Complex Event Processing with Triceps CEP v2.0

Developer's Guide

Sergey A. Babkin

Complex Event Processing with Triceps CEP v2.0 : Developer's Guide

Sergey A. Babkin

Copyright © 2014 Sergey A. Babkin

Table of Contents

List of Figures

Preface

1. About the manual

Before starting on the subject of the Triceps CEP itself, I want to tell some things about the organization of this manual.

It had grown quite large, and if it were printed on paper, I would have divided it into at least three volumes. But in the electronic form it's more convenient as a single document, this way the cross-references between any parts of it work seamlessly.

The manual keeps living and growing together with Triceps itself. As things change in Triceps, they change in the manual, but sometimes it's difficult to track down and update all the mentions of the changed subject. I've been spending a huge effort on tracking all such instances down but sometimes things slip through. Keep this in mind and don't be too scared when some paragraph says something contradictory.

A known issue with this manual is that it tends to describe the subjects in the bottom-up fashion, starting from the low-level details and then building up to the high-level concepts. This is partially because the manual has been growing together with Triceps, which is being built from the ground up. And partially it's because I like the details. When I read about a product, I want to understand, how exactly it works. When I write, I want to convey this information. I rewrote some of the chapters to put the high-level descriptions up front. But it's a huge work that will take some time to complete for the whole manual. In the meantime, I'd rather not delay the releases for it, they've been already slowed a lot by the documentation work. So it will get better with time, and in the meantime, if you feel that some details are too much for you, feel free to skip over them.

There are great many other improvements that can be done to the manual, and they will eventually be done. But my take on it is that it's better to have an imperfect manual now than a perfect one in some distant future. It had already been too long in the works, writing the manual for the version 2.0 had taken a whole year.

2. Some concepts

When talking about the CEP programs, I often use the term "model". What is a model? It's basically a CEP program. And more about the models and about what is the CEP itself is described in Chapter 1 .

Many of the examples are built around the world of stock trading. In the modern times almost everyone is probably familiar with the basics of this area. But if case if you're not, let me tell the most fundamental thing needed for understanding the examples: what is a *symbol*.

When the stock shares of some company are traded on an exchange, this company gets assigned a short identifier. This identifier is known as the stock symbol for this company. This word is also often used to mean not just the identifier but also the shares denoted by it. If a company has multiple classes of shares, each class would have its own symbol. And if a company is traded on multiple exchanges, each exchange may have its own identifier for its shares. The options and other derivative financial products also have their own symbols.

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 spreadsheet 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.

Let me elaborate a little on the point of "dragging around all your fields". Here is a typical example: you have a stream of data and you want to pass through only the rows that find a match in some reference table. Which is reasonable to do with something like:

```
insert into filtered_data
select
  incoming_data.*
from
  incoming_data as d left join reference_table as r
  on d.key_field = r.key_field;
```

Only you can't write `incoming_data.*` in their syntax, you have to list every single field of it explicitly. If the data has 90 fields, that becomes quite a drag.

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;

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

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

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

What happens there? First, we import the Triceps module. Then we create a Triceps execution unit. An execution unit keeps the Triceps context and controls the execution for one logical thread.

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 Triceps methods will confess. To be precise, call `Carp::confess`, which is like Perl's `die` but also prints the stack trace. Triceps also includes its own high-level call stack into this trace.

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. The "label" is the Triceps term for the same kind of stream processing elements as in the other CEP systems. The Coral8 term for the same concept is "stream". The "SQLy vs procedural" example in Section 1.4: "We're not in 1950s any more, or are we?" (p. 3) shows why these elements are analogs of labels in the procedural programming, and Triceps generally follows the procedural terminology.

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 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. Thus a Triceps label is still much more like a GOTO label.

Triceps has the streaming functions too, where the caller does provide the way to return the result. These are more than the ordinary labels.

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. Since the row operation object is also immutable, a reference to a row operation may be kept and reused again and again.

