP_INSERT,20120310,three,BBB,200,80,GBP
join.leftLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2"
```

The second position for GBP works differently: since there is no unmatched row any more (it was taken care of by the first position), there is nothing to delete. Just the second matched row gets inserted.

```
pos,OP_INSERT,20120310,three,AAA,100,300,RUR
join.leftLookup.out OP_INSERT date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
```

The position without a matching currency get through as well.

```
cur,OP_INSERT,20120310,RUR,0.04
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
    toUsd="0.04"
```

Now the RUR translation becomes available and it has to do the same things as we've seen before, only on the other side: delete the unmatched record and replace it with the matched one.

```
cur,OP_DELETE,20120310,GBP,2
join.rightLookup.out OP_DELETE date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2"
join.rightLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP"
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2"
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP"
cur,OP_INSERT,20120310,GBP,2.2
join.rightLookup.out OP_DELETE date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP"
join.rightLookup.out OP_INSERT date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2.2"
join.rightLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP"
join.rightLookup.out OP_INSERT date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2.2"
```

Then the GBP translation gets updated. First the old translation gets deleted and then the new one inserted. When the translation gets deleted, all the positions in GBP lose their match. So the matched rows gets deleted and replaced with the unmatched ones. When the new GBP translation is inserted, the replacement goes in the other direction.

```
pos,OP_DELETE,20120310,three,BBB,200,80,GBP
join.leftLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2.2"
```

When this position goes away, the row gets deleted from the result as well. However it was not the only position in GBP, so there is no need to insert an unmatched record for GBP.

```
pos,OP_DELETE,20120310,three,AAA,100,300,RUR
join.leftLookup.out OP_DELETE date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
    toUsd="0.04"
join.leftLookup.out OP_INSERT date="20120310" currency="RUR"
    toUsd="0.04"
```

This position was the last one in RUR. So when it gets deleted, the RUR translation has no match any more. That means, after deleting the matched row from the results, the unmatched row has to be inserted to keep the balance right.

This business with keeping track of the unmatched rows is not unique to the full outer joins. Remember, it was showing in the left outer joins too, and the right outer joins are no exception either. When the first matching row gets inserted or the last matching row gets deleted on the side that is opposite to the "outer side", the unmatched rows have to be handled in the result. (That would be the right side for the left outer joins, the left side for the right outer joins, and either side for the full outer joins). The special thing about the M:1 (and 1:M and M:M) joins is that there may be more than one matching row. On insertion, the second and following matching rows produce a different effect than the first one. On deletion, the opposite: all the rows but the last work differently from the last one. It's not limited to the full outer joins. M:1 or M:M with a right outer join, and 1:M or M:M with a left outer join will do it too.

If you're like me, by now you'd be wondering, how does it work? If the “opposite side” is of “one” variety (:1 or 1:), which can be known from it using a leaf index for the join, then every insert is the first insert of a matching row for this key, and every delete is the delete of the last row for this key. Which means, do the empty-match business every time.

If the “opposite side” is of the “many” variety (:M or M:), with a non-leaf index, then things get more complicated. The join works by processing the rowops coming out of the argument tables. When it gets the rowop in such a situation, it goes to the table and checks, was it the first (or last) row for this key? And then uses this knowledge to act.

12.9. The key field duplication in JoinTwo

JoinTwo in its raw form has the same problem of the key field duplication as LookupJoin (described in Section 12.6: “The key fields of LookupJoin” (p. 180)), only worse because the table-to-table outer joins must work with the updates from any side. So JoinTwo has more magic built into it: it knows how to recognize this situation and have the key fields copied into the result from whatever side happens to be present for a particular row, and does this by default. It makes these fields always available on both sides. And along the way it also takes care of modifying the option “rightFields” or “leftFields” to actually pass through only one of the copies.

The default behavior is good enough for most situations. But if you want more control, it's done with the option “fieldsUniqKey”. The default value of this option is “first”. It means: Enable the magic for copying the fields from the non-NULL side to the NULL side. Look at the option “fieldsLeftFirst” and figure out, which side goes first in the result. Let the key fields pass on that side unchanged (though the user can block them on that side manually too, or possibly rename them, it's his choice). On the other side, automatically generate the blocking specs for the key fields and prepend them to that side's result specification. It's smart enough to know that an undefined “leftFields” or “rightFields” means the same as “.*”, so an undefined result spec is replaced by the blocking specs followed by “.*”. If you later call the methods

```
$fspec = $join->getLeftFields();
$fspec = $join->getRightFields();
```

then you will actually get back the modified field specs.

If you want the key fields to be present in a different location in the result, you can set “fieldsUniqKey” to “left” or “right”. That will make them pass through on the selected side, and the blocking would be automatically added on the other side.

For more control yet, set this option to “manual”. The magic for making the key fields available on both sides will still be enabled, but no automatic blocking. You can pick and choose the result fields manually, exactly as you want. Remember though that there can't be multiple fields with the same name in the result, so if both sides have these fields named the same, you've got to block or rename one of the two copies.

The final choice is “none”: it simply disables the key field magic.

12.10. The override options in JoinTwo

Normally JoinTwo tries to work in a consistent manner, refusing to do the unsafe things that might corrupt the data. But if you really, really want, and are really sure of what you're doing, there are options to override these restrictions.

If you set

```
  overrideSimpleMinded => 1,
```

then the logic that produces the DELETE-INSERT sequences for the outer joins gets disabled. The only reason I can think of to use this option is if you want to simulate a CEP system that has no concept of opcodes. So if your data is INSERT-only and you want to produce the INSERT-only data too, and want the dumbed-down logic, this option is your solution.

The option

```
  overrideKeyTypes => 1,
```

disables the check for the exact match of the key field types. This might come helpful for example if you have an int32 field on one side and an int64 field on the other side, and you know that in reality they would always stay within the int32 range. Or if you have an integer on one side and a string that always contains an integer on the other side. Since you know that the type conversions can always be done with no loss, you can safely override the type check and still get the correct result.

12.11. JoinTwo input event filtering

Let's look at how the business day logic interacts with the joins. It's typical for the business applications to keep the full data for the current day, or a few recent days, then clear the data that became old and maybe keep it only in an aggregated form.

So, let's add the business day logic to the left join example. It uses the indexes by date to find the rows that have become old:

```
# exchange rates, to convert all currencies to USD
our $ttToUsd = Triceps::TableType->new($rtToUsd)
  ->addSubIndex("primary",
    Triceps::IndexType->newHashed(key => [ "date", "currency" ])
  )
  ->addSubIndex("byDate", # for cleaning by date
    Triceps::SimpleOrderedIndex->new(date => "ASC")
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttToUsd->initialize() or confess "$!";

# the positions in the original currency
our $ttPosition = Triceps::TableType->new($rtPosition)
  ->addSubIndex("primary",
    Triceps::IndexType->newHashed(key => [ "date", "customer", "symbol" ])
  )
  ->addSubIndex("currencyLookup", # for joining with currency conversion
    Triceps::IndexType->newHashed(key => [ "date", "currency" ])
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
  ->addSubIndex("byDate", # for cleaning by date
    Triceps::SimpleOrderedIndex->new(date => "ASC")
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttPosition->initialize() or confess "$!";

# remember the indexes for the future use
our $ixtToUsdByDate = $ttToUsd->findSubIndex("byDate") or confess "$!";
our $ixtPositionByDate = $ttPosition->findSubIndex("byDate") or confess "$!";

# Go through the table and clear all the rows where the field "date"
# is less than the date argument. The index type orders the table by date.
sub clearByDate($$$) # ($table, $ixt, $date)
{
  my ($table, $ixt, $date) = @_;

  my $next;
  for (my $rhit = $table->beginIdx($ixt); !$rhit->isNull(); $rhit = $next) {
    last if (($rhit->getRow()->get("date")) >= $date);
    $next = $rhit->nextIdx($ixt); # advance before removal
    $table->remove($rhit);
  }
}
```

The table types are the same as have been already shown before, they've been copied here for convenience. `clearByDate()` is an universal function that can clear the contents of any table by date, provided that the date is in the field “date” and the index type on this table that orders the rows by date is given as an argument. The index with ordering by date must be not just a leaf Ordered index, but have a FIFO index nested in it. Without that FIFO index, the Ordered index would allow only one row for each date.

The main loop gets extended with a few more commands:

```
our $businessDay = undef;

our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightIdxPath => [ "primary" ],
  type => "left",
); # would confess by itself on an error

# label to print the changes to the detailed stats
makePrintLabel("lbPrint", $join->getOutputLabel());

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "cur") {
    $uJoin->makeArrayCall($tToUsd->getInputLabel(), @data);
  } elsif ($type eq "pos") {
    $uJoin->makeArrayCall($tPosition->getInputLabel(), @data);
  } elsif ($type eq "day") { # set the business day
    $businessDay = $data[0] + 0; # convert to an int
  } elsif ($type eq "clear") { # clear the previous day
    # flush the left side first, because it's an outer join
    &clearByDate($tPosition, $ixtPositionByDate, $businessDay);
    &clearByDate($tToUsd, $ixtToUsdByDate, $businessDay);
  }
  $uJoin->drainFrame(); # just in case, for completeness
}
```

The roll-over to the next business day (after the input data previously shown with the left join example) then looks like this:

```
day,20120311
clear
join.leftLookup.out OP_DELETE date="20120310" customer="two"
    symbol="AAA" quantity="100" price="8" currency="GBP" toUsd="2.2"
join.leftLookup.out OP_DELETE date="20120310" customer="three"
    symbol="AAA" quantity="100" price="300" currency="RUR"
    toUsd="0.04"
join.leftLookup.out OP_DELETE date="20120310" customer="three"
    symbol="BBB" quantity="200" price="80" currency="GBP" toUsd="2.2"
join.leftLookup.out OP_DELETE date="20120310" customer="one"
    symbol="AAA" quantity="200" price="16" currency="USD" toUsd="1"
```

Clearing the left-side table before the right-side one is more efficient than the other way around, since this is a left outer join, and since it's an M:1 join. If the right-side table were cleared first, it would first update all the result records to change all the right-side fields in them to NULL, and then the clearing of the left-side table would finally delete these rows. Clearing the left side first removes this churn: it deletes all the rows from the result right away, and then when the right side is cleared, it still tries to look up the matching rows but finds nothing and produces no result. For an inner join the order would not matter: either one would produce the same amount of churn. For a full outer join, the M:1 consideration would come into play, and removing the rows from the left side first would still be more efficient. This way when it removes multiple position rows that match the same currency, all of them but one generate the simple DELETEs, and only the last one would follow up with an INSERT that has only the right-side data in it. That row with the right-side data will get deleted when the currency row gets deleted from the right side. If the right side were deleted first, deleting each row on the right side would cause an output of a DELETE-INSERT result pair for each of its matching position rows from the left side, and would produce more churn. For the 1:1 or M:M full outer joins, the order would not matter.

If you don't want these deletions to propagate though the rest of your model, you can just put a filtering logic after the join, to throw away all the modifications for the previous days. Through don't forget that you would have then to delete the previous-day data from the rest of the model's tables manually.

If you want to keep only the aggregated data, you may want to pass the join output to the aggregator without filtering and then filter the aggregator's output, thus stopping the updates to the aggregation results. You may even have a special logic in the aggregator, that would ignore the groups of the previous days. Such optimization of the aggregation filtering will be shown in the Section 13.1: “Time-limited propagation” (p. 211). And they aren't any less efficient than filtering on the output of the join, because if you filter after the join, you'd still have to remove the rows from the aggregation table, and would still have to filter after the aggregation too.

Now, suppose that you want to be extra optimal and don't want any join look-ups to happen at all when you delete the old data. JoinTwo has a feature that lets you do that. You can make it receive the events not directly from the tables but after filtering, using the options “leftFromLabel” and “rightFromLabel”:

```
our $lbPositionCurrent = $uJoin->makeDummyLabel(
  $tPosition->getRowType, "lbPositionCurrent") or confess "$!";
our $lbPositionFilter = $uJoin->makeLabel($tPosition->getRowType,
  "lbPositionFilter", undef, sub {
    if ($_[1]->getRow()->get("date") >= $businessDay) {
      $uJoin->call($lbPositionCurrent->adopt($_[1]));
    }
  }) or confess "$!";
$tPosition->getOutputLabel()->chain($lbPositionFilter) or confess "$!";

our $lbToUsdCurrent = $uJoin->makeDummyLabel(
  $tToUsd->getRowType, "lbToUsdCurrent") or confess "$!";
our $lbToUsdFilter = $uJoin->makeLabel($tToUsd->getRowType,
  "lbToUsdFilter", undef, sub {
    if ($_[1]->getRow()->get("date") >= $businessDay) {
      $uJoin->call($lbToUsdCurrent->adopt($_[1]));
    }
  }) or confess "$!";
$tToUsd->getOutputLabel()->chain($lbToUsdFilter) or confess "$!";

our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftFromLabel => $lbPositionCurrent,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightFromLabel => $lbToUsdCurrent,
  rightIdxPath => [ "primary" ],
  type => "left",
); # would confess by itself on an error
```

The same clearing now looks like this:

```
day,20120311
clear
```

No output is coming from the join whatsoever. It all gets cut off before it reaches the join. It's not such a great gain though. Remember that if you want to keep the aggregated data, you would still have to delete the original rows manually from the aggregation table afterwards. And the filtering logic will add overhead, not only during the clearing but all the time.

If you're not careful with the filtering conditions, it's also easy to make the results of the join inconsistent. This example filters both input tables on the same key field, with the same condition, so the output will stay always consistent. But if any of these elements were missing, it becomes possible to produce inconsistent output that has the DELETEs of different rows than INSERTs, and deletions of the rows that haven't been inserted in the first place. The reason is that even though the input events are filtered, the table look-ups done by JoinTwo aren't. If some row comes from the right side and gets thrown away by the filter, and then another row comes on the left side, passes the filter, and then finds a match in that thrown-away right-side row, it will use that row in the result. And the join would think that the right-side row has already been seen, and would produce an incorrect update.

So these options don't make a whole lot of a win but make a major opportunity for a mess, and probably should never be used. And will probably be deleted in the future, unless someone finds a good use for them. They have been added because at the time they provided a roundabout way to do a self-join. But the later fixes to the Table logic make the self-joins possible without this kind of perversions.

12.12. Self-join done with JoinTwo

The self-joins happen when a table is joined to itself. For an example of a model with self-joins, let's look at the Forex trading. People exchange the currencies in every possible direction in multiple markets. The Forex exchange rates are quoted for every pair of currencies, in every direction.

Naturally, if you exchange one currency into another and then back into the first one, you normally end up with less money than you've started with. The rest becomes the transaction cost and lines the pockets of the brokers, market makers and exchanges.

However once in a while some interesting things happen. If the exchange rates between the different currencies become disbalanced, you may be able to exchange the currency A for currency B for currency C and back for currency A, and end up with more money than you've started with. (You don't have to do it in sequence, you would normally do all three transactions in parallel). However it's a short-lived opportunity: as you perform the transactions, you'll be changing the involved exchange rates towards the balance, and you won't be the only one exploiting this opportunity, so you better act fast. This activity of bringing the market into balance while simultaneously extracting profit is called “arbitration”.

So let's make a model that will detect such arbitration opportunities, for the following automated execution. Mind you, it's all grossly simplified, but it shows the gist of it. And most importantly, it uses the self-joins. Here we go:

```
our $rtRate = Triceps::RowType->new( # an exchange rate between two currencies
  ccy1 => "string", # currency code
  ccy2 => "string", # currency code
  rate => "float64", # multiplier when exchanging ccy1 to ccy2
) or confess "$!";

# all exchange rates
our $ttRate = Triceps::TableType->new($rtRate)
  ->addSubIndex("byCcy1",
    Triceps::IndexType->newHashed(key => [ "ccy1" ])
    ->addSubIndex("byCcy12",
      Triceps::IndexType->newHashed(key => [ "ccy2" ])
    )
  )
  ->addSubIndex("byCcy2",
    Triceps::IndexType->newHashed(key => [ "ccy2" ])
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttRate->initialize() or confess "$!";

our $uArb = Triceps::Unit->new("uArb");

our $tRate = $uArb->makeTable($ttRate,
  &Triceps::EM_CALL, "tRate") or confess "$!";

our $join1 = Triceps::JoinTwo->new(
  name => "join1",
  leftTable => $tRate,
  leftIdxPath => [ "byCcy2" ],
  leftFields => [ "ccy1", "ccy2", "rate/rate1" ],
  rightTable => $tRate,
  rightIdxPath => [ "byCcy1" ],
  rightFields => [ "ccy2/ccy3", "rate/rate2" ],
); # would die by itself on an error
our $ttJoin1 = Triceps::TableType->new($join1->getResultRowType())
  ->addSubIndex("byCcy123",
    Triceps::IndexType->newHashed(key => [ "ccy1", "ccy2", "ccy3" ])
  )
  ->addSubIndex("byCcy31",
    Triceps::IndexType->newHashed(key => [ "ccy3", "ccy1" ])
    ->addSubIndex("grouping", Triceps::IndexType->newFifo())
  )
or confess "$!";
$ttJoin1->initialize() or confess "$!";
our $tJoin1 = $uArb->makeTable($ttJoin1,
  &Triceps::EM_CALL, "tJoin1") or confess "$!";
$join1->getOutputLabel()->chain($tJoin1->getInputLabel()) or confess "$!";

our $join2 = Triceps::JoinTwo->new(
  name => "join2",
  leftTable => $tJoin1,
  leftIdxPath => [ "byCcy31" ],
  rightTable => $tRate,
  rightIdxPath => [ "byCcy1", "byCcy12" ],
  rightFields => [ "rate/rate3" ],
  # the field ordering in the indexes is already right, but
  # for clarity add an explicit join condition too
  byLeft => [ "ccy3/ccy1", "ccy1/ccy2" ],
); # would die by itself on an error

# now compute the resulting circular rate and filter the profitable loops
our $rtResult = Triceps::RowType->new(
  $join2->getResultRowType()->getdef(),
  looprate => "float64",
) or confess "$!";
my $lbResult = $uArb->makeDummyLabel($rtResult, "lbResult");
my $lbCompute = $uArb->makeLabel($join2->getResultRowType(), "lbCompute", undef, sub {
  my ($label, $rowop) = @_;
  my $row = $rowop->getRow();
  my $looprate = $row->get("rate1") * $row->get("rate2") * $row->get("rate3");

  if ($looprate > 1) {
    $uArb->makeHashCall($lbResult, $rowop->getOpcode(),
      $row->toHash(),
      looprate => $looprate,
    );
  } else {
      print("__", $rowop->printP(), "looprate=$looprate \n"); # for debugging
  }
}) or confess "$!";
$join2->getOutputLabel()->chain($lbCompute) or confess "$!";

# label to print the changes to the detailed stats
makePrintLabel("lbPrint", $lbResult);
#makePrintLabel("lbPrintJoin1", $join1->getOutputLabel());
#makePrintLabel("lbPrintJoin2", $join2->getOutputLabel());

while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "rate") {
    $uArb->makeArrayCall($tRate->getInputLabel(), @data);
  }
  $uArb->drainFrame(); # just in case, for completeness
}
```

The rate quotes will be coming into `$tRate`. The indexes are provided to both work with the self-joins and to have a primary index as the first leaf.

There are no special options for the self-join in JoinTwo: just use the same table for both the left and right side. The first join represents two exchange transactions, so it's done by matching the second currency of the first quote to the first currency of the second quote. The result contains three currency names and two rate multipliers. The second join adds one more rate multiplier, returning back to the first currency. Now to learn the effect of the circular conversion we only need to multiply all the multipliers. If it comes out below 1, the cycling transaction would return a loss, if above 1, a profit.

The label `$lbCompute` with Perl handler performs the multiplication, and if the result is over 1, passes the result to the next label `$lbResult`, from which then the data gets printed. I've also added a debugging printout in case if the row doesn't get through. That one starts with “__” and helps seeing what goes on inside when no result is coming out.

Finally, the main loop reads the data and puts it into the rates table, thus driving the logic.

Now let's take a look at an example of a run, with interspersed commentary.

```
rate,OP_INSERT,EUR,USD,1.48
rate,OP_INSERT,USD,EUR,0.65
rate,OP_INSERT,GBP,USD,1.98
rate,OP_INSERT,USD,GBP,0.49
```

The rate quotes start coming in. Note that the rates are separate for each direction of exchange. So far nothing happens because there aren't enough quotes to complete a loop of three steps.