Triceps has the explicit operation codes, very much like Aleri/Sybase R5 (only Aleri doesn't differentiate between a row and row operation, every row there has an opcode in it). 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.

This is a very simple example, so it does only one call. The real Triceps programs get a stream of incoming data, and do the calls to handle each row of it.

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 and CPAN. The CPAN location is:

http://search.cpan.org/~babkin/triceps/

The Developer's Guide can also be found in the Kindle format on Amazon web site, for the Amazon's minimal price of $1.

The release policy of Triceps is aimed towards the ease of 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 http://svn.code.sf.net/p/triceps/code/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. If you do have a login and SSH key, you can use then as well:

```
svn co svn+ssh://your_username@svn.code.sf.net/p/triceps/code/trunk
```

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, and the testing reports from CPAN show that it usually works.

The build should work on the other Unix environments too but may require some manual configuration for the available libraries. The test reports from CPAN show that the BSD varieties (FreeBSD, OpenBSD, MidnightBSD) usually do well.

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 older GNU compiler 4.1 and the newer compiler versions have been reported to work as well. But if you build the trunk code checked out from SVN (or otherwise in the directory named "`trunk`"), there is a catch with the warning flags. This kind of build treats almost all warnings as errors, and this causes varying results with the different compiler versions. The version 4.1 doesn't have the option `-Wno-sign-conversion` and will fail on it. The newer compiler versions may have some extra warnings that will be treated as errors (and since my reference compiler doesn't check for them, the code may trigger them). The fix for this situation is to edit `cpp/Makefile.inc` and change the variable `CFLAGS_WARNINGS`, or just clear it altogether. In the release form this is not an issue, in the release directory the warnings are not treated as errors and no warning options are used.

GCC 4.1 is also known to have complaints about the construct `sizeof(field)`. I've modified the reported occurrences but more might creep up in the future. If this stops your build, change them to `sizeof(TypeOfField)`.

The tested Perl versions are 5.10.0 and 5.19.0, and should work on any recent version as well. With the earlier versions your luck may vary. The Makefile.PL has been configured to require at least 5.8.0. The older versions have a different threading module and definitely won't work.

The threads support in the Perl interpreter is needed to run the multithreaded API. If your Perl is built without threads, the single-threaded part is still usable but all the tests related to multithreading will fail. The last version of Triceps with no threads support at all is 1.0.1, and it's the last resort if you want to run without threads.

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 on 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 3.10: "Build configuration settings" (p. 17) .

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. Locale dependency

Some of the Perl tests depend on the locale. They expect the English text in some of the error strings received from the OS and Perl, so if you try to run them in a non-English locale, these tests fail.

To work around this issue, I've added `LANG=C` in the top-level Makefile, and when the tests run from there, they use this English locale.

However if you run `make test` directly in the `perl/Triceps` subdirectory, it has no such override (because the Makefile there is built by Perl). If you run the test from there and use a non-English locale, you'd have to set the locale for the command expicitly:

```
LANG=C make test
```

Some of these expected messages might also change between different OSes and between different versions of Perl. They seem pretty stable overall but you'd never known when something might change somewhere, and that would lead to the spurious failures that can be ignored. I'd be interested to learn of them, to support all known forms of messages in the future.

3.7. 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:

```perl
#!/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.

3.8. 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
```

3.9. 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.

3.10. Build configuration settings

Since Triceps has only a very limited 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. Normally the build attempts to auto-detect the location and name of the library and includes, or otherwise builds without it. Without it the code still works but uses a less efficient implementation of an integer or pointer protected by a mutex. If your system has a version of NSPR4 that doesn't get auto-detected, you can still enable it by changing the settings manually. For example, for Fedora Linux the auto-detected version amounts to the following settings:

```
TRICEPS_CONF += -DTRICEPS_NSPR -I/usr/include/nspr4
TRICEPS_XLIB += -lnspr4
```

`-DTRICEPS_NSPR` tells the code to compile with NSPR support enabled, and the other settings give the location of the includes and of the library.

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

When the Perl API of Triceps detects an error, it makes the interpreter die with an error message. Unless of course you catch it with `eval`. The message includes the call stack as the method `Carp::confess()` would. `confess()` is a very useful method that helps a lot with finding the source of the problem, it's much better than the plain `die()`. Triceps uses internally the methods from Carp to build the stack trace in the message. But it also does one better: it includes the stack of the Triceps label calls into the trace.

You are welcome to use `confess` directly as well, it's typically done in the following pattern:

```
&someFunction() or confess "Error message";
&someFunction() or confess "Error message: $!";
```

This is what the Triceps methods implemented in Perl do. The variable `$!` contains the error messages from the methods that deal with the system errors. To require the package with `confess`, do:

```
use Carp;
```

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.

There also 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.

A known problem with `confess` in a threaded program is that it leaks the scalars, apparently by leaving garbage on the Perl stack, even when intercepted with `eval`. It's actually not a problem when the confession is not intercepted, then the program exits anyway. But if confessing frequently and catching these confessions, the leak can accumulate to something noticeable.

The problem seems to be in the line

```
package DB;
```

in the middle of one of its internal functions. Perhaps changing the package in the middle of a function is not such a great idea, leaving some garbage on the stack. The most interesting part is that this line can be removed altogether, with no adverse effects, and then the leak stops. So be warned and don't be surprised. Maybe it will get fixed.

Now let's look at how the C++ parts of Triceps interact with confessions. When the Perl code inside a label or tracer or aggregator or index sorting handler dies, the C++ infrastructure around it catches the error. It unrolls the stack trace through the C++ code and passes the `die` request to the Perl code that called it. If that Perl code was called through another Triceps C++ code, that C++ code will catch the error and continue unrolling the stack and reporting back to Perl. 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.

More of the error handling details will be discussed later in Section 7.5: "Error handling during the execution" (p. 44) .

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. The memory management at the C++ level is described in more detail in Section 20.3: "Memory management in the C++ API and the Autoref reference" (p. 422) 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. 77) .

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. Code references and snippets

Many of the Triceps Perl API objects accept the Perl code arguments, to be executed as needed. This code can be specified as either a function reference or a string containing the source code snippet. The major reason to accept the arguments in the source code format is the ability to pass them through between the threads, which cannot be done with the compiled code. See more information on that in Section 16.4: "Object passing between threads" (p. 297) .

Only a few of the classes can be exported between the threads but for consistency all the classes support the code arguments in either format. This feature is built into the general way the Triceps XS methods handle the code references.

The following examples are equivalent, one using a function reference, another using a source code snippet. Of course, if you know that the created object will be exported to another thread, you must use the source code format. Otherwise you can take your pick.

```
$it= Triceps::IndexType->newPerlSorted("b_c", undef,
sub {
  my $res = ($_[0]->get("b") <=> $_[1]->get("b")
```

```
    || $_[0]->get("c") <=> $_[1]->get("c"));
  return $res;
}
);

$it= Triceps::IndexType->newPerlSorted("b_c", undef,
'
  my $res = ($_[0]->get("b") <=> $_[1]->get("b")
    || $_[0]->get("c") <=> $_[1]->get("c"));
  return $res;
'
);
```

As you can see, when specifying the handler as source code, you must specify only the function body, and the sub { ... } will be wrapped around it implicitly. Including the sub would be an error.

There are other differences between the code references and the source code format:

When you compile a function, it carries with it the lexical context. So you can make the closures that refer to the "my" variables in their lexical scope. With the source code snippets you can't do this. The source code gets compiled in the context of the main package, and that's all they can see. In some cases, it might not even be compiled immediately. If an object has an explicit initialization, the code snippets get compiled at the initialization time. And if the object is exported to another thread, the code snippets will be re-compiled when an object's copy is created and initialized in that another thread. Remember also that the global variables are not shared between the threads, so if you refer to a global variable in the code snippet and rely on a value in that variable, it won't be present in the other threads (unless the other threads are direct descendants and the value was set before their creation).