```
rate,OP_INSERT,EUR,GBP,0.74
__join2.leftLookup.out OP_INSERT ccy1="EUR" ccy2="GBP" rate1="0.74"
    ccy3="USD" rate2="1.98" rate3="0.65" looprate=0.95238
__join2.leftLookup.out OP_INSERT ccy1="USD" ccy2="EUR" rate1="0.65"
    ccy3="GBP" rate2="0.74" rate3="1.98" looprate=0.95238
__join2.rightLookup.out OP_INSERT ccy1="GBP" ccy2="USD" rate1="1.98"
    ccy3="EUR" rate2="0.65" rate3="0.74" looprate=0.95238
rate,OP_INSERT,GBP,EUR,1.30
__join2.leftLookup.out OP_INSERT ccy1="GBP" ccy2="EUR" rate1="1.3"
    ccy3="USD" rate2="1.48" rate3="0.49" looprate=0.94276
__join2.leftLookup.out OP_INSERT ccy1="USD" ccy2="GBP" rate1="0.49"
    ccy3="EUR" rate2="1.3" rate3="1.48" looprate=0.94276
__join2.rightLookup.out OP_INSERT ccy1="EUR" ccy2="USD" rate1="1.48"
    ccy3="GBP" rate2="0.49" rate3="1.3" looprate=0.94276
```

Now there are enough currencies in play to complete the loop. None of them get the loop rate over 1 though, so the only printouts are from the debugging logic. There are only two loops, but each of them is printed three times. Why? It's a loop, so you can start from each of its elements and come back to the same element. One row for each starting point. And the joins find all of them.

To find and eliminate the duplicates, the order of currencies in the rows can be rotated to put the alphabetically lowest currency code first. Note that they can't be just sorted because the relative order matters. Trading in the order GBP-USD-EUR will give a different result than GBP-EUR-USD. The relative order has to be preserved. I didn't put any such elimination into the example to keep it smaller.

```
rate,OP_DELETE,EUR,USD,1.48
__join2.leftLookup.out OP_DELETE ccy1="EUR" ccy2="USD" rate1="1.48"
    ccy3="GBP" rate2="0.49" rate3="1.3" looprate=0.94276
__join2.leftLookup.out OP_DELETE ccy1="GBP" ccy2="EUR" rate1="1.3"
    ccy3="USD" rate2="1.48" rate3="0.49" looprate=0.94276
__join2.rightLookup.out OP_DELETE ccy1="USD" ccy2="GBP" rate1="0.49"
    ccy3="EUR" rate2="1.3" rate3="1.48" looprate=0.94276
rate,OP_INSERT,EUR,USD,1.28
__join2.leftLookup.out OP_INSERT ccy1="EUR" ccy2="USD" rate1="1.28"
    ccy3="GBP" rate2="0.49" rate3="1.3" looprate=0.81536
__join2.leftLookup.out OP_INSERT ccy1="GBP" ccy2="EUR" rate1="1.3"
    ccy3="USD" rate2="1.28" rate3="0.49" looprate=0.81536
__join2.rightLookup.out OP_INSERT ccy1="USD" ccy2="GBP" rate1="0.49"
    ccy3="EUR" rate2="1.3" rate3="1.28" looprate=0.81536
```

Someone starts changing lots of euros for dollars, and the rate moves. No good news for us yet though.

```
rate,OP_DELETE,USD,EUR,0.65
__join2.leftLookup.out OP_DELETE ccy1="USD" ccy2="EUR" rate1="0.65"
    ccy3="GBP" rate2="0.74" rate3="1.98" looprate=0.95238
__join2.leftLookup.out OP_DELETE ccy1="GBP" ccy2="USD" rate1="1.98"
    ccy3="EUR" rate2="0.65" rate3="0.74" looprate=0.95238
__join2.rightLookup.out OP_DELETE ccy1="EUR" ccy2="GBP" rate1="0.74"
    ccy3="USD" rate2="1.98" rate3="0.65" looprate=0.95238
rate,OP_INSERT,USD,EUR,0.78
lbResult OP_INSERT ccy1="USD" ccy2="EUR" rate1="0.78" ccy3="GBP"
    rate2="0.74" rate3="1.98" looprate="1.142856"
lbResult OP_INSERT ccy1="GBP" ccy2="USD" rate1="1.98" ccy3="EUR"
    rate2="0.78" rate3="0.74" looprate="1.142856"
lbResult OP_INSERT ccy1="EUR" ccy2="GBP" rate1="0.74" ccy3="USD"
    rate2="1.98" rate3="0.78" looprate="1.142856"
```

The rate for dollars-to-euros follows its opposite. This creates an arbitration opportunity! Step two: trade in the direction USD-EUR-GBP-USD, step three: PROFIT!!!

```
rate,OP_DELETE,EUR,GBP,0.74
lbResult OP_DELETE ccy1="EUR" ccy2="GBP" rate1="0.74" ccy3="USD"
    rate2="1.98" rate3="0.78" looprate="1.142856"
lbResult OP_DELETE ccy1="USD" ccy2="EUR" rate1="0.78" ccy3="GBP"
    rate2="0.74" rate3="1.98" looprate="1.142856"
lbResult OP_DELETE ccy1="GBP" ccy2="USD" rate1="1.98" ccy3="EUR"
    rate2="0.78" rate3="0.74" looprate="1.142856"
rate,OP_INSERT,EUR,GBP,0.64
__join2.leftLookup.out OP_INSERT ccy1="EUR" ccy2="GBP" rate1="0.64"
    ccy3="USD" rate2="1.98" rate3="0.78" looprate=0.988416
__join2.leftLookup.out OP_INSERT ccy1="USD" ccy2="EUR" rate1="0.78"
    ccy3="GBP" rate2="0.64" rate3="1.98" looprate=0.988416
__join2.rightLookup.out OP_INSERT ccy1="GBP" ccy2="USD" rate1="1.98"
    ccy3="EUR" rate2="0.78" rate3="0.64" looprate=0.988416
```

Our trading (and perhaps other people's trading too) moves the exchange rate of euros to pounds. And with that the balance of currencies is restored, and the arbitration opportunity disappears.

Now let's have a look inside JoinTwo. What is so special about the self-join? Normally the join works on two separate tables. They get updated one at a time. Even if some common reason causes both tables to be updated, the update arrives from one table first. The join sees this incoming update, looks in the unchanged second table, produces an updated result. Then the update from the second table comes to the join, which takes it, looks in the already modified first table, and produces another updated result.

If both inputs are from the same table, this logic breaks. Two copies of the updates will arrive, but by the time the first one arrives, the contents of the table has been already changed. When the join looks in the table, it gets the unexpected results and creates a mess.

But JoinTwo has a fix for this. It makes use of the Pre label of the table for its left-side update (the right side would have worked just as good, it's just a random choice):

```
  my $selfJoin = $self->{leftTable}->same($self->{rightTable});
  if ($selfJoin && !defined $self->{leftFromLabel}) {
    # one side must be fed from Pre label (but still let the user override)
    $self->{leftFromLabel} = $self->{leftTable}->getPreLabel();
  }
```

This way when the join sees the first update, the table hasn't changed yet. And then the second copy of that update comes though the normal output label, after the table has been modified. Everything just works out as normal and the self-joins produce the correct result.

Normally you don't need to concern yourself with this, except if you're trying to filter the data coming to the join. Then remember that for “leftFromLabel” you have to receive the data from the table's `getPreLabel()`, not `getOutputLabel()`.

12.13. Self-join done manually

In many cases the self-joins are better suited to be done by the manual looping through the data. This is especially true if the table represents a tree, linked by the parent-child node id and the processing has to navigate through the tree. Indeed, if the tree may be of an arbitrary depth, there is no way to handle if with the common joins, you will need as many joins as the depth of the tree (through there are some SQL extensions for the recursive self-joins).

The arbitration example can also be conveniently rewritten through the manual loops. The input row type, table type, table, unit, and the main loop do not change, so I won't copy them the second time. The rest of the code is:

```
our $rtResult = Triceps::RowType->new(
  ccy1 => "string", # currency code
  ccy2 => "string", # currency code
  ccy3 => "string", # currency code
  rate1 => "float64",
  rate2 => "float64",
  rate3 => "float64",
  looprate => "float64",
) or confess "$!";
my $ixtCcy1 = $ttRate->findSubIndex("byCcy1") or confess "$!";
my $ixtCcy12 = $ixtCcy1->findSubIndex("byCcy12") or confess "$!";

my $lbResult = $uArb->makeDummyLabel($rtResult, "lbResult");
my $lbCompute = $uArb->makeLabel($rtRate, "lbCompute", undef, sub {
  my ($label, $rowop) = @_;
  my $row = $rowop->getRow();
  my $ccy1 = $row->get("ccy1");
  my $ccy2 = $row->get("ccy2");
  my $rate1 = $row->get("rate");

  my $rhi = $tRate->findIdxBy($ixtCcy1, ccy1 => $ccy2);
  my $rhiEnd = $rhi->nextGroupIdx($ixtCcy12)
    or confess "$!";
  for (; !$rhi->same($rhiEnd); $rhi = $rhi->nextIdx($ixtCcy12)) {
    my $row2 = $rhi->getRow();
    my $ccy3 = $row2->get("ccy2");
    my $rate2 = $row2->get("rate");

    my $rhj = $tRate->findIdxBy($ixtCcy12, ccy1 => $ccy3, ccy2 => $ccy1);
    # it's a leaf primary index, so there may be no more than one match
    next
      if ($rhj->isNull());
    my $row3 = $rhj->getRow();
    my $rate3 = $row3->get("rate");
    my $looprate = $rate1 * $rate2 * $rate3;

    # now build the row in normalized order of currencies
    print("____Order before: $ccy1, $ccy2, $ccy3\n");
    my $result;
    if ($ccy2 lt $ccy3) {
      if ($ccy2 lt $ccy1) { # rotate left
        $result = $lbResult->makeRowopHash($rowop->getOpcode(),
          ccy1 => $ccy2,
          ccy2 => $ccy3,
          ccy3 => $ccy1,
          rate1 => $rate2,
          rate2 => $rate3,
          rate3 => $rate1,
          looprate => $looprate,
        );
      }
    } else {
      if ($ccy3 lt $ccy1) { # rotate right
        $result = $lbResult->makeRowopHash($rowop->getOpcode(),
          ccy1 => $ccy3,
          ccy2 => $ccy1,
          ccy3 => $ccy2,
          rate1 => $rate3,
          rate2 => $rate1,
          rate3 => $rate2,
          looprate => $looprate,
        );
      }
    }
    if (!defined $result) { # use the straight order
      $result = $lbResult->makeRowopHash($rowop->getOpcode(),
        ccy1 => $ccy1,
        ccy2 => $ccy2,
        ccy3 => $ccy3,
        rate1 => $rate1,
        rate2 => $rate2,
        rate3 => $rate3,
        looprate => $looprate,
      );
    }
    if ($looprate > 1) {
      $uArb->call($result);
    } else {
      print("__", $result->printP(), "\n"); # for debugging
    }
  }
}) or confess "$!";
$tRate->getOutputLabel()->chain($lbCompute) or confess "$!";
makePrintLabel("lbPrint", $lbResult);
```

Whenever a new rowop is processed in the table, it goes to the label `$lbCompute`. The row in this rowop is the first leg of the triangle. The loop then finds all the possible second legs that can be connected to the first leg. And then for each second leg it checks whether it can make the third leg back to the original currency. If it can, good, we've found a candidate for a result row.

The way the loops work, this time there is no triplication. But the same triangle still can be found starting from any of its three currencies. This means that to keep the data consistent, no matter what was the first currency in a particular run, it still must produce the exact same result row. To achieve that, the currencies get rotated as explained in the previous section, making sure that the first currency is has the lexically smallest name. These if-else statements do that by selecting the direction of rotation (if any) and build the result record in one of three ways.

Finally it compares the combined rate to 1, and if greater then sends the result. If not, a debugging printout starting with “__” prints the row, so that is can be seen. Another debugging printout prints the original order of the currencies, letting us check that the rotation was performed correctly.

On feeding the same input data this code produces the result:

```
rate,OP_INSERT,EUR,USD,1.48
rate,OP_INSERT,USD,EUR,0.65
rate,OP_INSERT,GBP,USD,1.98
rate,OP_INSERT,USD,GBP,0.49
rate,OP_INSERT,EUR,GBP,0.74
```

```
____Order before: EUR, GBP, USD
__lbResult OP_INSERT ccy1="EUR" ccy2="GBP" ccy3="USD" rate1="0.74"
    rate2="1.98" rate3="0.65" looprate="0.95238"
rate,OP_INSERT,GBP,EUR,1.30
____Order before: GBP, EUR, USD
__lbResult OP_INSERT ccy1="EUR" ccy2="USD" ccy3="GBP" rate1="1.48"
    rate2="0.49" rate3="1.3" looprate="0.94276"
rate,OP_DELETE,EUR,USD,1.48
____Order before: EUR, USD, GBP
__lbResult OP_DELETE ccy1="EUR" ccy2="USD" ccy3="GBP" rate1="1.48"
    rate2="0.49" rate3="1.3" looprate="0.94276"
rate,OP_INSERT,EUR,USD,1.28
____Order before: EUR, USD, GBP
__lbResult OP_INSERT ccy1="EUR" ccy2="USD" ccy3="GBP" rate1="1.28"
    rate2="0.49" rate3="1.3" looprate="0.81536"
rate,OP_DELETE,USD,EUR,0.65
____Order before: USD, EUR, GBP
__lbResult OP_DELETE ccy1="EUR" ccy2="GBP" ccy3="USD" rate1="0.74"
    rate2="1.98" rate3="0.65" looprate="0.95238"
rate,OP_INSERT,USD,EUR,0.78
____Order before: USD, EUR, GBP
lbResult OP_INSERT ccy1="EUR" ccy2="GBP" ccy3="USD" rate1="0.74"
    rate2="1.98" rate3="0.78" looprate="1.142856"
rate,OP_DELETE,EUR,GBP,0.74
____Order before: EUR, GBP, USD
lbResult OP_DELETE ccy1="EUR" ccy2="GBP" ccy3="USD" rate1="0.74"
    rate2="1.98" rate3="0.78" looprate="1.142856"
rate,OP_INSERT,EUR,GBP,0.64
____Order before: EUR, GBP, USD
__lbResult OP_INSERT ccy1="EUR" ccy2="GBP" ccy3="USD" rate1="0.64"
    rate2="1.98" rate3="0.78" looprate="0.988416"
```

It's the same result as before, only without the triplicates. And you can see that the rotation logic works right. The manual self-joining has produced the result without triplicates, without an intermediate table, and for me writing and understanding its logic is much easier than with the “proper” joins. I'd say that the manual self-join is a winner in every respect.

An interesting thing is that this manual logic produces the same result independently of whether it's connected to the Output or Pre label of the table. Try changing it, it works the same. This is because the original row is taken directly from the input rowop, and never participates in the join again; it's never read from the table by any of the loops. If it were read again from the table by the loops, the table connection would matter. And the correct one would be fairly weird: the INSERT rowops would have to be processed coming from the Output label, the DELETE rowops coming from the Pre label.

This is because the row has to be in the table to be found. And for an INSERT the row gets there only after it goes through the table and comes out on the Output label. But for a DELETE the row would get already deleted from the table by that time. Instead it has to be handled before that, on the Pre label, when the table only prepares to delete it.

If you look at the version with JoinTwo, that's also how an inner self-join works. Since it's an inner join, both rows on both sides must be present to produce a result. An INSERT first arrives from the Pre label on the left side, doesn't find itself in the table, and produces no result (again, we're talking here about the situation when a row has to get joined to itself; it might well find the other pairs for itself and produce a result for them but not for itself joined with itself). Then it arrives the second time from the Output label on the right side. Now it looks in the table, and finds itself, and produces the result (an INSERT coming form the join). A DELETE also first arrives from the Pre label on the left side. It finds its copy in the table and produces the result (a DELETE coming from the join). When the second copy of the row arrives from the Output label on the right side, it doesn't find its copy in the table any more, and produces nothing. In the end it's the same thing, an INSERT comes out of the join triggered by the table Output label, a DELETE comes out of the join triggered by the table Pre label. It's not a whimsy, it's caused by the requirements of the correctness. The manual self-join would have to mimic this order to produce the correct result. In such a situation perhaps JoinTwo would be easier to use than doing things manually.

12.14. Self-join done with a LookupJoin

The experience with the manual join has made me think about using a similar approach to avoid triplication of the data in the version with join templates. And after some false-starts, I've realized that what that version needs is the LookupJoins. They replace the loops. So, one more version is:

```
our $join1 = Triceps::LookupJoin->new(
  name => "join1",
  leftFromLabel => $tRate->getOutputLabel(),
  leftFields => [ "ccy1", "ccy2", "rate/rate1" ],
  rightTable => $tRate,
  rightIdxPath => [ "byCcy1" ],
  rightFields => [ "ccy2/ccy3", "rate/rate2" ],
  byLeft => [ "ccy2/ccy1" ],
  isLeft => 0,
); # would die by itself on an error

our $join2 = Triceps::LookupJoin->new(
  name => "join2",
  leftFromLabel => $join1->getOutputLabel(),
  rightTable => $tRate,
  rightIdxPath => [ "byCcy1", "byCcy12" ],
  rightFields => [ "rate/rate3" ],
  byLeft => [ "ccy3/ccy1", "ccy1/ccy2" ],
  isLeft => 0,
); # would die by itself on an error

# now compute the resulting circular rate and filter the profitable loops
our $rtResult = Triceps::RowType->new(
  $join2->getResultRowType()->getdef(),
  looprate => "float64",
) or confess "$!";
my $lbResult = $uArb->makeDummyLabel($rtResult, "lbResult");
my $lbCompute = $uArb->makeLabel($join2->getResultRowType(), "lbCompute", undef, sub {
  my ($label, $rowop) = @_;
  my $row = $rowop->getRow();

  my $ccy1 = $row->get("ccy1");
  my $ccy2 = $row->get("ccy2");
  my $ccy3 = $row->get("ccy3");
  my $rate1 = $row->get("rate1");
  my $rate2 = $row->get("rate2");
  my $rate3 = $row->get("rate3");
  my $looprate = $rate1 * $rate2 * $rate3;

  # now build the row in normalized order of currencies
  print("____Order before: $ccy1, $ccy2, $ccy3\n");
  my $result;
  if ($ccy2 lt $ccy3) {
    if ($ccy2 lt $ccy1) { # rotate left
      $result = $lbResult->makeRowopHash($rowop->getOpcode(),
        ccy1 => $ccy2,
        ccy2 => $ccy3,
        ccy3 => $ccy1,
        rate1 => $rate2,
        rate2 => $rate3,
        rate3 => $rate1,
        looprate => $looprate,
      );
    }
  } else {
    if ($ccy3 lt $ccy1) { # rotate right
```

```
      $result = $lbResult->makeRowopHash($rowop->getOpcode(),
        ccy1 => $ccy3,
        ccy2 => $ccy1,
        ccy3 => $ccy2,
        rate1 => $rate3,
        rate2 => $rate1,
        rate3 => $rate2,
        looprate => $looprate,
      );
    }
  }
  if (!defined $result) { # use the straight order
    $result = $lbResult->makeRowopHash($rowop->getOpcode(),
      ccy1 => $ccy1,
      ccy2 => $ccy2,
      ccy3 => $ccy3,
      rate1 => $rate1,
      rate2 => $rate2,
      rate3 => $rate3,
      looprate => $looprate,
    );
  }
  if ($looprate > 1) {
    $uArb->call($result);
  } else {
    print("__", $result->printP(), "\n"); # for debugging
  }
}) or confess "$!";
$join2->getOutputLabel()->chain($lbCompute) or confess "$!";
```

It produces the exact same result as the version with the manual loops, with the only minor difference of the field order in the result rows.

And, in retrospect, I should have probably made a function for the row rotation, so that I would not have to copy that code here.

Well, it works the same as the version with the loops and maybe even looks a little bit neater, but in practice it's much harder to write, debug and understand. The caveat for the situation where the incoming row might participate in the join the second time applies to this version of the code as well. The same thing about the Pre and Output labels would have to be done, resulting in four LookupJoins instead of two. Each pair would become a manually-built analog of JoinTwo, and probably it's easier to use a JoinTwo to start with.

12.15. A glimpse inside JoinTwo and the hidden options of LookupJoin

The internals of JoinTwo provide an interesting example of a template that builds upon other template (LookupJoin). For a while JoinTwo was compact and straightforward, and easy to demonstrate. Then it has grown all these extra features, options and error checks, and became quite complicated. So I'll show only the selected portions of the JoinTwo constructor, with the gist of its functionality:

```
...
  my $selfJoin = $self->{leftTable}->same($self->{rightTable});
  if ($selfJoin && !defined $self->{leftFromLabel}) {
    # one side must be fed from Pre label (but still let the user override)
    $self->{leftFromLabel} = $self->{leftTable}->getPreLabel();
  }
...

  my ($leftLeft, $rightLeft);
  if ($self->{type} eq "inner") {
    $leftLeft = 0;
    $rightLeft = 0;
  } elsif ($self->{type} eq "left") {
    $leftLeft = 1;
    $rightLeft = 0;
  } elsif ($self->{type} eq "right") {
    $leftLeft = 0;
    $rightLeft = 1;
  } elsif ($self->{type} eq "outer") {
    $leftLeft = 1;
    $rightLeft = 1;
  } else {
    Carp::confess("Unknown value '" . $self->{type} . "' of option 'type', must be one of
 inner|left|right|outer");
  }

  $self->{leftRowType} = $self->{leftTable}->getRowType();
  $self->{rightRowType} = $self->{rightTable}->getRowType();
...

  for my $side ( ("left", "right") ) {
    if (defined $self->{"${side}FromLabel"}) {
...
    } else {
      $self->{"${side}FromLabel"} = $self->{"${side}Table"}->getOutputLabel();
    }

    my @keys;
    ($self->{"${side}IdxType"}, @keys) = $self->{"${side}Table"}->getType()-
>findIndexKeyPath(@{$self->{"${side}IdxPath"}});
    # would already confess if the index is not found

    if (!$self->{overrideSimpleMinded}) {
      if (!$self->{"${side}IdxType"}->isLeaf()

      && ($self->{type} ne "inner" && $self->{type} ne $side) ) {
        my $table = $self->{"${side}Table"};
        my $ixt = $self->{"${side}IdxType"};
        if ($selfJoin && $side eq "left") {
          # the special case, reading from the table's Pre label;
          # must adjust the count for what will happen after the row gets processed
          $self->{"${side}GroupSizeCode"} = sub { # (opcode, row)
            if (&Triceps::isInsert($_[0])) {
              $table->groupSizeIdx($ixt, $_[1])+1;
            } else {
              $table->groupSizeIdx($ixt, $_[1])-1;
            }
          };
        } else {
          $self->{"${side}GroupSizeCode"} = sub { # (opcode, row)
            $table->groupSizeIdx($ixt, $_[1]);
          };
        }
      }
    }

...
  my $fieldsMirrorKey = 1;
  my $uniq = $self->{fieldsUniqKey};
  if ($uniq eq "first") {
    $uniq = $self->{fieldsLeftFirst} ? "left" : "right";
  }
  if ($uniq eq "none") {
    $fieldsMirrorKey = 0;
  } elsif ($uniq eq "manual") {
    # nothing to do
  } elsif ($uniq =~ /^(left|right)$/) {
    my($side, @keys);
    if ($uniq eq "left") {
      $side = "right";
      @keys = @rightkeys;
    } else {
      $side = "left";
      @keys = @leftkeys;
    }
    if (!defined $self->{"${side}Fields"}) {
      $self->{"${side}Fields"} = [ ".*" ]; # the implicit pass-all
    }
    unshift(@{$self->{"${side}Fields"}}, map("!$_", @keys) );
  } else {
    Carp::confess("Unknown value '" . $self->{fieldsUniqKey} . "' of option
 'fieldsUniqKey', must be one of none|manual|left|right|first");
  }

  # now create the LookupJoins
  $self->{leftLookup} = Triceps::LookupJoin->new(
    unit => $self->{unit},
    name => $self->{name} . ".leftLookup",
    leftRowType => $self->{leftRowType},
    rightTable => $self->{rightTable},
    rightIdxPath => $self->{rightIdxPath},
    leftFields => $self->{leftFields},
    rightFields => $self->{rightFields},
    fieldsLeftFirst => $self->{fieldsLeftFirst},
    fieldsMirrorKey => $fieldsMirrorKey,
    by => \@leftby,
    isLeft => $leftLeft,
    automatic => 1,
    oppositeOuter => ($rightLeft && !$self->{overrideSimpleMinded}),
    groupSizeCode => $self->{leftGroupSizeCode},
    saveJoinerTo => $self->{leftSaveJoinerTo},
  );
  $self->{rightLookup} = Triceps::LookupJoin->new(
    unit => $self->{unit},
    name => $self->{name} . ".rightLookup",
    leftRowType => $self->{rightRowType},
    rightTable => $self->{leftTable},
    rightIdxPath => $self->{leftIdxPath},
    leftFields => $self->{rightFields},
    rightFields => $self->{leftFields},
    fieldsLeftFirst => !$self->{fieldsLeftFirst},
    fieldsMirrorKey => $fieldsMirrorKey,
    by => \@rightby,
    isLeft => $rightLeft,
    automatic => 1,
    oppositeOuter => ($leftLeft && !$self->{overrideSimpleMinded}),
    groupSizeCode => $self->{rightGroupSizeCode},
    saveJoinerTo => $self->{rightSaveJoinerTo},
  );

  # create the output label
  $self->{outputLabel} = $self->{unit}->makeDummyLabel($self->{leftLookup}-
>getResultRowType(), $self->{name} . ".out");
  Carp::confess("$!") unless (ref $self->{outputLabel} eq "Triceps::Label");

  # and connect them together
  $self->{leftFromLabel}->chain($self->{leftLookup}->getInputLabel());
  $self->{rightFromLabel}->chain($self->{rightLookup}->getInputLabel());
  $self->{leftLookup}->getOutputLabel()->chain($self->{outputLabel});
  $self->{rightLookup}->getOutputLabel()->chain($self->{outputLabel});
```

In the end it boils down to two LookupJoins, with the options computed from the JoinTwo's options. But you might notice that there are a few LookupJoin options that haven't been described before.

Despite the title of the section, these options aren't really hidden, just they aren't particularly useful unless you want to use a LookupJoin as a part of a multi-sided join, like JoinTwo does. It's even hard to explain what do they do without explaining the JoinTwo first. If you're not interested in such details, you can as well skip them.

So, setting

```
  oppositeOuter => 1,
```

tells that this LookupJoin is a part of an outer join, with the opposite side (right side, for this LookupJoin) being an outer one (well, this side might be outer too if `isLeft => 1`, but that's a whole separate question). This enables the logic that checks whether the row inserted here is the first one that matches a row in the right-side table, and whether the row deleted here was the last one that matches. If the condition is satisfied, not a simple INSERT or DELETE rowop is produced but a correct DELETE-INSERT pair that replaces the old state with the new one. It has been described in detail in Section 12.8: “JoinTwo joins two tables” (p. 185) .

But how does it know whether the current row if the first one or last one or neither? After all, LookupJoin doesn't have any access to the left-side table.

It has two ways to know. First, by default it simply assumes that it's an one-to-something (1:1 or 1:M) join. Then there may be no more than one matching row on this side, and every row inserted is the first one, and every row deleted is the last one. Then it does the DELETE-INSERT trick every time.

Second, the option

```
  groupSizeCode => \&groupSizeComputation,
```

can be used to compute the current group size for the current row. It provides a function that does the computation and gets called as

```
$gsz = &{$self->{groupSizeCode}}($opcode, $row);
```

Note that it doesn't get the table reference nor the index type reference as arguments, so it has to be a closure with the references compiled into it. JoinTwo does it with the definition

```
sub { # (opcode, row)
  $table->groupSizeIdx($ixt, $_[1]);
}
```

Why not just pass the table and index type references to JoinTwo and let it do the same computation without the mess of the closure references? Because the group size computation may need to be different. When the JoinTwo does a self-join, it feeds the left side from the table's Pre label, and the normal group size computation would be incorrect because the rowop didn't get applied to the table yet. Instead it has to predict what will happen when the rowop will get applied:

```
sub { # (opcode, row)
  if (&Triceps::isInsert($_[0])) {
    $table->groupSizeIdx($ixt, $_[1])+1;
  } else {
    $table->groupSizeIdx($ixt, $_[1])-1;
  }
}
```

If you set the option “groupSizeCode” to `undef`, that's the default value that triggers the one-to-something behavior.

The option

```
  fieldsMirrorKey => 1,
```

has been already described. It enables another magic behavior: mirroring the values of key fields to both sides before they are used to produce the result row. This is the heavy machinery that underlies the JoinTwo's high-level option “fieldsUniqKey”. But it hasn't been described yet that the mirroring goes both ways: If this is a left join and no matching row is found on the right, the values of the key fields will be copied from the left to the right. If the option “oppositeOuter” is set and causes a row with the empty left side to be produced as a part of DELETE-INSERT pair, the key fields will be copied from the right to the left.

Chapter 13. Time processing

13.1. Time-limited propagation

When aggregating data, often the results of the aggregation stay relevant longer than the original data.

For example, in the financials the data gets collected and aggregated for the current business day. After the day is closed, the day's detailed data are not interesting any more, and can be deleted in preparation for the next day. However the daily results stay interesting for a long time, and may even be archived for years.

This is not limited to the financials. A long time ago, in the times of slow and expensive Internet connections, I've done a traffic accounting system. It did the same: as the time went by, less and less detail was kept about the traffic usage. The modern accounting of the click-through advertisement also works in a similar way.

An easy way to achieve this result is to put a filter on the way of the aggregation results. It would compare the current idea of time and the time in the rows going by, and throw away the rows that are too old. This can be done as a label that gets the data from the aggregator and then forwards or doesn't forward the data to the real destination, and has been already shown. This solves the propagation problem but as the obsolete original data gets deleted, the aggregator will still be churning and producing the updates, only to have them thrown away at the filter. A more efficient way is to stop the churn by placing the filter right into the aggregator.

The next example demonstrates such an aggregator, in a simplified version of that traffic accounting system that I've once done. The example is actually about more than just stopping the data propagation. That stopping accounts for about three lines in it. But I also want to show a simple example of traffic accounting as such. And to show that the lack of the direct time support in Triceps does not stop you from doing any time-based processing. Because of this I'll show the whole example and not just snippets from it. But since the example is biggish, I'll paste it into the text in pieces with commentaries for each piece.

```
our $uTraffic = Triceps::Unit->new("uTraffic");

# one packet's header
our $rtPacket = Triceps::RowType->new(
  time => "int64", # packet's timestamp, microseconds
  local_ip => "string", # string to make easier to read
  remote_ip => "string", # string to make easier to read
  local_port => "int32",
  remote_port => "int32",
  bytes => "int32", # size of the packet
) or confess "$!";

# an hourly summary
our $rtHourly = Triceps::RowType->new(
  time => "int64", # hour's timestamp, microseconds
  local_ip => "string", # string to make easier to read
  remote_ip => "string", # string to make easier to read
  bytes => "int64", # bytes sent in an hour
) or confess "$!";
```

The router to the ISP forwards us the packet header information from all the packets that go though the outside link. The `local_ip` is always the address of a machine on our network, `remote_ip` outside our network, no matter in which direction the packet went. With a slow and expensive connection, we want to know two things: First, that the provider's billing at the end of the month is correct. Second, to be able to find out the high traffic users, when was the traffic used, and then maybe look at the remote addresses and decide whether that traffic was used for the business purposes or not. This example goes up to aggregation of the hourly summaries and then stops, since the further aggregation by days and months is straightforward to do.

If there is no traffic for a while, the router is expected to periodically communicate its changing idea of time as the same kind of records but with the non-timestamp fields as NULLs. That by the way is the right way to communicate the time-based information between two machines: do not rely on any local synchronization and timeouts but have the master send the periodic time updates to the slave even if it has no data to send. The logic is then driven by the time reported by the master. A nice side effect is that the logic can also easily be replayed later, using these timestamps and without any concern of the real time. If there are multiple masters, the slave would have to order the data coming from them according to the timestamps, thus synchronizing them together.

The hourly data drops the port information, and sums up the traffic between two addresses in the hour. It still has the timestamp but now this timestamp is rounded to the start of the hour:

```
# compute an hour-rounded timestamp
sub hourStamp # (time)
{
  return $_[0]  - ($_[0] % (1000*1000*3600));
}
```

Next, to the aggregation. The SimpleAggregator has no provision for filtering in it, the aggregation has to be done raw.

```
# the current hour stamp that keeps being updated
our $currentHour;

# aggregation handler: recalculate the summary for the last hour
sub computeHourly # (table, context, aggop, opcode, rh, state, args...)
{
  my ($table, $context, $aggop, $opcode, $rh, $state, @args) = @_;
  our $currentHour;

  # don't send the NULL record after the group becomes empty
  return if ($context->groupSize()==0
    || $opcode == &Triceps::OP_NOP);

  my $rhFirst = $context->begin();
  my $rFirst = $rhFirst->getRow();
  my $hourstamp = &hourStamp($rFirst->get("time"));

  return if ($hourstamp < $currentHour);

  if ($opcode == &Triceps::OP_DELETE) {
    $context->send($opcode, $$state) or confess "$!";
    return;
  }

  my $bytes = 0;
  for (my $rhi = $rhFirst; !$rhi->isNull();
      $rhi = $context->next($rhi)) {
    $bytes += $rhi->getRow()->get("bytes");
  }

  my $res = $context->resultType()->makeRowHash(
    time => $hourstamp,
    local_ip => $rFirst->get("local_ip"),
    remote_ip => $rFirst->get("remote_ip"),
    bytes => $bytes,
  ) or confess "$!";
  ${$state} = $res;
  $context->send($opcode, $res) or confess "$!";
}

sub initHourly #  (@args)
```

```
{
  my $refvar;
  return \$refvar;
}
```

The aggregation doesn't try to optimize by being additive, to keep the example simpler. The model keeps the notion of the current hour. As soon as the hour stops being current, the aggregation for it stops. The result of that aggregation will then be kept unchanged in the hourly result table, no matter what happens to the original data.

The tables are defined and connected thusly:

```
# the full stats for the recent time
our $ttPackets = Triceps::TableType->new($rtPacket)
  ->addSubIndex("byHour",
    Triceps::IndexType->newPerlSorted("byHour", undef, sub {
      return &hourStamp($_[0]->get("time")) <=> &hourStamp($_[1]->get("time"));
    })
    ->addSubIndex("byIP",
      Triceps::IndexType->newHashed(key => [ "local_ip", "remote_ip" ])
      ->addSubIndex("group",
        Triceps::IndexType->newFifo()
        ->setAggregator(Triceps::AggregatorType->new(
          $rtHourly, "aggrHourly", \&initHourly, \&computeHourly)
        )
      )
    )
  )
or confess "$!";

$ttPackets->initialize() or confess "$!";
our $tPackets = $uTraffic->makeTable($ttPackets,
  &Triceps::EM_CALL, "tPackets") or confess "$!";

# the aggregated hourly stats, kept longer
our $ttHourly = Triceps::TableType->new($rtHourly)
  ->addSubIndex("byAggr",
    Triceps::SimpleOrderedIndex->new(
      time => "ASC", local_ip => "ASC", remote_ip => "ASC")
  )
or confess "$!";

$ttHourly->initialize() or confess "$!";
our $tHourly = $uTraffic->makeTable($ttHourly,
  &Triceps::EM_CALL, "tHourly") or confess "$!";

# connect the tables
$tPackets->getAggregatorLabel("aggrHourly")->chain($tHourly->getInputLabel())
  or confess "$!";
```

The table of incoming packets has a 3-level index: it starts with being sorted by the hour part of the timestamp, then goes by the ip addresses to complete the aggregation key, and then a FIFO for each aggregation group. Arguably, maybe it would have been better to include the ip addresses straight into the top-level sorting index, I don't know, and it doesn't seem worth measuring. The top-level ordering by the hour is important, it will be used to delete the rows that have become old.

The table of hourly aggregated stats uses the same kind of index, only now there is no need for a FIFO because there is only one row per this key. And the timestamp is already rounded to the hour right in the rows, so a SimpleOrderedIndex can be used without writing a manual comparison function, and the ip fields have been merged into it too.

The output of the aggregator on the packets table is connected to the input of the hourly table.

```
# label to print the changes to the detailed stats
```

```
makePrintLabel("lbPrintPackets", $tPackets->getOutputLabel());
# label to print the changes to the hourly stats
makePrintLabel("lbPrintHourly", $tHourly->getOutputLabel());

# dump a table's contents
sub dumpTable # ($table)
{
  my $table = shift;
  for (my $rhit = $table->begin(); !$rhit->isNull(); $rhit = $rhit->next()) {
    print($rhit->getRow()->printP(), "\n");
  }
}

# how long to keep the detailed data, hours
our $keepHours = 2;

# flush the data older than $keepHours from $tPackets
sub flushOldPackets
{
  my $earliest = $currentHour - $keepHours * (1000*1000*3600);
  my $next;
  # the default iteration of $tPackets goes in the hour stamp order
  for (my $rhit = $tPackets->begin(); !$rhit->isNull(); $rhit = $next) {
    last if (&hourStamp($rhit->getRow()->get("time")) >= $earliest);
    $next = $rhit->next(); # advance before removal
    $tPackets->remove($rhit);
  }
}
```

The print labels generate the debugging output that shows what is going on with both tables. Next go a couple of helper functions.

The `dumpTable()` is a straightforward iteration through a table and print. It can be used on any table, `printP()` takes care of any differences.

The flushing goes through the packets table and deletes the rows that belong to an older hour than the current one or `$keepHours` before it. For this to work right, the rows must go in the order of the hour stamps, which the outer index “byHour” takes care of.

All the time-related logic expects that the time never goes backwards. This is a simplification to make the example shorter, a production code can not assume this.

```
while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "new") {
    my $rowop = $tPackets->getInputLabel()->makeRowopArray(@data);
    # update the current notion of time (simplistic)
    $currentHour = &hourStamp($rowop->getRow()->get("time"));
    if (defined($rowop->getRow()->get("local_ip"))) {
      $uTraffic->call($rowop) or confess "$!";
    }
    &flushOldPackets(); # flush the packets
    $uTraffic->drainFrame(); # just in case, for completeness
  } elsif ($type eq "dumpPackets") {
    &dumpTable($tPackets);
  } elsif ($type eq "dumpHourly") {
    &dumpTable($tHourly);
  }
}
```

The final part is the main loop. The input comes in the CSV form as a command followed by more data. If the command is “new” then the data is the opcode and data fields, as it would be sent by the router. The commands “dumpPackets” and “dumpHourly” are used to print the contents of the tables, to see, what is going on in them.

In an honest implementation there would be a separate label that would differentiate between a reported packet and just a time update from the router. Here for simplicity this logic is placed right into the main loop. On each input record it updates the model's idea of the current timestamp, then if there is a packet data, it gets processed, and finally the rows that have become too old for the new timestamp get flushed.

Now a run of the model. Its printout is also broken up into the separately commented pieces. Of course, it's not like a real run, it just contains one or two packets per hour to show how things work.

```
new,OP_INSERT,1330886011000000,1.2.3.4,5.6.7.8,2000,80,100
tPackets.out OP_INSERT time="1330886011000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="100"
tHourly.out OP_INSERT time="1330884000000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="100"
new,OP_INSERT,1330886012000000,1.2.3.4,5.6.7.8,2000,80,50
tHourly.out OP_DELETE time="1330884000000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="100"
tPackets.out OP_INSERT time="1330886012000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="50"
tHourly.out OP_INSERT time="1330884000000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
```

The two input rows in the first hour refer to the same connection, so they go into the same group and get aggregated together in the hourly table. The rows for the current hour in the hourly table get updated immediately as more data comes in. The `tHourly.out OP_DELETE` comes out even before `tPackets.out OP_INSERT` because it's driven by the output of the aggregator on `$tPackets`, and the operation `AO_BEFORE_MOD` on the aggregator that drives the deletion is executed before `$tPackets` gets modified.

```
new,OP_INSERT,1330889811000000,1.2.3.4,5.6.7.8,2000,80,300
tPackets.out OP_INSERT time="1330889811000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="300"
tHourly.out OP_INSERT time="1330887600000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="300"
```

Only one packet arrives in the next hour.

```
new,OP_INSERT,1330894211000000,1.2.3.5,5.6.7.9,3000,80,200
tPackets.out OP_INSERT time="1330894211000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" local_port="3000" remote_port="80" bytes="200"
tHourly.out OP_INSERT time="1330891200000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" bytes="200"
new,OP_INSERT,1330894211000000,1.2.3.4,5.6.7.8,2000,80,500
tPackets.out OP_INSERT time="1330894211000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="500"
tHourly.out OP_INSERT time="1330891200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="500"
```

And two more packets in the next hour. They are for the different connections, so they do not get summed together in the aggregation. When the hour changes again, the old data will start being deleted (because of `$keepHours = 2`, which ends up keeping the current hour and two before it), so let's take a snapshot of the tables' contents.

```
dumpPackets
time="1330886011000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="100"
time="1330886012000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="50"
time="1330889811000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="300"
```

```
time="1330894211000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="500"
time="1330894211000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    local_port="3000" remote_port="80" bytes="200"
dumpHourly
time="1330884000000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="150"
time="1330887600000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="300"
time="1330891200000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="500"
time="1330891200000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    bytes="200"
```

The packets table shows all the 5 packets received so far, and the hourly aggregation results for all 3 hours (with two separate aggregation groups in the same last hour, for different ip pairs).

```
new,OP_INSERT,1330896811000000,1.2.3.5,5.6.7.9,3000,80,10
tPackets.out OP_INSERT time="1330896811000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" local_port="3000" remote_port="80" bytes="10"
tHourly.out OP_INSERT time="1330894800000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" bytes="10"
tPackets.out OP_DELETE time="1330886011000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="100"
tPackets.out OP_DELETE time="1330886012000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="50"
```

When the next hour's packet arrives, it gets processed as usual, but then the removal logic finds the packet rows that have become too old to keep. It kicks in and deletes them. But notice that the deletions affect only the packets table, the aggregator ignores this activity as too old and does not propagate it to the hourly table.

```
new,OP_INSERT,1330900411000000,1.2.3.4,5.6.7.8,2000,80,40
tPackets.out OP_INSERT time="1330900411000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="40"
tHourly.out OP_INSERT time="1330898400000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="40"
tPackets.out OP_DELETE time="1330889811000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="300"
```

One more hour's packet, flushes out the data for another hour.