The code written in Perl can make use of the source code snippets as well. If it just passes these code arguments to the XS methods, it will get this support automatically. But if it wants to call these snippets directly from the Perl code, Triceps provides a convenience method that would accept the code in either format and compile it if needed:

```
$code = Triceps::Code::compile($code_ref_or_source);
```

It takes either a code reference or a source code string as an argument and returns the reference to the compiled code. If the argument was a code reference, it just passes through unchanged. If it was a source code snippet, it gets compiled (and the rules are the same, the text gets the sub { ... } wrapper added around it implicitly).

If the argument was an undef, it also passes through unchanged. This is convenient in case if the code is optional. But if it isn't then the caller should check for undef.

If the compilation fails, the method confesses, and includes the error and the source code into the message, in the same way as the XS methods do.

The optional second argument can be used to provide information about the meaning of the code for the error messages. If it's undefined then the default is "Code snippet":

```
$code = Triceps::Code::compile($code_ref_or_source, $description);
```

For example, if the code represents an error handler, the call can be done as follows:

```
$code = Triceps::Code::compile($code, "Error handler");
```

4.5. 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 confess. There is also a version of these functions ending with `Safe`:

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

The difference is that it returns `undef` if the input value is out of range, thus being safe from confessions.

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

```
&Triceps::stringOpcode("OP_INSERT")
&Triceps::stringOpcodeSafe("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 either confess without `Safe` in the name or return `undef` with it.

4.6. 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. 30) .

4.7. 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.

4.8. The Perl libraries and examples

The official Triceps classes are collected in the Triceps package (and its subpackages).

However when writing tests and examples I've found that there are also some repeating elements. Initially I've been handling the situation by either combining all examples using such an element into a single file or by copying it around. Then I've collected all such fragments under the package Triceps::X. X can be thought of as a mark of eXperimental, eXample, eXtraneous code.

While the code in the official part of the library is extensively tested, the X-code is tested only in its most important functionality and not in the details. This code is not exactly of production quality but is good enough for the examples, and can be used as a starting point for development of the better code. Quite a few fragments of Triceps went this way: the joins have been done as an example first, and then solidified for the main code base, and so did the aggregation.

One of these modules is Triceps::X::TestFeed. It's a small infrastructure to run the examples, pretending that it gets the input from stdin and sends output to stdout, while actually doing it all in memory. All of the more complicated examples have been written to use it. When you look in the code of the actual running examples and compare it to the code snippets in the manual, you can see the differences. A `&readLine` shows instead of `<STDIN>`, and a `&send` instead of `print` (and for the manual, I have a script that does the reverse substitutions automatically when I insert the code examples into it).

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. 219) .

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!
);
```

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",
);

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

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

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

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

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 confess.

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);
$row  = $rowType->makeRowArray($a, $b, $c);
$row  = $rowType->makeRowHash(%fields);
$row  = $rowType->makeRowHash(a => $a, b => $b);
```

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[]",
);
$row = $rt1->makeRowArray("abcd", [1, 2, 3]);
```