```
new,OP_INSERT,1330904011000000
tPackets.out OP_DELETE time="1330894211000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="500"
tPackets.out OP_DELETE time="1330894211000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" local_port="3000" remote_port="80" bytes="200"
```

And just a time update for another hour, when no packets have been received. The removal logic still kicks in and works the same way, deleting raw data for one more hour. After all this activity let's dump the tables again:

```
dumpPackets
time="1330896811000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    local_port="3000" remote_port="80" bytes="10"
time="1330900411000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="40"
dumpHourly
time="1330884000000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="150"
time="1330887600000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="300"
time="1330891200000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="500"
```

```
time="1330891200000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    bytes="200"
time="1330894800000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    bytes="10"
time="1330898400000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="40"
```

The packets table only has the data for the last 3 hours (there are no rows for the last hour because none have arrived). But the hourly table contains all the history. The rows weren't getting deleted here.

13.2. Periodic updates

In the previous example if we keep aggregating the data from hours to days and the days to months, then the arrival of each new packet will update the whole chain. Sometimes that's what we want, sometimes it isn't. The daily stats might be fed into some complicated computation, with nobody looking at the results until the next day. In this situation each packet will trigger these complicated computations, for no good reason, since nobody cares for them until the day is closed.

These unnecessary computations can be prevented by disconnecting the daily data from the hourly data, and performing the manual aggregation only when the day changes. Then these complicated computations would happen only once a day, not many times per second.

Here is how the last example gets amended to produce the once-a-day daily summaries of all the traffic (as before, in multiple snippets, this time showing only the added or changed code):

```
# an hourly summary, now with the day extracted
our $rtHourly = Triceps::RowType->new(
  time => "int64", # hour's timestamp, microseconds
  day => "string", # in YYYYMMDD
  local_ip => "string", # string to make easier to read
  remote_ip => "string", # string to make easier to read
  bytes => "int64", # bytes sent in an hour
) or confess "$!";

# a daily summary: just all traffic for that day
our $rtDaily = Triceps::RowType->new(
  day => "string", # in YYYYMMDD
  bytes => "int64", # bytes sent in an hour
) or confess "$!";
```

The hourly rows get an extra field, for convenient aggregation by day. And the daily rows are introduced. The notion of the day is calculated as:

```
# compute the date of a timestamp, a string YYYYMMDD
sub dateStamp # (time)
{
  my @ts = gmtime($_[0]/1000000); # microseconds to seconds
  return sprintf("%04d%02d%02d", $ts[5]+1900, $ts[4]+1, $ts[3]);
}

# the current hour stamp that keeps being updated
our $currentHour = undef;
# the current day stamp that keeps being updated
our $currentDay = undef;
```

The calculation is done in GMT, so that the code produces the same result all around the world. If you're doing this kind of project for real, you may want to use the local time zone instead (but be careful with the changing daylight saving time).

And the model keeps a global notion of the current day in addition to the current hour.

```
# aggregation handler: recalculate the summary for the last hour
```

```
sub computeHourlywDay # (table, context, aggop, opcode, rh, state, args...)
{
...
  my $res = $context->resultType()->makeRowHash(
    time => $hourstamp,
    day => &dateStamp($hourstamp),
    local_ip => $rFirst->get("local_ip"),
    remote_ip => $rFirst->get("remote_ip"),
    bytes => $bytes,
  ) or confess "$!";
  ${$state} = $res;
  $context->send($opcode, $res) or confess "$!";
}
```

The packets-to-hour aggregation function now populates this extra field, the rest of it stays the same.

```
# the aggregated hourly stats, kept longer
our $ttHourly = Triceps::TableType->new($rtHourly)
  ->addSubIndex("byAggr",
    Triceps::SimpleOrderedIndex->new(
      time => "ASC", local_ip => "ASC", remote_ip => "ASC")
  )
  ->addSubIndex("byDay",
    Triceps::IndexType->newHashed(key => [ "day" ])
    ->addSubIndex("group",
      Triceps::IndexType->newFifo()
    )
  )
or confess "$!";

$ttHourly->initialize() or confess "$!";
our $tHourly = $uTraffic->makeTable($ttHourly,
  &Triceps::EM_CALL, "tHourly") or confess "$!";

# remember the daily secondary index type
our $idxHourlyByDay = $ttHourly->findSubIndex("byDay")
  or confess "$!";
our $idxHourlyByDayGroup = $idxHourlyByDay->findSubIndex("group")
  or confess "$!";
```

The hourly table type grows an extra secondary index for the manuall aggregation into the daily data.

```
# the aggregated daily stats, kept even longer
our $ttDaily = Triceps::TableType->new($rtDaily)
  ->addSubIndex("byDay",
    Triceps::IndexType->newHashed(key => [ "day" ])
  )
or confess "$!";

$ttDaily->initialize() or confess "$!";
our $tDaily = $uTraffic->makeTable($ttDaily,
  &Triceps::EM_CALL, "tDaily") or confess "$!";

# label to print the changes to the daily stats
makePrintLabel("lbPrintDaily", $tDaily->getOutputLabel());
```

And a table for the daily data is created but not connected to any other tables.

Instead it gets updated manually with the function that performs the manual aggregation of the hourly data:

```
sub computeDay # ($dateStamp)
{
  our $uTraffic;
  my $bytes = 0;

  my $rhFirst = $tHourly->findIdxBy($idxHourlyByDay, day => $_[0]);
  my $rhEnd = $rhFirst->nextGroupIdx($idxHourlyByDayGroup)
    or confess "$!";
  for (my $rhi = $rhFirst;
      !$rhi->same($rhEnd); $rhi = $rhi->nextIdx($idxHourlyByDay)) {
    $bytes += $rhi->getRow()->get("bytes");
  }
  $uTraffic->makeHashCall($tDaily->getInputLabel(), "OP_INSERT",
    day => $_[0],
    bytes => $bytes,
  );
}
```

This logic doesn't check whether any data for that day existed. If none did, it would just produce a row with traffic of 0 bytes anyway. This is different from the normal aggregation but here may actually be desirable: it shows for sure that yes, the aggregation for that day really did happen.

```
while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "new") {
    my $rowop = $tPackets->getInputLabel()->makeRowopArray(@data);
    # update the current notion of time (simplistic)
    $currentHour = &hourStamp($rowop->getRow()->get("time"));
    my $lastDay = $currentDay;
    $currentDay = &dateStamp($currentHour);
    if (defined($rowop->getRow()->get("local_ip"))) {
      $uTraffic->call($rowop) or confess "$!";
    }
    &flushOldPackets(); # flush the packets
    if (defined $lastDay && $lastDay ne $currentDay) {
      &computeDay($lastDay); # manual aggregation
    }
    $uTraffic->drainFrame(); # just in case, for completeness
  } elsif ($type eq "dumpPackets") {
    &dumpTable($tPackets);
  } elsif ($type eq "dumpHourly") {
    &dumpTable($tHourly);
  } elsif ($type eq "dumpDaily") {
    &dumpTable($tDaily);
  }
}
```

The main loop gets extended with the day-keeping logic and with the extra command to dump the daily data. It now maintains the current day, and after the packet computation is done, looks, whether the day has changed. If it did, it calls the manual aggregation of the last day.

And here is an example of its work:

```
new,OP_INSERT,1330886011000000,1.2.3.4,5.6.7.8,2000,80,100
tPackets.out OP_INSERT time="1330886011000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="100"
tHourly.out OP_INSERT time="1330884000000000" day="20120304"
    local_ip="1.2.3.4" remote_ip="5.6.7.8" bytes="100"
new,OP_INSERT,1330886012000000,1.2.3.4,5.6.7.8,2000,80,50
tHourly.out OP_DELETE time="1330884000000000" day="20120304"
    local_ip="1.2.3.4" remote_ip="5.6.7.8" bytes="100"
```

```
tPackets.out OP_INSERT time="1330886012000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="50"
tHourly.out OP_INSERT time="1330884000000000" day="20120304"
    local_ip="1.2.3.4" remote_ip="5.6.7.8" bytes="150"
new,OP_INSERT,1330889811000000,1.2.3.4,5.6.7.8,2000,80,300
tPackets.out OP_INSERT time="1330889811000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="300"
tHourly.out OP_INSERT time="1330887600000000" day="20120304"
    local_ip="1.2.3.4" remote_ip="5.6.7.8" bytes="300"
```

So far all the 3 packets are for the same day, and nothing new has happened.

```
new,OP_INSERT,1330972411000000,1.2.3.5,5.6.7.9,3000,80,200
tPackets.out OP_INSERT time="1330972411000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" local_port="3000" remote_port="80" bytes="200"
tHourly.out OP_INSERT time="1330970400000000" day="20120305"
    local_ip="1.2.3.5" remote_ip="5.6.7.9" bytes="200"
tPackets.out OP_DELETE time="1330886011000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="100"
tPackets.out OP_DELETE time="1330886012000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="50"
tPackets.out OP_DELETE time="1330889811000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="300"
tDaily.out OP_INSERT day="20120304" bytes="450"
```

When a packet for the next day arrives, it has three effects:

1. inserts the packet data as usual,

2. finds that the previous packet data is obsolete and flushes it (without upsetting the hourly summaries), and

3. finds that the day has changed and performs the manual aggregation of last day's hourly data into daily.

```
new,OP_INSERT,1331058811000000
tPackets.out OP_DELETE time="1330972411000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" local_port="3000" remote_port="80" bytes="200"
tDaily.out OP_INSERT day="20120305" bytes="200"
```

A time update for the yet next day flushes out the previous day's detailed packets and again builds the daily summary of that day.

```
new,OP_INSERT,1331145211000000
tDaily.out OP_INSERT day="20120306" bytes="0"
```

Yet another day's time roll now has no old data to delete (since none arrived in the previous day) but still produces the daily summary of 0 bytes.

```
dumpDaily
day="20120305" bytes="200"
day="20120304" bytes="450"
day="20120306" bytes="0"
```

This shows the eventual contents of the daily summaries. The order of the rows is fairly random, because of the hashed index. Note that the hourly summaries weren't flushed either, they are all still there too. If you want them eventually deleted after some time, you would need to provide more of the manual logic for that.

13.3. The general issues of time processing

After a couple of examples, it's time to do some generalizations. What these examples did manually, with the data expiration by time, the more mature CEP systems do internally, using the statements for the time-based work.

Which isn't always better though. The typical issues are with:

- fast replay of data,
- order of execution,
- synchronization between modules.

The problem with the fast replay is that those time based-statements use the real time and not the timestamps from the incoming rows. Sure, in Coral8 you can use the incoming row timestamps but they still are expected to have the time generally synchronized with the local clock (they are an attempt to solve the inter-module synchronization problem, not fast replay). You can't run them fast. And considering the Coral8 fashion of dropping the data when the input buffer overflows, you don't want to feed the data into it too fast to start with. In the Aleri system you can accelerate the time but it's by a fixed factor. You can run the logical time there say 10 times faster and feed the data 10 times faster but there are no timestamps in the input rows, and you simply can't feed the data precisely enough to reproduce the exact timing. And 10 times faster is not the same thing as just as fast as possible. I don't know for sure what the StreamBase does, it seems to have the time acceleration by a fixed rate too. Esper apparently allows the full control over timing, but I don't know much about it.

Your typical problem with fast replay in Coral8/CCL is this: you create a time limited window

```
create window ... keep 3 hours;
```

and then feed the data for a couple of days in say 20 minutes. Provided that you don't feed it too fast and none of it gets dropped, all of the data ends up in the window and none of it expires, since the window goes by the physical time, and the physical time was only 20 minutes. The first issue is that you may not have enough memory to store the data for two days, and everything would run out of memory and crash. The second issue is that if you want to do some time-based aggregation relying on the window expiration, you're out of luck.

Why would you want to feed the data so fast in the first place? Two reasons:

1. Testing. When you test your time-based logic, you don't want your unit test to take 3 hours, let alone multiple days. You also want your unit tests to be fully repeatable, without any fuzz.
2. State restoration after a planned shutdown or crash. No matter what everyone says, the built-in persistence features work right only for a small subset of the simple models. Getting the persistence work for the more complex models is difficult, and for all I know nobody has bothered to get it working right. The best approach in reality is to preserve a subset of the state, and get the rest of it by replaying the recent input data after restart. The faster you re-feed the data, the faster your model comes back online. (Incidentally, that's what Aleri does with the “persistent source streams”, only losing all the timing information of the rows and having the same above-mentioned issue as CCL).

Next issue, the execution order. The last example was relying on `$currentHour` being updated before `flushOldPackets()` runs. Otherwise the deletions would propagate through the aggregator where they should not. In a system like Aleri with each element running in its own thread there is no way to ensure any particular timing between the threads. In a system with single-threaded logic, like Coral8/Sybase or StreamBase, there is a way. But getting the order right is tricky. It depends on what the compiler and scheduler decide, and may require a few attempts to get the order right. Well, technically, Aleri can control the time too: you can run in artificial time, setting and stopping it. So you can stop the time, set to record timestamp, feed the record, wait for processing to complete, advance time, wait for any time-based processing to complete, and so on. I'm not sure if it made to Sybase R5, but it definitely worked on Aleri. However there was no tool that did it for you easily, and also all these synchronous calls present a pretty high overhead.

The procedural execution makes things much more straightforward.

Now, the synchronization between modules. When the data is passed between multiple threads or processes, there is always a jigger in the way the data goes through the inter-process communications and even more so through the network. Relying on the timing of the data after it arrives is usually a bad idea if you want to get any repeatability and precision. Instead the data has to be timestamped by the sender and then these timestamps used by the receiver instead of the real time.

And Coral8 allows you to do so. But what if there is no data coming? What do you do with the time-based processing? The Coral8 approach is to allow some delay and then proceed at the rate of the local clock. Note that the logical time is not exactly the same as the local clock, it generally gets behind the local clock by no more than the delay amount, or might go faster if the sender's clock goes faster. The question is, what delay amount do you choose? If you make it too short, the small hiccups in the data flow throw the timing off, the local clock runs ahead, and then the incoming data gets thrown away because it's too old. If you make it too long, you potentially add a large amount of latency. As it turns out, no reasonable amount of delay works well with Coral8. To get things working at least sort of reliably, you need horrendous delays, on the order of 10 seconds or more. Even then the sender may get hit by a long-running request and the connection would go haywire anyway.

The only reliable solution is to drive the time completely by the sender. Even if there is no data to send, it must still send the periodic time updates, and the receiver must use the incoming timestamps for its time-based processing. Sending one or even ten time-update packets per second is not a whole lot of overhead, and sure works much better than the 10-second delays. And along the way it gives the perfect repeatability and fast replay for the unit testing. So unless your CEP system can be controlled in this way, getting any decent distributed timing control requires doing it manually. The reality is that Aleri can't, Coral8 can't, the Sybase R4/R5 descended from them can't, and I could not find anything related to the time control in the StreamBase documentation, so my guess is that it can't either.

And if you have to control the time-based processing manually, doing it in the procedural way is at least easier.

An interesting side subject is the relation of the logical time to the real time. If the input data arrives faster than the CEP model can process it, the logical time will be getting behind the real time. Or if the data is fed at the artificially accelerated rate, the logical time will be getting ahead of the real time. There could even be a combination thereof: making the "real" time also artificial (driven by the sender) and artificially make the data get behind it for the testing purposes. The getting-behind can be detected and used to change the algorithm. For example, if we aggregate the traffic data in multiple stages, to the hour, to the day and to the month, the whole chain does not have to be updated on every packet Just update the first level on every packet, and then propagate further when the traffic burst subsides and gives the model a breather.

So far the major CEP systems don't seem to have a whole lot of direct support for it. There are ways to reduce the load by reducing the update frequency to a fixed period (like the `OUTPUT EVERY` statement in CCL, or periodic subscription in Aleri), but not much of the load-based kind. If the system provides ways to get both the real time and logical time of the row, the logic can be implemented manually. But the optimizations of the time-reading, like in Coral8, might make it unstable.

The way to do it in Triceps is by handling it in the Perl (or C++) code of the main event loop. When it has no data to read, it can create an “idle” row that would push through the results as a more efficient batch.

Chapter 14. The other templates and solutions

14.1. The dreaded diamond

The “diamond” is a particular topology of the data flow, when the computation separates based on some condition and then merges again. Like in Figure 14.1 . It is also known as “fork-join” (the “join” here has nothing to do with the SQL join, it just means that the arrows merge to the same block).

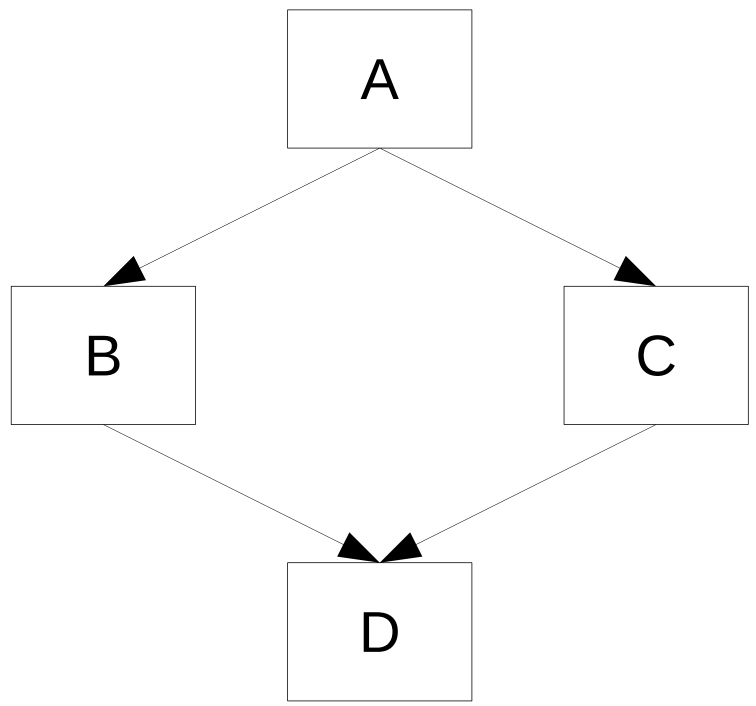

Figure 14.1. The diamond topology.

This topology is a known source of two problems. The first problem is about the execution order. To make things easier to see, let's consider a simple example. Suppose the rows come into the block A with the schema:

```
key => string,
value => int32,
```

And come out of the blocks B and C into D with schema

```
key => string,
value => int32,
negative => int32,
```

With the logic in the blocks being:

```
A:
  if value < 0 then B else C
B:
  negative = 1
C:
  negative = 0
```

Yes, this is a very dumb example that can usually be handled by a conditional expression in a single block. But that's to keep it small and simple. A real example would often include some SQL joins, with different joins done on condition.

Suppose A then gets the input, in CSV form:

```
INSERT,key1,10
DELETE,key1,10
INSERT,key1,20
DELETE,key1,20
INSERT,key1,-1
```

What arrives at D should be

```
INSERT,key1,10,0
DELETE,key1,10,0
INSERT,key1,20,0
DELETE,key1,20,0
INSERT,key1,-1,1
```

And with the first four rows this is not a problem: they follow the same path and are queued sequentially, so the order is preserved. But the last row follows a different path. And the last two rows logically represent a single update and would likely arrive closely together. The last row might happen to overtake the one before it, and D would see the incorrect result:

```
INSERT,key1,10,0
DELETE,key1,10,0
INSERT,key1,20,0
INSERT,key1,-1,1
DELETE,key1,20,0
```

If all these input rows arrive closely one after another, the last row might overtake even more of them and produce an even more disturbing result like

```
INSERT,key1,-1,1
INSERT,key1,10,0
DELETE,key1,10,0
INSERT,key1,20,0
DELETE,key1,20,0
```

Such misorderings may also happen between the rows with different keys. Those are usually less of a problem, because usually if D keeps a table, the rows with different keys may be updated in any order without losing the meaning. But in case if D keeps a FIFO index (say, for a window based on a row count), and the two keys fall into the same FIFO bucket, their misordering would also affect the logic.

The reasons for this can be subdivided further into two classes:

- asynchronous execution,

- incorrect scheduling in the synchronous execution.

If each block executes asynchronously in its own thread, there is no way to predict, in which order they will actually execute. If some data is sent to B and C at about the same time, it becomes a race between them. One of the paths might also be longer than the other, making one alternative always win the race. This kind of problems is fairly common for the Aleri system that is highly multithreaded. But this is the problem of absolutely any CEP engine if you split the execution by multiple threads or processes.

But the single-threaded execution is not necessarily a cure either. Then the order of execution is up to the scheduler. And if the scheduler gets all these rows close together, and then decides to process all the input of A, then all the input of B, of C and of D, then D will receive the rows in the order:

```
INSERT,key1,-1,1
INSERT,key1,10,0
DELETE,key1,10,0
INSERT,key1,20,0
DELETE,key1,20,0
```

Which is typical for, say, Coral8 if all the input rows arrive in a single bundle (see also the Section 7.2: “No bundling” (p. 39)).

At the moment Triceps does not directly support the multithreaded execution, so that renders the first sub-case moot for now. But the multithreading will be added soon, and then I'll return to this aspect.

When the single-threaded scheduling is concerned, Triceps provides two answers.

First, the conditional logic can often be expressed procedurally:

```
if ($a->get("value") < 0) {
  D($rtD->makeRowHash($a->toHash(), negative => 1));
} else {
  D($rtD->makeRowHash($a->toHash(), negative => 0));
}
```

The procedural if-else logic can easily handle not only the simple expressions but things like look-ups and modifications in the tables.

Second, if the logic is broken into the separate labels, the label call semantics provides the same ordering as well:

```
$lbA = $unit->makeLabel($rtA, "A", undef, sub {
  my $rop = $_[1];
  my $op = $rop->getOpcode(); my $a = $rop->getRow();
  if ($a->get("value") < 0) {
    $unit->call($lbB->makeRowop($op, $a));
  } else {
    $unit->call($lbC->makeRowop($op, $a));
  }
}) or die "$!";

$lbB = $unit->makeLabel($rtA, "B", undef, sub {
  my $rop = $_[1];
  my $op = $rop->getOpcode(); my $a = $rop->getRow();
  $unit->makeHashCall($lbD, $op, $a->toHash(), negative => 1)
    or die "$!";
}) or die "$!";

$lbC = $unit->makeLabel($rtA, "C", undef, sub {
  my $rop = $_[1];
  my $op = $rop->getOpcode(); my $a = $rop->getRow();
  $unit->makeHashCall($lbD, $op, $a->toHash(), negative => 0)
```

```
    or die "$!";
}) or die "$!";
```

When the label A calls the label B or C, which calls the label D, A does not get to see its next input row until the whole chain of calls to D and beyond completes. B and C may be replaced with the label chains of arbitrary complexity, including loops, without disturbing the logic.

The second problem with the diamond topology happens when the blocks B and C keep the state, and the input data gets updated by simply re-sending a record with the same key. This kind of updates is typical for the systems that do not have the concept of opcodes.

Consider a CCL example (approximate, since I can't test it) that gets the reports about borrowing and loaning securities, using the sign of the quantity to differentiate between borrows (-) and loans (+). It then sums up the borrows and loans separately:

```
create schema s_A (
  id integer,
  symbol string,
  quantity long
);
create input stream i_A schema s_A;

create schema s_D (
  symbol string,
  borrowed boolean, // flag: loaned or borrowed
  quantity long
);
// aggregated data
create public window w_D schema s_D
keep last per symbol, borrowed;

// collection of borrows
create public window w_B schema s_A keep last per id;
// collection of loans
create public window w_C schema s_A keep last per id;

insert when quantity < 0
  then w_B
  else w_C
select * from i_A;

// borrows aggregation
insert into w_D
select
  symbol,
  true,
  sum(quantity)
group by symbol
from w_B;

// loans aggregation
insert into w_D
select
  symbol,
  false,
  sum(quantity)
group by symbol
from w_C;
```

It works OK until a row with the same id gets updated to a different sign of quantity:

```
1,AAA,100
....
1,AAA,-100
```

If the quantity kept the same sign, the new row would simply replace the old one in w_B or w_C, and the aggregation result would be right again. But when the sign changes, the new row goes into a different direction than the previous one. Now it ends up with both w_B and w_C having rows with the same id: one old and one new!

In this case really the problem is at the “fork” part of the “diamond”, the merging part of it is just along for the ride, carrying the incorrect results.

This problem does not happen in the systems that have both inserts and deletes. Then the data sequence becomes

```
INSERT,1,AAA,100
....
DELETE,1,AAA,100
INSERT,1,AAA,-100
```

The DELETE goes along the same branch as the first insert and undoes its effect, then the second INSERT goes into the other branch.

Since Triceps has both INSERT and DELETE opcodes, it's immune to this problem, as long as the input data has the correct DELETEs in it.

If you wonder, the CCL example can be fixed too but in a more round-about way, by adding a couple of statements before the “insert-when” statement:

```
on w_A
delete from w_B
  where w_A.id = w_B.id;

on w_A
delete from w_C
  where w_A.id = w_C.id;
```

This generates the matching DELETEs. Of course, if you want, you can use this way with Triceps too.

14.2. Collapsed updates

First, a note: the collapse described here has nothing to do with the collapsing of the aggregation groups. It's just the same word reused for a different purpose.

Sometimes the exact sequence of how a row at a particular key was updated does not matter, the only interesting part is the end result. Like the `OUTPUT EVERY` statement in CCL or the pulsed subscription in Aleri. It doesn't have to be time-driven either: if the data comes in as batches, it makes sense to collapse the modifications from the whole batch into one, and send it at the end of the batch.

To do this in Triceps, I've made a template. Here is an example of its use with interspersed commentary:

```
our $rtData = Triceps::RowType->new(
  # mostly copied from the traffic aggregation example
  local_ip => "string",
  remote_ip => "string",
  bytes => "int64",
) or confess "$!";
```

The meaning of the rows is not particularly important for this example. It just uses a pair of the IP addresses as the collapse key. The collapse absolutely needs a primary key, since it has to track and collapse multiple updates to the same row.

```
my $unit = Triceps::Unit->new("unit");

my $collapse = Triceps::Collapse->new(
  unit => $unit,
  name => "collapse",
  data => [
    name => "idata",
    rowType => $rtData,
    key => [ "local_ip", "remote_ip" ],
  ],
);
```

Most of the options are self-explanatory. The dataset is defined with nested options to make the API extensible, to allow multiple datasets to be defined in the future. But at the moment only one is allowed. A dataset collapses the data at one label: an input label and an output label get defined for it, just as for the table. The data arrives at the input label, gets collapsed by the primary key, and then stays in the Collapse until the flush. When the Collapse gets flushed, the data is sent out of its output label. After the flush, the Collapse has no data in it, and starts collecting the updates again from scratch. The labels gets named by connecting the names of the Collapse element, of the dataset, and "in" or "out". For this Collapse, the label names will be "collapse.idata.in" and "collapse.idata.out".

Note that the dataset options are specified in a referenced array, not a hash! If you try to use a hash, it will fail. When specifying the dataset options, put the "name" first. It's used in the error messages about any issues in the dataset, and the code really expects the name to go first.

Like with the other shown templates, if something goes wrong, Collapse will confess. No need to follow its methods with `or confess`.

```
my $lbPrint = makePrintLabel("print", $collapse->getOutputLabel("idata"));
```

The print label gets connected to the Collapse's output label. The method to get the collapse's output label is very much like table's. Only it gets the dataset name as an argument.

```
sub mainloop($$$) # ($unit, $datalabel, $collapse)
{
  my $unit = shift;
  my $datalabel = shift;
  my $collapse = shift;
  while(<STDIN>) {
    chomp;
    my @data = split(/,/); # starts with a command, then string opcode
    my $type = shift @data;
    if ($type eq "data") {
      my $rowop = $datalabel->makeRowopArray(@data);
      $unit->call($rowop);
      $unit->drainFrame(); # just in case, for completeness
    } elsif ($type eq "flush") {
      $collapse->flush();
    }
  }
}

&mainloop($unit, $collapse->getInputLabel($collapse->getDatasets()), $collapse);
```

There will be a second example, so I've placed the main loop into a function. It works in the same way as in the examples before: extracts the data from the CSV format and sends it to a label. The first column contains the command: "data" sends the data, and "flush" performs the flush from the Collapse. The flush marks the end of the batch. Here is an example of a run, with the input lines shown as usual in bold:

```
data,OP_INSERT,1.2.3.4,5.6.7.8,100
data,OP_INSERT,1.2.3.4,6.7.8.9,1000
```

```
data,OP_DELETE,1.2.3.4,6.7.8.9,1000
flush
collapse.idata.out OP_INSERT local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="100"
```

The row for (1.2.3.4, 5.6.7.8) gets plainly inserted, and goes through on the flush. The row for (1.2.3.4, 6.7.8.9) gets first inserted and then deleted, so by the flush time it becomes a no-operation.

```
data,OP_DELETE,1.2.3.4,5.6.7.8,100
data,OP_INSERT,1.2.3.4,5.6.7.8,200
data,OP_INSERT,1.2.3.4,6.7.8.9,2000
flush
collapse.idata.out OP_DELETE local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="200"
collapse.idata.out OP_INSERT local_ip="1.2.3.4" remote_ip="6.7.8.9"
    bytes="2000"
```

The original row for (1.2.3.4, 5.6.7.8) gets modified, and the modification goes through. The new row for (1.2.3.4, 6.7.8.9) gets inserted now, and also goes through.

```
data,OP_DELETE,1.2.3.4,6.7.8.9,2000
data,OP_INSERT,1.2.3.4,6.7.8.9,3000
data,OP_DELETE,1.2.3.4,6.7.8.9,3000
data,OP_INSERT,1.2.3.4,6.7.8.9,4000
data,OP_DELETE,1.2.3.4,6.7.8.9,4000
flush
collapse.idata.out OP_DELETE local_ip="1.2.3.4" remote_ip="6.7.8.9"
    bytes="2000"
```

The row for (1.2.3.4, 6.7.8.9) now gets modified twice, and after that deleted. After collapse it becomes the deletion of the original row, the one that was inserted before the previous flush.

The Collapse also allows to specify the row type and the input connection for a dataset in a different way:

```
my $lbInput = $unit->makeDummyLabel($rtData, "lbInput");

my $collapse = Triceps::Collapse->new(
  name => "collapse",
  data => [
    name => "idata",
    fromLabel => $lbInput,
    key => [ "local_ip", "remote_ip" ],
  ],
);

&mainloop($unit, $lbInput, $collapse);
```

Normally `$lbInput` would be not a dummy label but the output label of some element. The dataset option “fromLabel” tells that the dataset input will be coming from that label. So the Collapse can automatically both copy its row type for the dataset, and also chain the dataset's input label to that label. And also allowing to skip the option “unit” at the main level. It's a pure convenience, allowing to skip the manual steps. In the future a Collapse dataset should probably take a whole list of source labels and chain itself to all of them, but for now only one.

This example produces exactly the same output as the previous one, so there is no use in copying it again.

Another item that hasn't been shown yet, you can get the list of dataset names (well, currently only one name):

```
@names = $collapse->getDatasets();
```

The Collapse implementation is reasonably small, and is another worthy example to show. It's a common template, with no code generation whatsoever, just a combination of ready components. As with SimpleAggregator, the current Collapse is quite simple and will grow more features over time, so I've copied the original simple version into `t/xCollapse.t` to stay there unchanged.

The most notable thing about Collapse is that it took just about an hour to write the first version of it and another three or so hours to test it. Which is a lot less than the similar code in the Aleri or Coral8 code base took. The reason for this is that Triceps provides the fairly flexible base data structures that can be combined easily directly in a scripting language. There is no need to re-do a lot from scratch every time, just take something and add a little bit on top.

So here it is, with the interspersed commentary.

```
sub new # ($class, $optName => $optValue, ...)
{
  my $class = shift;
  my $self = {};

  &Triceps::Opt::parse($class, $self, {
    unit => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Unit") } ],
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    data => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "ARRAY") } ],
  }, @_);

  # parse the data element
  my $dataref = $self->{data};
  my $dataset = {};
  # dataref->[1] is the best guess for the dataset name, in case if the option "name" goes
 first
  &Triceps::Opt::parse("$class data set (" . $dataref->[1] . ")", $dataset, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    key => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "ARRAY", "") } ],
    rowType => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::RowType"); } ],
    fromLabel => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Label"); } ],
  }, @$dataref);
```

The options parsing goes as usual. The option “data” is parsed again for the options inside it, and those are places into the hash `%$dataset`.

```
  # save the dataset for the future
  $self->{datasets}{$dataset->{name}} = $dataset;
  # check the options
  &Triceps::Opt::handleUnitTypeLabel("Triceps::Collapse data set (". $dataset->{name} .
 ")",
    "unit at the main level", \$self->{unit},
    "rowType", \$dataset->{rowType},
    "fromLabel", \$dataset->{fromLabel});
  my $lbFrom = $dataset->{fromLabel};
```

If “fromLabel” is used, the row type and possibly unit are found from it by `Triceps::Opt::handleUnitTypeLabel()`. Or if the unit was specified explicitly, it gets checked for consistency with the label's unit. See Section 10.5: “Template options” (p. 117) for more detail. The early version of Collapse in `t/xCollapse.t` actually pre-dates `Triceps::Opt::handleUnitTypeLabel()`, and there the similar functionality is done manually.

```
  # create the tables
  $dataset->{tt} = Triceps::TableType->new($dataset->{rowType})
    ->addSubIndex("primary",
      Triceps::IndexType->newHashed(key => $dataset->{key})
    );
  $dataset->{tt}->initialize()
    or confess "Collapse table type creation error for dataset '" . $dataset->{name} . "':
\n$! ";

  $dataset->{tbInsert} = $self->{unit}->makeTable($dataset->{tt}, "EM_CALL", $self-
>{name} . "." . $dataset->{name} . ".tbInsert")
    or confess "Collapse internal error: insert table creation for dataset '" . $dataset-
>{name} . "':\n$! ";
  $dataset->{tbDelete} = $self->{unit}->makeTable($dataset->{tt}, "EM_CALL", $self-
>{name} . "." . $dataset->{name} . ".tbInsert")
    or confess "Collapse internal error: delete table creation for dataset '" . $dataset-
>{name} . "':\n$! ";
```

The state is kept in two tables. The reason for their existence is that after collapsing, the Collapse may send for each key one of:

- a single INSERT rowop, if the row was not there before and became inserted,
- a DELETE rowop if the row was there before and then became deleted,
- a DELETE followed by an INSERT if the row was there but then changed its value,
- or nothing if the row was not there before, and then was inserted and deleted, or if there was no change to the row.

Accordingly, this state is kept in two tables: one contains the DELETE part, another the INSERT part for each key, and either part may be empty (or both, if the row at that key has not been changed). After each flush both tables become empty, and then start collecting the modifications again.

```
  # create the labels
  $dataset->{lbIn} = $self->{unit}->makeLabel($dataset->{rowType}, $self->{name} . "." .
 $dataset->{name} . ".in",
    undef, \&_handleInput, $self, $dataset)
      or confess "Collapse internal error: input label creation for dataset '" . $dataset-
>{name} . "':\n$! ";
  $dataset->{lbOut} = $self->{unit}->makeDummyLabel($dataset->{rowType}, $self->{name} .
 "." . $dataset->{name} . ".out")
    or confess "Collapse internal error: output label creation for dataset '" . $dataset-
>{name} . "':\n$! ";
```

The input and output labels get created. The input label has the function with the processing logic set as its handler. The output label is just a dummy. Note that the tables don't get connected anywhere, they are just used as storage, without any immediate reactions to their modifications.

```
  # chain the input label, if any
  if (defined $lbFrom) {
    $lbFrom->chain($dataset->{lbIn})
      or confess "Collapse internal error: input label chaining for dataset '" . $dataset-
>{name} . "' to '" . $lbFrom->getName() . "' failed:\n$! ";
    delete $dataset->{fromLabel}; # no need to keep the reference any more, avoid a
 reference cycle
  }
```

And if the “fromLabel” was used, the Collapse gets connected to it. After that there is no good reason to keep a separate reference to that label, especially considering that it creates a reference loop that would not be cleaned until the input label get cleaned by the unit. So it gets deleted early instead.

```
  bless $self, $class;
  return $self;
}
```

The final blessing is boilerplate. The constructor creates the data structures but doesn't implement any logic. The logic goes next:

```
# (protected)
# handle one incoming row on a dataset's input label
sub _handleInput # ($label, $rop, $self, $dataset)
{
  my $label = shift;
  my $rop = shift;
  my $self = shift;
  my $dataset = shift;

  if ($rop->isInsert()) {
    # Simply add to the insert table: the effect is the same, independently of
    # whether the row was previously deleted or not. This also handles correctly
    # multiple inserts without a delete between them, even though this kind of
    # input is not really expected.
    $dataset->{tbInsert}->insert($rop->getRow());
```

The Collapse object knows nothing about the data that went through it before. After each flush it starts again from scratch. It expects that the stream of rows is self-consistent, and makes the conclusions about the previous data based on the new data it sees. An INSERT rowop may mean one of two things: either there was no previous record with this key, or there was a previous record with this key and then it got deleted. The Delete table can be used to differentiate between these situations: if there was a row that was then deleted, the Delete table would contain that row. But for the INSERT it doesn't matter: in either case it just inserts the new row into the Insert table. If there was no such row before, it would be the new INSERT. If there was such a row before, it would be an INSERT following a DELETE.

```
  } elsif($rop->isDelete()) {
    # If there was a row in the insert table, delete that row (undoing the previous
 insert).
    # Otherwise it means that there was no previous insert seen in this round, so this
 must be a
    # deletion of a row inserted in the previous round, so insert it into the delete
 table.
    if (! $dataset->{tbInsert}->deleteRow($rop->getRow())) {
      $dataset->{tbDelete}->insert($rop->getRow());
    }
  }
}
```

The DELETE case is more interesting. If we see a DELETE rowop, this means that either there was an INSERT sent before the last flush and now that INSERT becomes undone, or that there was an INSERT after the flush, which also becomes undone. The actions for these cases are different: if the INSERT was before the flush, this row should go into the Delete table, and eventually propagate as a DELETE during the next flush. If the last INSERT was after the flush, then its row would be stored in the Insert table, and now we just need to delete that row and pretend that it has never been.

That's what the logic does: first it tries to remove from the Insert table. If succeeded, then it was an INSERT after the flush, that became undone now, and there is nothing more to do. If there was no row to delete, this means that the INSERT must have happened before the last flush, and we need to remember this row in the Delete table and pass it on in the next flush.

This logic is not resistant to the incorrect data sequences. If there ever are two DELETEs for the same key in a row (which should never happen in a correct sequence), the second DELETE will end up in the Delete table.

```
# Unlatch and flush the collected data, then latch again.
sub flush # ($self)
{
  my $self = shift;
  my $unit = $self->{unit};
  my $OP_INSERT = &Triceps::OP_INSERT;
  my $OP_DELETE = &Triceps::OP_DELETE;
  foreach my $dataset (values %{$self->{datasets}}) {
    my $tbIns = $dataset->{tbInsert};
    my $tbDel = $dataset->{tbDelete};
    my $lbOut = $dataset->{lbOut};
    my $next;
    # send the deletes always before the inserts
    for (my $rh = $tbDel->begin(); !$rh->isNull(); $rh = $next) {
      $next = $rh->next(); # advance the irerator before removing
      $tbDel->remove($rh);
      $unit->call($lbOut->makeRowop($OP_DELETE, $rh->getRow()));
    }
    for (my $rh = $tbIns->begin(); !$rh->isNull(); $rh = $next) {
      $next = $rh->next(); # advance the irerator before removing
      $tbIns->remove($rh);
      $unit->call($lbOut->makeRowop($OP_INSERT, $rh->getRow()));
    }
  }
}
```

The flushing is fairly straightforward: first it sends on all the DELETEs, then all the INSERTs, clearing the tables along the way. At first I've though of matching the DELETEs and INSERTs together, sending them next to each other in case if both are available for some key. It's not that difficult to do. But then I've realized that it doesn't matter and just did it the simple way.

```
# Get the input label of a dataset.
# Confesses on error.
sub getInputLabel($$) # ($self, $dsetname)
{
  my ($self, $dsetname) = @_;
  confess "Unknown dataset '$dsetname'"
    unless exists $self->{datasets}{$dsetname};
  return $self->{datasets}{$dsetname}{lbIn};
}

# Get the output label of a dataset.
# Confesses on error.
sub getOutputLabel($$) # ($self, $dsetname)
{
  my ($self, $dsetname) = @_;
  confess "Unknown dataset '$dsetname'"
    unless exists $self->{datasets}{$dsetname};
  return $self->{datasets}{$dsetname}{lbOut};
}

# Get the lists of datasets (currently only one).
sub getDatasets($) # ($self)
{
  my $self = shift;
  return keys %{$self->{datasets}};
}
```

The getter functions are fairly simple. The only catch is that the code has to check for `exists` before it reads the value of `$self->{datasets}{$dsetname}{lbOut}`. Otherwise, if an incorrect `$dsetname` is used, the reading would return an `undef` but along the way would create an unpopulated `$self->{datasets}{$dsetname}`. Which would then cause a crash when `flush()` tries to iterate through it and finds the dataset options missing.

That's it, Collapse in a nutshell!

14.3. Large deletes in small chunks

If you have worked with Coral8 and similar CEP systems, you should be familiar with the situation when you ask it to delete a million rows from the table and the model goes into self-contemplation for half an hour, not reacting to any requests. It starts responding again only when the deletes are finished. That's because the execution is single-threaded, and deleting a million rows takes time.

Triceps is succeptible to the same issue. So, how to avoid it? Even better, how to make the deletes work “in the background”, at a low priority, kicking in only when there is no other pending requests?

The solution is do do it in smaller chunks. Delete a few rows (say, a thousand or so) then check if there are any other requests. Keep processing these other request until the model becomes idle. Then continue with deleting the next chunk of rows.

Let's make a small example of it. First, let's make a table.