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

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

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

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->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]);
print $row->printP(), "\n";
```

will produce

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

It's possible to check quickly if all the fields of a row are NULL:

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

It returns 1 if all the fields are NULL and 0 otherwise.

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.
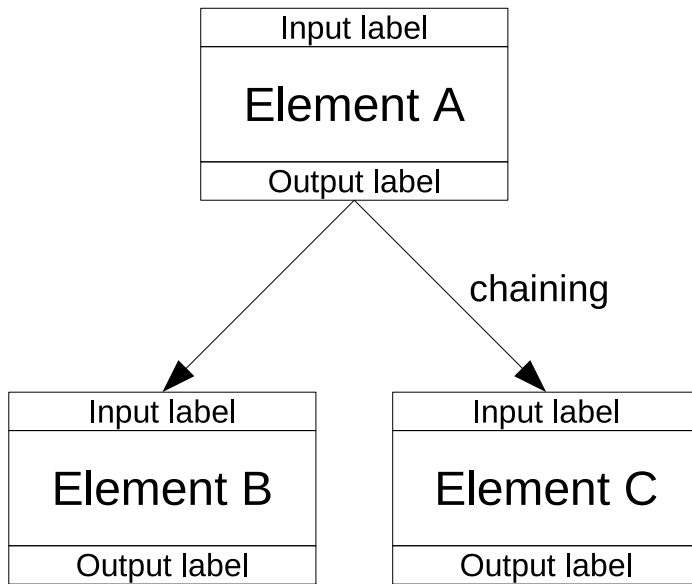
Figure 6.1. Stateful elements with chained labels.

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");
```

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 two arguments provide the references to the Perl functions that perform the actions. They can be specified as a function reference or a source code string, see Section 4.4: "Code references and snippets" (p. 21). `$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.

`$clearSub` is the function that clears the label. It will be explained in the Section 8.2: "Clearing of the labels" (p. 78) Either of `$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. 78)).

```
$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 method:

```
$label1->chain($label2);
```

`$label2` becomes chained to `$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).

When `$label1` executes, its chained labels will normally be executed in the order they were chained. However sometimes it's necessary to add a label to the chain later but have it called first. This is done with the method:

```
$label1->chainFront($label2);
```

It chains `$label2` at the start of the chain. Of course, if more labels will be chained at the front afterwards, `$label2` will be called only after them. But usually there is a need for only one such label, and it's usually connected to the FnReturn and Facet objects. For an example, see Section 16.3: "Multithreaded pipeline" (p. 290) .

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.

There is also a convenience method that creates a new label by chaining it from an existing label:

```
$label2 = $label1->makeChained($name, $subClear, $subExec, @args);
```

The arguments are very much the same as in `Unit::makeLabel()`, only there is no need to specify the row type for the new label (nor obviously the Unit), these are taken from the original label. It's really a wrapper that finds the unit and row type from `label1`, makes a new label, and then chains it off `label1`.

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. To check whether the label has been already cleared use:

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

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 created 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.

And for completeness I'll mention the methods used to mark the label as non-reentrant and to read this mark back. They will be described in detail in Section 7.13: "Recursion control" (p. 74) .

```
$label->setNonReentrant();
$val = $label->isNonReentrant();
```

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. 38) .

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. 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));
  }
});
```

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

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.

The optional argument allows to override the name of the label printed. For example, if in the example above the last line were to be replaced with

```
print $rowop->printP("OtherLabel"), "\n";
```

the result will become:

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

It makes the printing of rowops in the chained labels more convenient. A chained label's execution handler receives the original unchanged rowop that refers to the first label in the chain. So when it gets printed, it will print the name of the first label in the chain, which might be very surprising. The explicit argument allows to override it to the name of the chained label (or to any other value).

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. Introduction to 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.

The Triceps approach to scheduling varied over time. Initially it looked like the purely procedural execution will be enough, with the order determined by the order of the procedural execution, and no explicit scheduling would be needed. This has proved to have its own limitations, and thus the labels and their scheduling were born. Then it had turned out that the most typical thing to do with a label is to call it, again in the purely procedural order.

So for the most part you don't need to think about scheduling in Triceps. It just works as expected: when you call a label with a rowop, the call returns after the label's work is all done. You can pretty much skip over the section with the low-level details altogether, just read the high-level sections. The only important exception is the topological loops, where the rowops go repeatedly through a closed loop of the labels. But even for them the Perl API provides the high-level methods that take care of the details under the hood. And there is another way to deal with the loops by using the streaming functions and procedural loops.

If you want to understand the loop scheduling better, skim over the sections with the details. You'd also need to do this if you plan to write the Triceps models in C++, since as of version 2.0 the C++ API does not provide the high-level methods for building the loops yet.