```
our $uChunks = Triceps::Unit->new("uChunks");

# data is just some dumb easily-generated filler
our $rtData = Triceps::RowType->new(
  s => "string",
  i => "int32",
) or confess "$!";

# the data is auto-generated by a sequence
our $seq = 0;

our $ttData = Triceps::TableType->new($rtData)
  ->addSubIndex("fifo", Triceps::IndexType->newFifo())
or confess "$!";
$ttData->initialize() or confess "$!";
our $tData = $uChunks->makeTable($ttData,
  &Triceps::EM_CALL, "tJoin1"
) or confess "$!";
makePrintLabel("lbPrintData", $tData->getOutputLabel());
```

The data in the table is completely silly, just something to put in there. Even the index is a simple FIFO, just something to keep the table together.

Next, the clearing logic.

```
# notifications about the clearing
our $rtNote = Triceps::RowType->new(
  text => "string",
) or confess "$!";

# rowops to run when the model is otherwise idle
our $trayIdle = $uChunks->makeTray();

our $lbReportNote = $uChunks->makeDummyLabel($rtNote, "lbReportNote"
) or confess "$!";
makePrintLabel("lbPrintNote", $lbReportNote);

# code that clears the table in small chunks
our $lbClear = $uChunks->makeLabel($rtNote, "lbClear", undef, sub {
  my $limit = 2; # no more than 2 rows per run
  my $next;
  for (my $rhit = $tData->begin(); !$rhit->isNull(); $rhit = $next) {
    if ($limit-- <= 0) {
      # request to be called again when the model becomes idle
      $trayIdle->push($_[0]->adopt($_[1]));
      return;
    }
    $next = $rhit->next(); # advance before removal
    $tData->remove($rhit);
  }
  $uChunks->makeHashCall($lbReportNote, "OP_INSERT",
    text => "done clearing",
  );
}) or confess "$!";
```

We want to get a notification when the clearing is done. This notification will be sent as a rowop with row type `$rtNote` to the label `$lbReportNote`. Which then just gets printed, so that we can see it. In a production system it would be sent back to the requestor.

The clearing is initiated by sending a row (of the same type `$rtNote`) to the label `$lbClear`. Which does the job and then sends the notification of completion. In the real world not the whole table would probably be erased but only the old data, from before a certain date, like was shown in the Section 12.11: “JoinTwo input event filtering” (p. 193) . Here for simplicity all the data get wiped out.

But the loop stops after the number of deleted rows reaches the limit. Since it's real inconvenient to play with a million rows, we'll play with just a few rows. And so the chunk size limit is also set smaller, to just two rows instead of a thousand. When the limit is reached, the code pushes the command row into the idle tray for later rescheduling and returns. The adoption part is not strictly necessary, and this small example would work fine without it. But it's a safeguard for the more complicated programs that may have the labels chained, with our clearing label being just one link in a chain. If the incoming rowop gets rescheduled as is, the whole chain will get executed again. which might not be desirable. Re-adopting it to our label will cause only our label (okay, and everything chained from it) to be executed.

How would the rowops in the idle tray get executed? In the real world, the main loop logic would be like this pseudocode:

```
while(1) {
  if (idle tray is empty)
    timeout = infinity;
  else
    timeout = 0;
  poll(file descriptors, timeout);
  if (poll timed out)
    run the idle tray;
  else
    process the incoming data;
}
```

The example from Section 7.8: “Main loop with a socket” (p. 51) can be extended to work like this. But it's hugely inconvenient for a toy demonstration, getting the timing right would be a major pain. So instead let's just add the command “idle” to the main loop, to trigger the idle logic at will. The main loop of the example is:

```
while(<STDIN>) {
  chomp;
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  if ($type eq "data") {
    my $count = shift @data;
    for (; $count > 0; $count--) {
      ++$seq;
      $uChunks->makeHashCall($tData->getInputLabel(), "OP_INSERT",
        s => ("data_" . $seq),
        i => $seq,
      );
    }
  } elsif ($type eq "dump") {
    for (my $rhit = $tData->begin(); !$rhit->isNull(); $rhit = $rhit->next()) {
      print("dump: ", $rhit->getRow()->printP(), "\n");
    }
    for my $r ($trayIdle->toArray()) {
      print("when idle: ", $r->printP(), "\n");
    }
  } elsif ($type eq "clear") {
    $uChunks->makeHashCall($lbClear, "OP_INSERT",
      text => "clear",
    );
  } elsif ($type eq "idle") {
    $uChunks->schedule($trayIdle);
    $trayIdle->clear();
  }
  $uChunks->drainFrame(); # just in case, for completeness
}
```

The data is put into the table by the main loop in a silly way: When we send the command like “`data,3`”, the mail loop will insert 3 new rows into the table. The contents is generated with sequential numbers, so the rows can be told apart. As the table gets changed, the updates get printed by the label `lbPrintData`.

The command “dump” dumps the contents of both the table and of the idle tray.

The command “clear” issues a clearing request by calling the label `$lbClear`. The first chunk gets cleared right away but then the control returns back to the main loop. If not all the data were cleared, an idle rowop will be placed into the idle tray.

The command “idle” that simulates the input idleness will then pick up that rowop from the idle tray and reschedule it.

All the pieces have been put together, let's run the code. The commentary are interspersed, and as usual, the input lines are shown in bold:

```
data,1
tJoin1.out OP_INSERT s="data_1" i="1"
clear
tJoin1.out OP_DELETE s="data_1" i="1"
lbReportNote OP_INSERT text="done clearing"
```

This is pretty much a dry run: put in one row (less than the chunk size), see it deleted on clearing. And see the completion reported afterwards.

```
data,5
tJoin1.out OP_INSERT s="data_2" i="2"
tJoin1.out OP_INSERT s="data_3" i="3"
tJoin1.out OP_INSERT s="data_4" i="4"
tJoin1.out OP_INSERT s="data_5" i="5"
tJoin1.out OP_INSERT s="data_6" i="6"
```

Add more data, which will be enough for three chunks.

```
clear
tJoin1.out OP_DELETE s="data_2" i="2"
tJoin1.out OP_DELETE s="data_3" i="3"
```

Now the clearing does one chunk and stops, waiting for the idle condition.

```
dump
dump: s="data_4" i="4"
dump: s="data_5" i="5"
dump: s="data_6" i="6"
when idle: lbClear OP_INSERT text="clear"
```

See what's inside: the remaining 3 rows, and a row in the idle tray saying that the clearing is in progress.

```
idle
tJoin1.out OP_DELETE s="data_4" i="4"
tJoin1.out OP_DELETE s="data_5" i="5"
```

The model goes idle once more, one more chunk of two rows gets deleted.

```
data,1
tJoin1.out OP_INSERT s="data_7" i="7"
```

```
dump
dump: s="data_6" i="6"
dump: s="data_7" i="7"
when idle: lbClear OP_INSERT text="clear"
```

What will happen if we add more data in between the chunks of clearing? Let's see, let's add one more row. It shows up in the table as usual.

```
idle
tJoin1.out OP_DELETE s="data_6" i="6"
tJoin1.out OP_DELETE s="data_7" i="7"
lbReportNote OP_INSERT text="done clearing"
dump
idle
```

On the next idle condition the clearing picks up whatever was in the table for the next chunk. Since there were only two rows left, it's the last chunk, and the clearing reports a successful completion. And a dump shows that there is nothing left in the table nor in the idle tray. The next idle condition does nothing, because the idle tray is empty.

The deletion could also be interrupted and cancelled, by removing the row from the idle tray. That would involve converting the tray to an array, finding and deleting the right rowop, and converting the array back into the tray. Overall it's fairly straightforward. The search in the array is linear but there should not be that many idle requests, so it should be quick enough.

The delete-by-chunks logic can be made into a template, just I'm not sure yet what is the best way to do it. It would have to have a lot of configurable parts.

On another subject, scheduling the things to be done on idle adds an element of unpredictability to the model. It's impossible to predict the exact timing of the incoming requests, and the idle work may get inserted between any of them. Presumably it's OK because the data being deleted should not be participating in any logic at this time any more. For repeatability in the unit tests, make the chunk size adjustable and adjust it to a size larger than the biggest amount of data used in the unit tests.

A similar logic can also be used in querying the data. But it's more difficult. For deletion the continuation is easy: just take the first row in the index, and it will be the place to continue (because the index is ordered correctly, and because the previous rows are getting deleted). For querying you would have to remember the next row handle and continue from it. Which is OK if it can not get deleted in the meantime. But if it can get deleted, you'll have to keep track of that too, and advance to the next row handle when this happens. And if you want to receive a full snapshot with the following subscription to all updates, you'd have to check whether the modified rows are before or after the marked handle, and pass them through if they are before it, letting the user see the updates to the data already received. And since the data is being sent to the user, filling up the output buffer and stopping would stop the whole model too, and not restart until the user reads the buffered data. So there has to be a flow control logic that would stop the query when output buffer fills up, return to the normal operation, and then reschedule the idle job for the query only when the output buffer drains down. I've kind of started on doing an example of the chunked query too, but then because of all these complications decided to leave it for later.

Chapter 15. Triceps Perl API Reference

There are two distinct ways to descibe something, a “guide” and a “reference”. Most of this manual is a guide: it goes by describing things by examples, together with the idioms of their usage, and with the explanation of the internal structure and underlying reasons. However if you already know how things work and need to look up the trivia, a reference with its short and dry descriptions comes handy. Besides, some methods of the classes essentially are the trivia, and there is no point in making elaborate examples about them. The reference for the whole Perl API is collected here, organized by classes. Eventually it should be placed into the man pages as well, but so far I haven't got around to do it.

Some of the classes are so fundamental that the guide sections about them were essentially of the reference type, with the use being shown in all the examples of the manual. In such cases I'm not copying them here, instead please refer to the relevant chapters:

Simple field types in Section 5.1: “Simple types” (p. 25) .

RowType in Section 5.2: “Row types” (p. 25) .

Row in Section 5.4: “Rows” (p. 28) .

Label in Chapter 6: “*Labels and Row Operations*” (p. 31) .

Rowop in Section 6.4: “Row operations” (p. 34) .

Unit in Section 7.5: “Execution unit” (p. 46) .

15.1. TableType reference

The TableType is the information about the structure of a Table. It can be used to create multiple Tables in the same mold.

```
$tt = Triceps::TableType->new($rowType) or confess "$!";
```

Constructs the TableType. The TableType is anonymous, it has no string name. After that it can be configured by adding the index types. Eventually it has to be initialized and that freezes the table type and makes it immutable. All the steps up to and including the initialization must be done from a single thread, after initialization a table type may be shared between multiple threads.

```
$tt->addSubIndex("indexName", $indexType) or confess "$!";
```

Adds an index type, naming it within the scope of the table type. The result is the same table type (unless it's an `undef` signifying an error), so the index type additions can be chained with each other and with the construction:

```
$tt = Triceps::TableType->new($rowType)
  ->addSubIndex("indexName1", $indexType1)
  ->addSubIndex("indexName2", $indexType2)
  or confess "$!";
```

The table type initialization freezes not only the table type itself but also all the index types in it. Also, the index types become permanently tied to this one table type. That would make things difficult if the same index type is added to two table types. To avoid these issues, `addSubIndex()` adds not the actual argument index type but first creates a fresh uninitialized deep copy of it, and then adds it.

```
$tt->initialize() or confess "$!";
```

Initializes the table type. The index types check most of their arguments at the initialization time, so that's where most of the errors will be reported. Calling `initialize()` repeatedly will have no effect and just return the same result again and again.

```
$result = $tt->isInitialized();
```

Checks whether the table type has been initialized.

```
$rowType = $tt->rowType();
$rowType = $tt->getRowType();
```

Returns the row type. One method name is historic, the other has been added for consistency.

```
$indexType = $tt->findSubIndex("indexName") or confess "$!";
```

Finds an index type by name. This is symmetric with `addSubIndex()`, so it works only for the top-level index types. To get the nested ones, repeat the same call on the found index types or see the following methods.

```
$indexType = $tt->findIndexPath( ["indexName", "nestedIndexName"] );
```

Finds an index type by the path of names leading to it in the index type tree. If the path is not found, the function would confess. An empty path is also illegal and would cause the same result. The argument is not an array but a reference to array of names.

```
($indexType, @keys) = $tt->findIndexKeyPath( ["indexName", "nestedIndexName"] );
```

Finds by path an index type that allows the direct look-up by key fields. It requires that every index type in the path returns a non-empty array of fields in `getKey()`. In practice it means that every index in the path must be a Hashed index. Otherwise the method confesses. When the Sorted and maybe other index types will support `getKey()`, they will be usable with this method too. The argument is not an array but a reference to array of names.

Besides checking that each index type in the path works by keys, this method builds and returns the list of all the key fields required for a look-up in this index. Note that `@keys` is an actual array and not a reference to array. The return protocol of this method is a little weird: it returns an array of values, with the first value being the reference to the index type, and the rest of them the names of the key fields.

```
$indexType = $tt-findSubIndexById($indexTypeId) or confess "$!";
```

Finds the first top-level index type of a particular kind. The `$indexTypeId` is one of the `IT_*` constants in integer or string form.

```
$indexType = $tt->getFirstLeaf();
```

Returns the first leaf index type (the one used for the default look-ups and iteration on the tables of this type).

```
@indexTypes = $tt->getSubIndexes();
%indexTypes = $tt->getSubIndexes();
```

Returns all the top-level index types. The resulting array contains the pairs of names and index types. If the order is not important but you want to perform the look-ups by name, the result can be stored directly into a hash. However if you plan to use the data to add index types to another table type, don't use the hash because the order of indexes is important and the hash loses it.

```
$result = $tt1->same($tt2);
$result = $tt1->equals($tt2);
$result = $tt1->match($tt2);
```

The usual reference comparison methods.

Two table types are considered equal when they have the equal row types, and exactly the same set of index types, with the same names.

Two table types are considered matching when they have the matching row types, and matching set of index types, although the names of the index types may be different.

```
$res = $tt->print();
```

Presents the content of a table type as a human-readable description. Accepts the usual `print()` arguments.

15.2. IndexType reference

The IndexType is a part of TableType and defines the structuring of rows in the table. It provides the order of rows and optionally a way to find them quickly by the key. The configuration of the index type defines the parameters for each index instance, i.e. each row group in an index of this type, not for the whole table. The difference between indexes and index types is explained in the Section 9.10 . The index types are connected in a table type to form a tree.

The index types in Triceps are available in the following kinds:

Hashed
: Provides quick random access based on the key formed from the fields of the row in the table. May be leaf or non-leaf. The order of the rows in the index will be repeatable between the runs of the same program on the same machine architecture, but not easily predictable. Internally the rows are stored in a tree but the comparisons of the rows are accelerated by pre-calculating a hash value from the key fields and keeping it in the row handle.

FIFO
: Keeps the rows in the order they were received. There is no efficient way to find a particular row in this index, the search in it works by going through all the rows sequentially and comparing the rows for exact equality. It provides the expiration policies based on the row count. It may only be a leaf.

PerlSorted
: Provides random access based on the key field comparison, expressed as a Perl function. This results in a predictable order of rows but the execution of the Perl code makes it slower than the Hashed index. May be leaf or non-leaf. Often also called simply “Sorted”.

SimpleOrdered
: A Perl template on top of the PerlSorted index, that allows to specify the keys in a more convenient way. Often also called simply “Ordered”.

```
$it = Triceps::IndexType->newHashed($optionName => $optionValue, ...)
  or confess "$!";
```

Creates a Hashed index type. The only available option is “key”, and it's mandatory. It's argument is the reference to an array of strings that specify the names of the key fields (`key => ["f1", "f2"]`).

```
$it = Triceps::IndexType->newFifo($optionName => $optionValue, ...)
  or confess "$!";
```

Creates a FIFO index type. The options are:

`limit`
: Sets the limit value for the replacement policy. Once the number of rows attempts to grow beyond this value, the older records get removed. Setting it to 0 disables the replacement policy, which is the default. Don't try to set it to negative values, they will be treated as unsigned, and thus become some very large positive ones.

`jumping`
: Determines the variation of the replacement policy in effect. If set to 0 (default), implements the sliding window policy, removing the older rows one by one. If non-0, implements the jumping window policy, removing all the older rows when a new row causes the limit overflow.

`reverse`
: Defines the iteration order. If non-0, the iteration on this index goes in the reverse order. However the expiration policy still works in the direct order! The default is 0.

```
$it = Triceps::IndexType->newPerlSorted($sortName, \&initFunc,
  \&compareFunc, @args...) or confess "$!";
```

Creates a PerlSorted index type. The arguments are:

`$sortName`
: a string describing the sorting order, used in `print()` and error messages.

`\&initFunc`
: a function reference that can be used to generate the comparison function dynamically at the table type initialization time (or use `undef` with a fixed comparison function).

`\&compareFunc`
: a function reference to the fixed comparison function, if preferred (or use `undef` if it will be generated dynamically by the init function).

`@args`
: optional extra arguments for the initialization and/or comparison function.

See the details in Section 9.8: “Sorted index” (p. 87) .

```
$it = Triceps::SimpleOrderedIndex->new($fieldName => $order, ...)
  or confess "$!";
```

Creates a SimpleOrdered index type. The arguments are the key fields. `$order` is one of the constants `"ASC"` for ascending or `"DESC"` for descending.

```
$indexType2->addSubIndex("indexName", $indexType1) or confess "$!";
```

Attaches the nested `$indexType1` under `$indexType2`. More exactly, attaches an uninitialized deep copy of `$indexType1`, the same as when adding an index type under a table type. It returns the reference to the same `$indexType2`, so these calls can be conveniently chained, to add multiple sub-indexes under it. If `$indexType2` can not be non-leaf, the call will fail.

```
$itSub = $it->findSubIndex("indexName") or confess "$!";
$itSub = $it->findSubIndexById($indexTypeId) or confess "$!";

@itSubs = $it->getSubIndexes();
$itSub = $it->getFirstLeaf();
```

Perform the same actions as the same-named methods in the TableType. If the index type is already a leaf, `getFirstLeaf()` will return itself.

```
$it->setAggregator($aggType) or confess "$!";
```

Sets an aggregator type on an index type. It will create aggregators that run on the rows stored withing the indexes of this type. The value returned is the same index type reference `$it`, allowing the chaining calls, along with the `addSubIndex()`. Only one aggregator type is allowed on an index type. Calling `setAggregator()` repeatedly will replace the aggregator type.

```
$aggType = $it->getAggregator();
```

Returns the aggregator type set on this index type. The returned value may be `undef` (but `$!` not set) if no aggregator type has been set.

```
$result = $it->isInitialized();
```

Returns, whether this type has been initialized. The index type gets initialized when the table type where it belongs gets initialized. After an index type has been initialized, it can not be changed any more, and any methods that change it will return an error.

```
$itCopy = $it->copy();
```

Creates a copy of the index type. The copy reverts to the un-initialized state. It's always a deep copy, with all the nested index and aggregator types copied. All of these copies are un-initialized.

```
$tabType = $it->getTabtype() or confess "$!";
```

Returns the table type, to which this index type is tied. When an index type becomes initialized, it becomes tied to a particular table type. If the index type is not initialized yet, this will return an error.

```
$result = $it1->same($it2);
$result = $it1->equals($it2);
$result = $it1->match($it2);
$result = $it->print();
```

The usual sameness comparisons and print methods.

Two index types are considered equal when they are of the same kind (type id), their type-specific parameters are equal, they have the same number of sub-indexes, with the same names, and equal pair-wise. They must also have the equal aggregators.

Two index types are considered matching when they are of the same kind, have matching type-specific parameters, they have the same number of sub-indexes, which are matching pair-wise, and the matching aggregators. The names of the sub-indexes may differ. As far as the type-specific parameters are concerned, it depends on the kind of the index type. The FIFO type considers any parameters matching. For a Hashed index the key fields must be the same. For a Sorted index the sorted condition must also be the same, and by extension this means the same condition for the Ordered index.

```
$result = $it->isLeaf();
```

Returns 1 if the index type is a leaf, 0 if not.

```
@keys = $it->getKey();
```

Returns the array of field names forming the key of this index. Currently works only on the Hashed index types. On the other index types it returns an empty array, though probably a better support would be available for the PerlSorted/SimpleOrdered indexes in the future.

```
$it->setComparator(\&compareFunc, @args...) or confess "$!";
```

A special method that works only on the PerlSorted index types. Sets an auto-generater comparator function and its optional arguments from an initializer function at the table initialization time. On success it returns 1. For all other index types this method returns an error.

15.3. AggregatorType reference

The aggregator type describes an aggregation. It gets connected to an index type which defines the grouping for the aggregator. Whenever the aggregation is performed, the code from the aggregator type receives the group context as its argument.

```
$at = Triceps::AggregatorType->new($resultRowType, "aggName", \&initFunc,
  \&handlerFunc, @args) or die "$!";
```

Creates an aggregator type. The rows created by the aggregator will be of `$resultRowType`. The aggregator name is used to name the aggregator result label in the table, “tableName.aggName”. It is also used to get the reference of that label from the table.

The optional `@args` are passed to both the init and handler functions (to which `\&initFunc` and `\&handlerFunc` are references). The init function is called when the row group (contained in an index of the type, on which this aggregator type is set) is created. It initializes the group's aggregation state. The handler function gets called on the changes to the group. See the details in Section 11.5: “Optimized DELETEs” (p. 147) , Section 11.6: “Additive aggregation” (p. 149) . and Section 11.7: “Computation function arguments” (p. 153) .

```
$result = $at1->same($at2);
$result = $at1->equals($at2);
$result = $at1->match($at2);
$result = $at->print();
$atCopy = $at->copy();
```

The methods for comparison, printing and copying work similarly to the index types.

The equal aggregator types have the equal result row types, same names, same initialization and handler function references, same arguments.

The matching aggregator types may differ in the aggregator name and in the field names of the result row type. However the function references and their arguments must still be the same.

15.4. SimpleAggregator reference

SimpleAggregator provides an easier way to describe aggregations with the SQL-like aggregation functions. It also supports the user-defined aggregation functions.

```
$tabType = Triceps::SimpleAggregator::make($optName => $optValue, ...);
```

Creates an aggregator type from the high-level description and sets it on an index type in the table type. Returns back the table type passed as an option argument. Confesses on errors. This is **not** a class constructor. It creates a common AggregatorType with the automatically generated code for the initialization and handler functions.

Most of the options are mandatory, unless noted otherwise. The options are:

`name`
: The aggregator type name.

`tabType`
: Table type to put the aggregator on. It must be un-initialized yet.

`idxPath`
: A reference to an array of index names, forming the path to the index where the aggregator type will be set. For example, `["index", "subIndex"]`.

`result`
: A reference to an array defining the result of the aggregation. It consists of the repeating groups of four elements:

```
fieldName => type, function, \&argFunc,
```

Here the type is the field type name, the function is the name of the aggregtion function (case-sensitive, see the list in Section 11.9: “SimpleAggregator” (p. 159)), and the `\&argFunc` computes the argument of the aggregation function. It's a reference to a function that receives the current row being aggregated as `$_[0]` and computes a value from its fields. These values from all the rows in the group then get fed to the aggregation function. If the aggregation function requires no argument, `argFunc` must be `undef`. For example:

```
  result => [
    symbol => "string", "last", sub {$_[0]->get("symbol");},
    count => "int32", "count_star", undef,
    cost => "float64", "sum", sub {$_[0]->get("size") * $_[0]->get("price")];},
    vwap => "float64", "nth_simple", sub { [1, $_[0]->get("price")];},
  ],
```

`saveRowTypeTo`
: Optional. A reference to a scalar where to save the result row type. It will be available when `Triceps::SimpleAggregator::make()` returns. Later when a table with this aggregator type gets construct-

ed, its result row type may also be found with `$table->getAggregatorLabel("aggName")->getRowType()`.

`saveInitTo`

Optional. A reference to a scalar where to save the auto-generated source code of the initialization function for diagnostics.

`saveComputeTo`

Optional. A reference to a scalar where to save the auto-generated source code of the handler function for diagnostics.

`functions`

Optional. The additional user-defined aggregation functions. See the description of their structure in Section 11.9: “SimpleAggregator” (p. 159) .

The aggregator types produced by the SimpleAggregator would be equal and matching only if they have been produced by copying (you can also copy a table type or index type with an AggregatorType in it).

15.5. Table reference

The Table provides the structured data storage in Triceps.

```
$t = $unit->makeTable($tabType, $enqMode, "tableName") or confess "$!";
```

Creates a table of a given table type. The table type must be initialized before it can be used to create tables. The tables are strictly single-threaded.

The enqueueing mode defines, how the rows will be enqueued to the output label. It can be specified as a string or Triceps constant. However in the modern reality you should use `"EM_CALL"` or `&Triceps::EM_CALL`. This argument is likely to be removed altogether in the future and become fixed to `"EM_CALL"`. For the Pre label, it is already fixed to `"EM_CALL"`.

The table name is used for the error messages and as a base for the names of the table labels.

```
$result = $t1->same($t2);
```

The usual comparison for sameness. (There is no comparison for equality and mathicng, use the table type for that; nor printing, use the table name and/or table type for that).

```
$lb = $t->getInputLabel();
$lb = $t->getPreLabel();
$lb = $t->getOutputLabel();
$lb = $t->getAggregatorLabel("aggName") or confess "$!";
```

Get the labels that are creates as a part of the table. With an invalid name for an argument, `getAggregatorLabel()` returns an `undef`.

```
$tt = $t->getType();
$u = $t->getUnit();
$rt = $t->getRowType();
$name = $t->getName();
```

Get back the information about the table configuration.

```
$result = $t->size();
```

Returns the number of rows in the table.

```
$rh = $t->makeRowHandle($row) or confess "$!";
```

```
$rh = $t->makeNullRowHandle();
```

Create the row handles. The row must be of a matching type, and it will be cast to the table's row type: when read back from the row handle, the row will have the table's row type as its type. The casting does not involve any copying or modification of the row, it's still shared by reference counting. And the original row as it was would still return the same type. Basically, the row itself is untyped, its type is determined by the container where it is stored. The requirement for the matching types ensures that when a row is passed between the containers, they have a compatible notion of the row type.

A NULL row handle is a handle without a row in it. It can not be placed into a table but this kind of row handle gets returned by table operations to indicate things not found. In case if you want to fool some of your code by slipping it a NULL handle, `makeNullRowHandle()` provides a way to do it. The row handles belong to a particular table and can not be mixed between them, even if the tables are of the same type.

The table operations can be done by either sending the rowops to the table's input label or by calling the operations directly.

```
$result = $t->insert($row_or_rh);
$result = $t->insert($row_or_rh, $copyTray);
```

Inserts a row or row handle into the table. The row handle must not be in the table before the call, it may be either freshly created or previously removed from the table. If a row is used as an argument, it is internally wrapped in a fresh row handle, and then that row handle inserted. An insert may trigger the replacement policy in the table's indexes and have some rows removed before the insert is done. The optional copy tray can be used to collect a copy of all the row updates that happen in the table as a result of the insert, both on the table output label and on all its aggregator labels. Returns 1 on success, 0 if the insert can not be done (the row handle is already in the table or NULL), confesses on an incorrect argument.

```
$result = $t->remove($rh);
$result = $t->remove($rh, $copyTray);
```

Removes a row handle from the table. The row handle must be previously inserted in the table, and either found in it or a reference to it remembered from before. An attempt to remove a newly created row handle will have no effect. The optional copy tray works in the same way as for `insert()`. The result is 1 on success (even if the row handle was not in the table), confesses on an incorrect argument.

```
$result= $t->deleteRow($row);
$result= $t->deleteRow($row, $copyTray);
```

Finds the handle of the matching row by the table's first leaf index and removes it. Returns 1 on success, 0 if the row was not found, confesses on an incorrect argument. Unlike `insert()`, the deletion methods for a row handle and a row are named differently to emphasize their difference. The method `remove()` must get a reference to the exactly same row handle that was previously inserted. The method `deleteRow()` does not have to get the same row as was previously inserted, instead it will find a row handle of the row that has the same key as the argument, according to the first leaf index. `deleteRow()` never deletes more than one row. If the index contains multiple matching rows (for example, if the first leaf is a FIFO index), only one of them will be removed, usually the first one (the exact choice depends on what row gets found by the index).

```
$rh = $t->find($row_or_rh);
$rh = $t->findIdx($idxType, $row_or_rh);
```

Find the row handle the table by indexes. If the row is not found, return a NULL row handle. If the row is of an incorrect type or the index type is incorrect, confesses. The index type must be exactly the one belonging to the type of this table (not its copy nor the original from which it was copied into the table's type), so the only way to get it is to find it in the table's type after it has been constructed. The default `find()` works using the first leaf index type, i.e. the following two are equivalent:

```
$t->find($r);
$t->findIdx($t->getType()->getFirstLeaf(), $r);
```

but the `find()` version is slightly more efficient because it handles the index types inside the C++ code and does not create the Perl wrappers for them.

The `find()` operation is also used internally by `deleteRow()` and to process the rowops received at the table's input label.

If a row is used as an argument for `find`, a temporary row handle is internally created for it, and then the find is performed on it. Note that if you have a row handle that is already in the table, there is generally no use calling `find` on it, you will just get the same row handle back (well, except for the case of multi-valued indexes, then you will get back some matching row handle, usually the first one, which may be the same or not).

A `findIdx()` with a non-leaf index argument is a special case: it returns the first row handle of the group that has the key matching the argument. The order of "first" in this case is defined according to that index'es first leaf sub-index.

```
$rh = $t->findBy("fieldName" => $fieldValue, ...);
$rh = $t->findIdxBy($idxType, "fieldName" => $fieldValue, ...);
```

Convenience methods that construct a row from the field arguments and then find it. They confess on incorrect arguments.

```
$rh = $t->begin();
$rh = $t->next($rh);
$rh = $t->beginIdx($idxType);
$rh = $t->nextIdx($idxType, $rh);
```

Iteration on the table. The methods `next()` and `nextIdx()` are equivalent to the same methods of the row handle. As usual, the versions without an explicit index type use the first leaf index type. The `begin` methods return the first row handle according to an index'es order, the `next` methods advance to the next row handle. When the end of the table is reached, these methods return a NULL row handle. The `next` methods also return a NULL row handle if their argument row handle is a NULL or not in the table. So, if you iterate and remove the row handles, make sure to advance the iterator first and only then remove the current row handle. If an error is detected, these methods confess.

If the index argument is non-leaf, it's equivalent to its first leaf.

```
$endrh = $t->nextGroupIdx($subIdxType, $rh_in_group);
```

Finds the first row handle of the next group, where `$subIdxType` is the first index inside the group (not its parent index!). Confesses on errors. The result also works as the end marker handle of the current group.

To iterate through only a group, use `findIdx()` on the parent index type of the group to find the first row of the group. Then things become tricky: take the first index type one level below it to determine the iteration order (a group may have multiple indexes in it, defining different iteration orders, the first one will give the group's default order). Use that index type with `nextGroupIdx()` to find the first row handle past the end of the group, and with the usual `nextIdx()` to advance the iterator. However the end of the group will not be signaled by a NULL row handle. It will be signaled by `nextIdx()` returning the same row handle as previously returned by `nextGroupIdx()` (compare them with `$endrh->same($itrh)`).

The value returned by `nextGroupIdx()` is actually the first row handle of the next group, so it can also be used to jump quickly to the next group, and essentially iterate by groups. After the last group, `nextGroupIdx()` will return a NULL row handle. Which is OK for iteration, because at the end of the last group `nextIdx()` will also return a NULL row handle.

What if a group has a whole sub-tree of indexes in it, and you want to iterate it by the order of not the first sub-index? Still use `findIdx()` in the same way to find a row handle in the desired group. But then convert it to the first row handle in the desired order:

```
$beginrh = $t->firstOfGroupIdx($subIdxType, $rh);
```

The `$subIdxType` is the same as used in `nextGroupIdx()`. After that proceed as before: get the end marker with `nextGroupIdx()` on the same sub-index, and iterate with `nextIdx()` on it.

This group iteration is somewhat messy and tricky, and maybe something better can be done with it in the future. If you look closely, you can also see that it doesn't allow to iterate the groups in every possible order. For example, if you have an index type hierarchy

```
A
+-B
| +-D
| | +-G
| | +-H
| +-E
+-C
```

and you want to iterate on the group inside B, you can go in the order of D or G (which is the same as D, since G is the first leaf of D) or of E, but you can not go in the order of H. But for most of the practical purposes it should be good enough.

```
$size = $table->groupSizeIdx($idxType, $row_or_rh);
```

Finds the size of a group without iteration on it. `$idxType` is the parent index of the group (the same as would be used with `findIdx()`). Naturally, it must be a non-leaf index. (Using a non-leaf index type is not an error but it always returns 0, because there are no groups under it). Confesses on errors. The argument may be a row or row handle that identifies any row in the group. If the argument is a row, it gets handled similarly to `findIdx()`: a temporary row handle gets created, used to find the result, and then destroyed. If there is no such group in the table, the result will be 0. If the argument is a row handle, that handle may be in the table or not in the table, either will be handled transparently (though calling it for a row handle that is in the table is more efficient because the group would not need to be found first).

15.6. RowHandle reference

A RowHandle is essentially the glue that keeps a row in the table. A row's handle keeps the position of the row in the table and allows to navigate from it in the direction of every index. It also keeps the helper information for the indexes. For example, the Hashed index calculates the has value for the row's fields once and remembers it in the handle. The table operates always on the handles, never directly on the rows. The table methods that accept rows as arguments, implicitly wrap them into handles before doing any operations.

A row handle always belongs to a particular table, and can not be mixed between the tables, even if the tables are of the same type. Even before a row handle has been inserted into the table and after it has been removed, it still belongs to that table and can not be inserted into any other one.

Just as the tables are single-threaded, the row handles are single-threaded.

```
$rh = $table->makeRowHandle($row) or confess "$!";
```

Creates the RowHandle. The newly created row handle is not inserted in the table. The type of the argument row must be matching the table's row type.

```
$result = $rh->isInTable();
```

Finds out, whether the row handle is inserted in the table.

```
$result = $rh->isNull();
```

Finds out if the RowHandle is NULL. A RowHandle may be NULL to indicate the special conditions. It pretty much means that there is only the Perl wrapper layer of RowHandle but no actual RowHandle under it. This happens to be much more convenient than dealing with undefined values at Perl level. The NULL row handles are returned by the certain table calls to indicate that the requested data was not found in the table.

```
$rh = $table->makeNullRowHandle();
```

Creates a NULL RowHandle.

```
$result = $rh1->same($rh2);
```

The usual comparison for sameness.

```
$row = $rh->getRow() or confess "$!";
```

Extracts the row from the handle. The row will have the type of the table's row type. A row can not be extracted from a NULL row handle.

```
$rh = $rh->next();
$rh = $rh->nextIdx($idxType);
$rh = $rh->firstOfGroupIdx($idxType);
$rh = $rh->nextGroupIdx($idxType);
```

These methods work exactly the same as the same-named table methods. They confess on errors. They are essentially syntactic sugar over the table methods.

15.7. AggregatorContext reference

AggregatorContext is one of the arguments passed to the aggregator computation function. It encapsulates the iteration through the aggregation group, in the order of the index on which the aggregator is defined. After the computation function returns, the context becomes invalidated and stops working, so there is no point in saving it between the calls. There is no way to construct the aggregator context directly.

All the methods of the AggregatorContext confess on errors.

An aggregator must never change the table. Any attempt to change the table is a fatal error.

```
$result = $ctx->groupSize();
```

Returns the number of rows in the group.

```
$rowType = $ctx->resultType();
```

Returns the row type of the aggregation result.

```
$rh = $ctx->begin();
```

Returns the first row handle of the iteration. In case of an empty group it would return a NULL handle.

```
$rh = $ctx->next($rh);
```

Returns the next row handle in order. If the argument handle was the last row in the group, returns a NULL handle. So the iteration through the group with a context is similar to iteration through the whole table: it ends when `begin()` or `next()` returns a NULL handle.

```
$rh = $ctx->beginIdx($idxType);
```

Returns the first row in the group, according to a specific index type's order. The index type must belong to the group, otherwise the result is undefined. If the group is empty, will return the same value as `endIdx()`. If `$idxType` is non-leaf, the effect is the same as if its first leaf were used.

```
$rh = $ctx->endIdx($idxType);
```

Returns the handle past the last row in the group, according to a specific index type's order. The index type must belong to the group, otherwise the result is undefined and might even result in an endless iteration loop. If `$idxType` is non-leaf, the effect is the same as if its first leaf were used. This kind of iteration uses the table's `$t->nextIdx($idxType, $rh)` or `$rh->next($idxType)` to advance the position. Unlike the general group iteration described in Section 15.5: "Table reference" (p. 245), the aggregator context does allow the iteration by every index in the group. You can pick any index in the group and iterate in its order. And aggregation is where this ability counts the most.

If the group happens to be the last group of this index type (not of `$idxType` but of the index on which the aggregator is defined) in the table, `endIdx()` would return a NULL row handle. If it's also empty, `beginIdx()` would also return a

NULL handle, and in general, for an empty group `beginIdx()` would return the same value as `endIdx()`. If the group is not the last one, `endIdx()` returns the handle of the first row in the next group.

```
$rh = $ctx->lastIdx($idxType);
```

Returns the last row in the group according to a particular index type's order. The index type must belong to the group, otherwise the result is undefined. If the group is empty, returns a NULL handle.

```
$ctx->send($opcode, $row);
```

Constructs a result rowop for the aggregator and arranges for it to be sent to the aggregator's output label. The actual sending is delayed: it will be done only after all the aggregators run. The runs before and after the table modifications are separate. The aggregator's output label is not directly visible in the computation function, so the rowop can not be constructed directly. Instead `send()` takes care of it. The row must be of a type matching the aggregator's result type (and of course the normal practice is to use the aggregator's result type to construct the row). On success returns 1, on error returns `undef` and the error message.

```
$ctx->makeHashSend($opcode, $fieldName => $fieldValue, ...);
```

A convenience method that produces the row from pairs of field names and values and sends it. A combination of `makeRowHash()` and `send()`.

```
$ctx->makeArraySend($opcode, @fields);
```

A convenience function that produces the row from the array of field values and sends it. A combination of `makeRowArray()` and `send()`.

15.8. Opt reference

Triceps::Opt is not a class but a package with a set of functions that help with processing the arguments to the class constructors and other functions when these arguments are represented as options.

```
&Triceps::Opt::parse($class, \%instance, \%optdescr, @opts);
```

Checks the options and copies their values into a class instance (or generally into a hash). Usually used with the class constructors, so the semantics of the arguments is oriented towards this usage. Confesses on errors. `$class` is the calling class name, for the error messages. `\%instance` is the reference to the object instance where to copy the options to. `\%optdescr` is the reference to a hash describing the valid options. `@opts` (all the remaining arguments) are the option name-value pairs passed through from the class constructor.

The entries in `\%optdescr` are references to arrays, each of them describing an option. They are usually written in the form:

```
optionName => [ $defaultValue, \&checkFunc ],
```

If there is no default value, it can be set to `undef`. `\&checkFunc` is a reference to a function that is used to check the option value. If the value is correct, the function returns, if incorrect, it confesses with a descriptive message. The default value is filled in for the missing options before the check function is called. If no checking is needed, the function reference may be `undef`. The check function is called as:

```
&$checkFunc($optionValue, $optionName, $class, $instance);
```

The class and instance are passed through from the arguments of `parse()`.

A user-defined anonymous function can be used to combine multiple checking functions, for example:

```
table => [ undef, sub {
  &Triceps::Opt::ck_mandatory(@_);
  &Triceps::Opt::ck_ref(@_, "Triceps::Table");
} ],
```

A number of ready checking function is provided. When these functions require extra arguments, by convention they go after the common arguments, as shown for `ck_ref()` above.

- `Triceps::Opt::ck_mandatory` checks that the value is defined.
- `Triceps::Opt::ck_ref` checks that the value is a reference to a particular class, or a class derived from it. Just give the class name as the extra argument. Or, to check that the reference is to array or hash, make the argument `"ARRAY"` or `"HASH"`. Or an empty string `""` to check that it's not a reference at all. For the arrays and hashes it can also check the values contained in them for being references to the correct types: give that type as the second extra argument. But it doesn't go deeper than that, just one nesting level. It might be extended later, but for now one nesting level has been enough.
- `Triceps::Opt::ck_refscalar` checks that the value is a reference to a scalar. This is designed to check the arguments which are used to return data back to the caller, and it would accept any previous value in that scalar: an actual scalar value, an `undef` or a reference, since it's about to be overwritten anyway.

The `ck_ref()` and `ck_refscalar()` allow the value to be undefined, so they can safely be used on the truly optional options. When I come up with more of the useful check functions, I'll add them.

```
&Triceps::Opt::handleUnitTypeLabel($caller,
  $nameUnit, \$refUnit,
  $nameRowType, \$refRowType,
  $nameLabel, \$refLabel);
```

A special post-processing that takes care of sorting out the compatibility of the options for the unit, input row type and the input label. Usually called after `parse()`. Confesses on errors.

`$caller` is the description of the caller, for the error messages. The rest are the pairs of the option names and the references to the option values in the instance hash.

Treats the options for input row type and input label as mutually exclusive but with exactly one of them required. If the input row type is used then the unit option is also required. If the input label is used, the unit is optional, but if it's specified anyway, the unit in the option must match the unit of the input label. If the input label is used, the values for the input row type and the unit are extracted from the input label and set into the references.

```
$which = &Triceps::Opt::checkMutuallyExclusive(
  $caller, $mandatory, $optName1, optValue1, ...);
```

Checks a set of mutually exclusive options. Usually called after `parse()`. Confesses on errors, returns the name of the only defined option on success. If no options are defined, returns `undef`.

`$caller` is the description of the caller, for the error messages. `$mandatory` is a flag telling that exactly one of the options must be defined; or the check will confess. The rest are the option name-value pairs (unlike `handleUnitTypeLabel()`, these are values, not references to them).

15.9. Fields reference

Triceps::Fields is a package with a set of functions that help with handling the variable sets of fields in the templates.

```
@fields = &Triceps::Fields::filter(
  $caller, \@inFields, \@translation);
```

Filters and renames the incoming set of fields from `\@inFields` (usually coming from some row type) according to `\@translation`. Returns the array of filtered names, positionally matching the names in the original array. When some field gets thrown away by filtering, its entry in the array will be `undef`. Confesses on errors. `$caller` is the caller's description for the error messages.

See the description of the translation format in Section 10.7: “Result projection in the templates” (p. 128) .

```
@pairs = &Triceps::Fields::filterToPairs(
  $caller, \@inFields, \@translation);
```

Performs the same actions as `filter()` but returns the result in a different format: an array of pairs of field names, where the old field name is paired with the new one. The field names that gets thrown away by filtering do not appear in the result array.

```
($rowType, $projectFunc) = &Triceps::Fields::makeTranslation(
  $optName => $optValue, ...);
```

Generates and compiles a function that performs the filtering of rows and creates the rows of the filtered type (a “projection” in SQL terms). It accepts multiple input row types, each with its own translation specification, and creates the result row type by combining them all. Returns two elements: the result row type and the reference to the compiled function. The function can then be called to perform the projection and combining of the original rows:

```
$resultRow = &$projectFunc($origRow1, $origRow2, ..., $origRowN);
```

If some of the original rows are not available, they may be passed as `undef`. The options are:

`rowTypes`
: Reference to an array of row types for the original rows.

`filterPairs`
: Reference to an array of arrays returned by filterPairs() for the original rows. Obviously, each of the original rows requires its own filter. The sizes of “rowTypes” and “filterPairs” arrays must match. The field names in the results must not have any duplicates.

`saveCodeTo`
: Optional. Reference to a scalar where to save the auto-generated source code of the projection function, for debugging.

```
$result = &Triceps::Fields::isArrayType($typeName);
```

Checks whether a simple type is represented in Perl as an array. Since `uint8[]` is represented as a string, it will return 0.

```
$result = &Triceps::Fields::isStringType($typeName);
```

Checks whether a simple type is represented in Perl as a string. `string`, `uint8` and `uint8[]` will return 1.

15.10. LookupJoin reference

LookupJoin receives the incoming rows and looks up the matches for them from a table, producing the joined rows.

```
$joiner = Triceps::LookupJoin->new(optionName => optionValue, ...);
```

Constructs the LookupJoin template. Confesses on any errors. The options are:

`unit`
: Scheduling unit object where this template belongs. May be skipped if “leftFromLabel” is used.

`name`
: Name of this LookupJoin object. Will be used as a prefix to create the names of internal objects. The input label will be named “name.in” and the output label “name.out”.

`leftRowType`
: Type of the rows that will be coming in at the left side of the join, and will be used for lookup. Mutually exclusive with “leftFromLabel”, one must be present.

`leftFromLabel`
: Source of rows for the left side of the join; implies their type and the scheduling unit where this object belongs. Mutually exclusive with “leftRowType”, one must be present.

`rightTable`
: Table object where to do the look-ups.

`rightIdxPath`
: Array reference containing the path name of index type in table used for the look-up. The index absolutely must be a Hash (leaf or non-leaf), not of any other kind. Optional. Default: first top-level Hash index type.

`leftFields`
: Reference to an array of patterns for the left-side fields to pass through. Syntax as described in `Triceps::Fields::filter()`. Optional. If not defined then pass everything.

`rightFields`
: Reference to an array of patterns for the right-side fields to pass through. Syntax as described in `Triceps::Fields::filter()`. Optional. If not defined then pass everything (which is probably a bad idea since it would include the second copy of the key fields, so better override at least one of the “leftFields” or “rightFields”).

`fieldsLeftFirst`
: Flag: in the resulting rows put the fields from the left side first, then from right side. If 0, then opposite. Optional. Default: 1.

`fieldsMirrorKey`
: Flag: even if the join is an outer join and the row on one side is absent, when generating the result row, the key fields in it will still be present by mirroring them from the other side. Used by JoinTwo. Optional. Default: 0.

`by`
: Reference to an array containing pairs of field names used for look-up, `[leftFld1, rightFld1, leftFld2, rightFld2, ...]`. The set of right-side fields must match the keys of the index path from the option “rightIdxPath”, though possibly in a different order. Mutually exclusive with “byLeft”, one must be present.

`byLeft`
: Reference to an array containing the patterns in the syntax of `Triceps::Fields::filter()`. It gets applied to the left-side fields, the fields that pass through become the key fields, and their translations are the names of the matching fields on the right side. The set of right-side fields must match the keys of the index path from the option `rightIdxPath`, though possibly in a different order. Mutually exclusive with “by”, one must be present.

`isLeft`
: Flag: 1 for left outer join, 0 for inner join. Optional. Default: 1.

`limitOne`
: Flag: 1 to return no more than one row even if multiple rows have been found by the look-up, 0 to return all the found matches. Optional. Default: 0 for the non-leaf right index, 1 for leaf right index. If the right index is leaf, this option will be always automatically set to 1, even if the user specified otherwise, since there is no way to look up more than one matching row in it.

`automatic`
: Flag: 1 means that the manual `lookup()` method will never be called. This allows to optimize the label handler code and always take the opcode into account when processing the rows. 0 means that `lookup()` will be used. Optional. Default: 1.

`oppositeOuter`
: Flag: 1 for the right outer join, 0 for inner join. If both options “isLeft” and “oppositeOuter” are set to 1, then this is a full outer join. If set to 1, each update that finds a match in the right table, may produce a DELETE-INSERT sequence that keeps the state of the right or full outer join consistent. The full outer or right outer join logic makes sense only if this LookupJoin is one of a pair in a bigger JoinTwo object. Each of these LookupJoins thinks of itself as “left” and

of the other one as “right”, while JoinTwo presents a consistent whole picture to the user. Used by JoinTwo. May be used only when “automatic” is 1. Optional. Default: 0.

`groupSizeCode`

Reference to a function that would compute the group size for this side's table. Optional, used only when “oppositeOuter” is 1.

The group size together with the opcode is then used to decide if a DELETE-INSERT sequence needs to be produced instead of a plain INSERT or DELETE. It is needed when this side's index (not visible here in LookupJoin but visible in the JoinTwo that envelopes it) is non-leaf, so multiple rows on this side may match each row on the other side. The DELETE-INSERT pair needs to be generated only if the current rowop was a deletion of the last matching row or insertion of the first matching row on this side. If “groupSizeCode” is not defined, the DELETE-INSERT pair is always generated (which is appropriate if this side's index is leaf, and every row is the last or first one). If “groupSizeCode” is defined, it should return the group size in the left table by the left index for the input row. If the operation is INSERT, the size of 1 would mean that the DELETE-INSERT pair needs to be generated. If the operation is DELETE, the size of 0 would mean that the DELETE-INSERT pair needs to be generated. Called as:

```
&$groupSizeCode($opcode, $leftRow)
```

The default undefined “groupSizeCode” is equivalent to

```
sub { &Triceps::isInsert($_[0]); }
```

but leaving it undefined is more efficient since allows to hardcode this logic at compile time instead of calling the function for every rowop.

`saveJoinerTo`

Reference to a scalar where to save a copy of the joiner function source code. Optional.

```
@rows = $joiner->lookup($leftRow);
```

Looks up the matches for the `$leftRow` and return the array of the result rows. If the option “isLeft” is 0, the array may be empty. If the option “limitOne” is 1, the array will contain no more than one row, and may be assigned directly to a scalar. May be used only when the option “automatic” is 0.

```
$rt = $joiner->getResultRowType();
```

Returns the row type of the join result.

```
$lb = $joiner->getInputLabel();
```

Returns the input label of the joiner. The rowops sent there will be processed as coming on the left side of the join. The result will be produced on the output label.

```
$lb = $joiner->getOutputLabel();
```

Returns the output label of the joiner. The results from processing of the input rowops come out here. Note that the results of the `lookup()` calls do not come out at the output label, they are only returned to the caller.

```
$res = $joiner->getUnit();
$res = $joiner->getName();
$res = $joiner->getLeftRowType();
$res = $joiner->getRightTable();
$res = $joiner->getRightIdxPath();
$res = $joiner->getLeftFields();
$res = $joiner->getRightFields();
$res = $joiner->getFieldsLeftFirst();
$res = $joiner->getFieldsMirrorKey();
$res = $joiner->getBy();
$res = $joiner->getByLeft();
$res = $joiner->getIsLeft();
```

```
$res = $joiner->getLimitOne();
$res = $joiner->getAutomatic();
$res = $joiner->getOppositeOuter();
$res = $joiner->getGroupSizeCode();
```

Get back the values of the options use to construct the object. If such an option was not set, returns the default value, or the automatically calculated value. Sometimes an automatically calculated value may even override the user-specified value. There is no way to get back "leftFromLabel", it is discarded after the LookupJoin is constructed and chained.

15.11. JoinTwo reference

JoinTwo is a template that joins two tables. As the tables are modified, the updates propagate through the join. The join itself keeps no state (other than the state of its input tables), so if it needs to be kept, it has to be saved into another table. There is no requirement of a primary key on either the input tables nor the join result. However if the result is saved into a table, that table would have to have a primary key, so by extension the join would have to produce the result with a primary key, or the table contents will become incorrect. The JoinTwo is internally implemented as a pair of LookupJoins.

```
$joiner = Triceps::JoinTwo->new(optionName => optionValue, ...);
```

Creates the JoinTwo object. Confesses on any errors. The options are:

`name`
: Name of this object. Will be used to create the names of internal objects.

`leftTable`
: Table object to join, for the left side. Both tables must be of the same unit.

`rightTable`
: Table object to join, for the right side. Both tables must be of the same unit.

`leftFromLabel`
: The label from which to receive the rows on the left side. Optional. Default: the Output label of "leftTable" unless it's a self-join; for a self-join the Pre label of "leftTable".

  Can be used to introduce a label that would filter out some of the input. THIS IS DANGEROUS! To preserve consistency, always filter by the key field(s) only, and apply the same condition on the left and right.

`rightFromLabel`
: The label from which to receive the rows on the right side. Optional. Default: the Output label of "rightTable".

  Can be used to introduce a label that would filter out some of the input. THIS IS DANGEROUS! To preserve consistency, always filter by the key field(s) only, and apply the same condition on the left and right.

`leftIdxPath`
: An array reference containing the path name of an index type in the left table used for look-up. The index absolutely must be a Hash (leaf or not), not of any other kind. The number and order of key fields in the left and right indexes must match, since indexes define the fields used for the join. The types of key fields have to match exactly unless the auto-casting is allowed by the option "overrideKeyTypes" being set to 1.

`rightIdxPath`
: An array reference containing the path name of an index type in the right table used for look-up. The index absolutely must be a Hash (leaf or not), not of any other kind. The number and order of key fields in the left and right indexes must match, since indexes define the fields used for the join. The types of key fields have to match exactly unless the auto-casting is allowed by the option "overrideKeyTypes" being set to 1.

`leftFields`
: Reference to an array of patterns for the left-side fields to pass through to the result rows, with the syntax of `Triceps::Fields::filter()`. Optional. If not defined then pass everything.

`rightFields`

Reference to an array of patterns for the right-side fields to pass through to the result rows, with the syntax of `Triceps::Fields::filter()`. Optional. If not defined then pass everything.

`fieldsLeftFirst`

Flag: if 1, in the result rows put the fields from the left side first, then from the right side; if 0, then in the opposite order. Optional. Default: 1.

`fieldsUniqKey`

Controls the logic that prevents the duplication of the key fields in the result rows (since by definition their originals are present in both the left and right tables). Optional.

This is done by setting the option “fieldsMirrorKey” of the underlying LookupJoins to 1 and by manipulating the left/rightFields options: one side is left unchanged, and thus lets the user pass the key fields as usual, while the other side gets `'!key'` specs prepended to the front of it for each key field, thus blocking these fields and removing the duplication.

The enumerated values of this option are one of:

“none”

Do not change either of the “left/rightFields”, and do not enable the key mirroring at all.

“manual”

Enable the key mirroring; do not change either of the “left/rightFields”, leaving the full control to the user.

“left”

Enable the key mirroring; do not change “leftFields” (and thus pass the key fields in there), block the keys from “rightFields”.

“right”

Enable the key mirroring; do not change “rightFields” (and thus pass the key fields in there), block the keys from “leftFields”.

“first”

The default value. Enable the key mirroring; do not change whatever side goes first according to the option “fieldsLeftFirst” (and thus pass the key in there), block the keys from the other side.

`by`

Reference to an array containing pairs of field names used for look-up, `[leftFld1, rightFld1, leftFld2, rightFld2, ...]`. Optional. The options “by” and “byLeft” are mutually exclusive. If none of them is used, by default the field lists are taken from the index type keys, matched up in the order they appear in the indexes. But if a different order is desired, this option can be used to override it. The fields must still be the same, just the order may change.

`byLeft`

Reference to an array containing the patterns in the syntax of `Triceps::Fields::filter()`. It gets applied to the left-side fields, the fields that pass through become the key fields, and their translations are the names of the matching fields on the right side. Optional. The options “by” and “byLeft” are mutually exclusive. If none of them is used, by default the field lists are taken from the index type keys, matched up in the order they appear in the indexes. But if a different order is desired, this option can be used to override it. The fields must still be the same, just the order may change.

`type`

The type of join from the inner/outer classification, one of: “inner”, “left” for left outer, “right” for right outer, “outer” for full outer. Optional. Default: “inner”.

`leftSaveJoinerTo`

Reference to a scalar where to save a copy of the joiner function source code for the left side. Optional.

`rightSaveJoinerTo`
: Reference to a scalar where to save a copy of the joiner function source code for the right side. Optional.

`overrideSimpleMinded`
: Flag: if 1, do not try to create the correct DELETE-INSERT sequence for the updates, just produce the rows with the same opcode as the incoming ones. The only possible usage of this option might be to simulate the CEP systems that do not support the opcodes and treat averything as an INSERT. The data produced is outright garbage. It can also be used for the entertainment value, to show, why it's garbage. Optional. Default: 0.

`overrideKeyTypes`
: Flag: if 1, allow the key field types to be not exactly the same. Optional. Default: 0.

```
$rt = $joiner->getResultRowType();
```

Returns the row type of the join result.

```
$lb = $joiner->getOutputLabel();
```

Returns the output label of the joiner. The results from processing of the input rowops come out here. Note that there is no input label, the join is fed by connecting to the tables (with the possible override with the options “left/rightFromLabel”).

```
$res = $joiner->getUnit();
$res = $joiner->getName();
$res = $joiner->getLeftTable();
$res = $joiner->getRightTable();
$res = $joiner->getLeftIdxPath();
$res = $joiner->getRightIdxPath();
$res = $joiner->getLeftFields();
$res = $joiner->getRightFields();
$res = $joiner->getFieldsLeftFirst();
$res = $joiner->getFieldsUniqKey();
$res = $joiner->getBy();
$res = $joiner->getByLeft();
$res = $joiner->getType();
$res = $joiner->getOverrideSimpleMinded();
$res = $joiner->getOverrideKeyTypes();
```

Get back the values of the options use to construct the object. If such an option was not set, returns the default value, or the automatically calculated value. Sometimes an automatically calculated value may even override the user-specified value. There is no way to get back “left/rightFromLabel”, they are discarded after the JoinTwo is constructed and chained.

15.12. Collapse reference

The Collapse template collapses multiple sequential modifications per primary key into one. On flush it sends out that single modification.

```
$collapse = Triceps::Collapse->new($optName => $optValue, ...);
```

Creates a new Collapse object. Confesses on errors. The options are:

`name`
: Name of this object. Will be used to create the names of internal objects.

`unit`
: The unit where this object belongs.

`data`
: The data set description. Each data set has an input label and an output label, and collapses one stream of modifications. Currently only one data set is supported, the options have been structured like this to allow for the future extension. This option's value is a reference to an array (not a hash!) that is itself structured as the nested option-value pairs.

The nested options in “data” are:

`name`
: The name of the data set. Used for the error messages. Put it first, this would let the constructor report nicely the errors in the other data set options.

`rowType`
: The row type of the data in this set. Mutually exclusive with “fromLabel”, one must be used.

`fromLabel`
: The source label for the data set, its input will be chained to this label. Mutually exclusive with “rowType”, one must be used.

`key`
: The primary key of the data. A reference to an array of strings with field names, same as for the Hash index type.

```
$collapse->flush();
```

Sends out the collected modifications to the output label(s) and clears the state of the collapse.

```
$lb = $collapse->getInputLabel($setName);
```

Returns the input label of a data set. Confesses if there is no data set with this name.

```
$lb = $collapse->getOutputLabel($setName);
```

Returns the output label of a data set. Confesses if there is no data set with this name.

```
@setNames = $collapse->getDatasets();
```

Returns the names of all the data sets (though since currently only one data set is supported, only one name will be returned).

Chapter 16. Release Notes

16.1. Release 1.1.0

- Documentation for the C++ API.
- Streaming functions.
- No more copy trays in the tables, they got replaced by treating the table as a streaming function (the automatically generated FnReturn).
- When a hashed index type is initialized, its match() method takes the field name-to-index translation into account, and matches even if the key field names are different but translating to the same indexes.
- The recusrion is now permitted, and the limits on it can be adjusted per-unit (the defaults still forbid it). The labels can be marked non-re-entrant to forbid the recursion on them.
- The execution of rowops enqueued by fork() and loopAt() has changed: now they reuse the parent's stack frame. This changed the looping logic: the marks are now set on the current, not parent's frame, and the makeLoop* calls don't need, don't create and don't return the begin label.
- The change in the execution of forked rowops led to the different trace sequence, and a modified set of TraceWhen states. The “before” and “after” states now always come in pairs, and there are methods to generally differentiate between them.
- The Labels are marked as cleared before their subclass clearing function is called, not after it. The repeated calls to clear them are ignored.
- The Unit ignores the attempts to remember, forget or clear labels while it's already clearing labels.
- Added `Table::clear()`.
- Added handling of the broken Perl versions that return spurious errors on the command execution.
- Better C++ NameSet constructors.
- The C++ API always throws Exceptions instead of direct abort().
- In the C++ API AggregatorType accept the NULL result row type until the initialization is completed.
- In C++ added a Rhref constructor directly from FdataVec.
- In C++ added the method `Label::adopt()`, making it easier to remember.

16.2. Release 1.0.1

- Fixed the version information that was left incorrect, as 0.99.
- Added the scripts to check the version and Perl MANIFEST before doing a release, script to set the version (ckversion, setversion), explicit version option -v to mkrelease.
- Added the Release Notes.

16.3. Release 1.0.0

- The first official release with full documentation.

- Many additional examples, code clean-ups and small features resulting from the experience of writing the documentation.

16.4. Release 0.99

- The first published pre-release. Basic functionality, no documentation.

Bibliography

[Babkin10] BABKIN, Sergey A. *The practice of parallel programming*. Createspace, ©2010. ISBN 1-451-53661-5.

[Esper] ESPERTECH INC.. *Esper Tutorials: http://esper.codehaus.org/tutorials/tutorials.html* .

[Hyvonen86] HYVÖNEN, Eero and SEPPÄNEN, Jouko. *Lisp-maailma: Johdatus kieleen ja ohjelmointiin. (Lisp World: Introduction to Language and Programming).: Russia Edition: Mir Lispa* . Kirjayhtymä, ©1986. ISBN 9512627876.

[Stayton07] STAYTON, Bob. *DocBook XSL: The Complete Guide (4th Edition): http://www.sagehill.net/docbookxsl/* . Sagehill Enterprises, ©2007. ISBN 0-974-15213-7.

[StreamBase] STREAMBASE INC.. *StreamBase Documentation: http://docs.streambase.com/* .

[Aleri] SYBASE INC.. *Sybase Aleri Streaming Platform 3.2: http://infocenter.sybase.com/help/index.jsp?docset=/com.sybase.infocenter.help.aleri.3.2/title.htm&docSetID=1733* .

[Coral8] SYBASE INC.. *Sybase CEP Option R4: http://infocenter.sybase.com/help/index.jsp?docset=/com.sybase.infocenter.help.cep.4.0/doc/html/title.html&docSetID=1659* .

[SybaseR5] SYBASE INC.. *Sybase Event Stream Processor 5.0: http://infocenter.sybase.com/help/index.jsp?docset=/com.sybase.infocenter.help.esp.5.0/doc/html/title.html&docSetID=1788* .

[Walsh99] WALSH, Norman and MUELLNER, Leonard. *DocBook: The Definitive Guide: http://www.oasis-open.org/docbook/documentation/reference/html/* . O'Reilly Media, ©1999. ISBN 156592-580-7.

Index

Colophon

This manual has been typeset using the Docbook tools.