Only if you are a serious CEP affictionado and want to understand how everything really works, you need to seriously read all the details.

7.2. Comparative scheduling in the various CEP systems

There are multiple approaches to scheduling employed by different CEP systems. The classic Aleri CEP essentially didn't have any, except for the flow control between threads, because each its element is a separate thread. Coral8 had an intricate scheduling algorithm. Sybase R5.1 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 R5.1. 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.

7.3. Execution unit basics

An execution unit (often called simply "unit") keeps the state of the Triceps execution for one thread. Each thread running Triceps must have its own execution unit.

It's perfectly possible to have multiple execution units in the same thread. This is typically done when there is some permanent model plus some small intermittent sub-models created on demand to handle the user requests. These small sub-models would be created in the separate units, to be destroyed when their work is done. But this is a somewhat advanced usage, more examples will be shown in Section 15.11: "Streaming functions and unit boundaries" (p. 281) The TQL implementation also does this, as described in Chapter 17: "*TQL, Triceps Trivial Query Language*" (p. 333) .

This section describes the basic methods of the units, the most often used ones. The more advanced ones are described in the following sections, and the full reference is located in Section 19.1: "Unit and FrameMark reference" (p. 359) .

A unit is created with:

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

The name argument will be used in the error messages, making easier to find, which exact part of the model is having troubles. By convention the name should be the same as the name of the unit variable ("myUnit" in this case).

The name can be read back:

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

Also, as usual, the variable $myUnit 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 (discussed in Section 8.2: "Clearing of the labels" (p. 78) and Section 6.2: "Label construction" (p. 34)), 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 labels are called with:

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

The identity of the label being called is embedded in the row operation. The "..." shows that multiple rowops may be passed as arguments. So the real signature of this method is:

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

But this way it looks more confusing. A call 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.

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

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

The methods for creation of labels have been already discussed in Section 6.2: "Label construction" (p. 34) . 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");

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

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

$table = $unit->makeTable($tableType, "name");

$tray = $unit->makeTray(@rowops);
```

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. The references are at the C++ level, not Perl level.

With the tables, the references 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 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. 77) .

7.4. Trays

The easiest way to store a sequence of 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 new and better approach for catching the results in a tray catches the results of streaming functions.

The trays get created as:

```
$tray = $unit->makeTray(@rowops);
```

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 calling/en-queueing methods, just like the direct rowops (the details of the enqueueing methods will be described later in Section 7.11: "The gritty details of Triceps scheduling" (p. 69) and in Section 19.1: "Unit and FrameMark reference" (p. 359)).

```
$unit->call($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 arguments of these functions, for example:

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

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 found 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 an equivalent of:

```
$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.

7.5. Error handling during the execution

The basics of error handling have been described in Section 4.2: "Errors, deaths and confessions" (p. 19) . Now let's look more in-depth. 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 by default 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, the best way is to use instead schedule() or loopAt() or the streaming functions with trays. That way you avoid overrunning the stack.

It's also possible to relax the recursion checks by specifying higher limits for the recursion count and stack depth. How to do it is described in Section 7.13: "Recursion control" (p. 74). It comes useful in some special cases, as described in Section 15.9: "Streaming functions and recursion" (p. 267). However such higher limits best be avoided unless really needed.

What particular stack is meant here? The execution of Triceps in Perl has three stacks:

• The system stack used by the underlying Triceps C++ code and by the internal functions of the Perl interpreter.

• The Perl call stack, keeping the call history of the Perl code.

• The Triceps call stack, keeping the call history of the Triceps labels in a Unit.

The answer is "all three of these stacks". As the calls are made, frames are pushed onto all these stacks, logically intermingling.

Whichever way the error is detected, it causes the stacks to be unwound, undoing the intermingling in the opposite order. The Perl error messages from die or confess and the Triceps tracing (in the C++ code) of the rowop calls and label chainings get combined into a common stack trace as the stacks are being unwound. When the code gets back to Perl, the XS code triggers a confess with the message containing the unwound stack trace up to this point. If that happens to be in the handler of another label, it continues the hybrid stack unwinding. If not caught by eval, it keeps going to the topmost Triceps Unit call() or drainFrame() and causes the whole program to die, printing the stack trace. In a multithreaded

Triceps model there is also a step of interrupting all the threads in the model, but in the end it still ends up dying and printing the stack trace along with the information, what thread caused it. 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 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 they have a chance to propagate.

In case if the program runs multiple models (multiple Units, or multiple multithreaded Apps) in it, it can also wrap the outermost call in `eval`, and discard just this one erroneous model while leaving the other models running. If the erroneous units get properly cleared, they will free their memory and cause no leaks.

What happens to the rowops that were enqueued 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 out of the stack. The reason is basically that there may be no catching of the errors until unwinding to the outermost call. 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).

7.6. No bundling

The most important principle of Triceps scheduling is: No Bundling. Every rowop is for itself.

I've seen the most damage done by bundling in the Coral8/Sybase R4 scheduling, so I'll refer to it when explaining the dangers of bundling.

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.

Even though the trays in Triceps store multiple rowops, they are not bundles. When a tray is called, it works exactly as if every rowop from it were called separately in order. The first rowop fully propagates, then the second one, and so on. The ordered storage in the trays only provides the order for that future execution or for a manual iteration over the rowops.

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 always be a problem, and even could be occasionally useful, if the bundles were always created explicitly. In the reality of Coral8/Sybase R4 scheduling, every time a statement produces multiple rows from a single one (think of a join that picks multiple rows 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 (the Sybase modeling language), such as "ON UPDATE", had to be designated to always ignore the bundles, otherwise they would not work at all. At my past work 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 rows 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 rows. 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 rows in it. Assuming that nothing runs out of memory along the way, it will be a long time until the client sees the first row. Very, very annoying.

The Aleri CEP also had its own version of bundles, called transactions, but a more smart one. Aleri always relied 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.

7.7. Topological loops

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 it requires that all the rowops to loop over are known in advance. In some situations this might not be true, but instead the rowop entering a loop iteration gets produced by the previous iteration. These situations are better served by the topological loops, formed by connecting the labels in a loop as shown in Figure 7.1 .
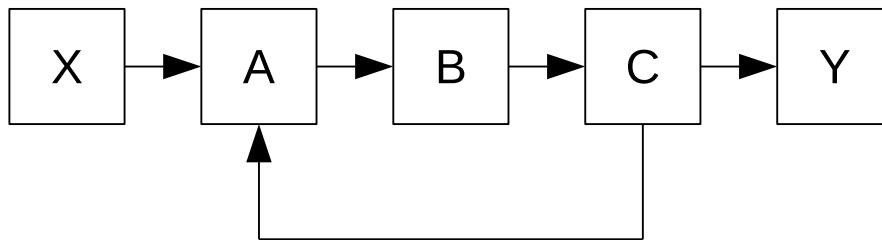
Figure 7.1. Labels forming a topological loop.

However if the labels are simplemindedly doing the calls through a topology like this, the loop becomes a recursion: each label ends up indirectly calling itself for the next iteration of the loop, which repeats the same thing again ang again. This arrangement would quickly use up the stack and crash, so Triceps normally prohibits the recursive calls.

There are two ways to get around that problem. The first one is to use the trays and streaming functions as described in Section 15.5: "Streaming functions and loops" (p. 257). It might be the more powerful alternative of the two, however the concept of streaming functions takes a fair amount of explaining and thus is placed later in the manual. The second way is to use the more advanced scheduling capabilities of the Triceps units, which is described here.

The detailed explanation of how it all works is somewhat complicated, split into a separate section Section 7.12: "The gritty details of Triceps loop scheduling" (p. 72) for those interested. But there are the easy methods that cover up all the complexity.

The first part is done by creating the first label of the loop (such as the label A in Figure 7.1) through a special wrapper. This can be done in one of two ways:

```
my ($lbFirst, $mark) = $unit->makeLoopHead($rowType, "name", $clearSub,
  $execSub, @args);
my ($lbFirst, $mark) = $unit->makeLoopAround("name", $lbToWrap);
```

`makeLoopHead()` is the way to use if you're creating a new Perl label to be the first one in the loop. It has the exact same arguments as `makeLabel()`, which is described in Section 6.2: "Label construction" (p. 34) . It will put an appropriate wrapper directly into the Perl code, that would do all the required magic before your code executes.

`makeLoopAround()` is the way to use if you want to start the loop with some existing label (such as an input label of a table). It will create a new label that does the necessary magic, then chain its argument label from the new one. Nothing really stops you from creating a Perl label manually and then wrapping it in `makeLoopAround()` but `makeLoopHead()` produces a slightly more efficient code.

Either way, two values are returned: the newly created label and a special FrameMark object.

When you send the rows into the loop, you absolutely must send them to this newly created label, **not** directly to the underlying wrapped label! Otherwise the magic won't work.

The FrameMark is a special opaque object that is used to remember the state of the Triceps call stack at the start of the loop, to get back to it on the next iterations. It will be used when sending the rowops to the next iteration of the loop. Naturally, this object must be made accessible in the label handlers that do this sending.

The name argument will become the name of the created label. The FrameMark object also has a name, useful for diagnostics, that gets created by adding a suffix to the argument: "*name*.mark".

The second part, whenever you need to send a rowop back to the start of the loop, such as in the label C in Figure 7.1 , don't call it but use a special method:

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

This will remember this rowop for the future. When the processing of the current iteration is all done, the scheduler in the unit will pick up the next remembered looped rowop and will feed it into the next iteration, until there are no more remembered rowops. Only after that will the first call of the first label in the loop return to its caller. In Figure 7.1 the said caller will be the label X.

The rowops sent back must always be for the label `$lbFirst`, returned by the `makeLoop*()`.

It's perfectly fine to send multiple rowops back from a single iteration of the loop, each of these rowops will be processed in its own iteration in the order they were sent.

It's also perfectly fine to have the nested loops, as long as each loop uses its own frame mark object and starts from a separate label (add an empty label if needed).

There also are the convenience methods that create a rowop and loop it back in one go, just like `makeHash-Call()`/`makeArrayCall()`:

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

Now with all this knowledge let's write an example. It will compute 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 a 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 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");

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

my $lbPrint = $uFib->makeLabel($rtFib, "Print", undef, sub {
```

```
    print($_[1]->getRow()->get("cur"));
});

my $lbCompute; # will fill in later

my ($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,
  );
});

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

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.

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.

Just to rub it in a bit more, $lbMain itself doesn't get back the result of the computation, because the Triceps call() has no way to return any results. The intermediate states circle through the loop until the computation is completed, and the results are forwarded out of the loop to $lbPrint(). All this time $lbMain sits and waits for its call to complete. After the execution gets back to $lbMain, it knows that $lbPrint() already ran and printed the result, so it prints more detail after it. Another option would be for the loop result label to put the result value into some static variable, letting $lbMain read it and print the whole message in one statement.

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 ($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,
    );
  }
});

($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.

Since both $lbNext and $markFib need to be accessible inside $lbCompute, they are created in advance and become visible in the closure scope. But the values are placed into these variables only after $lbCompute is already defined (since $lbCompute is an argument to build these values).

For the more curious, let's dig a little into what happens inside the makeLoop*() methods. The same effect can be (and in C++ API has to be) achieved by calling the slightly lower-level methods.

The frame mark is created as follows:

```
my $mark = Triceps::FrameMark->new("markName");
```

It has to be remembered and then used in the first label of the loop to remember the state of the Triceps call stack:

```
$unit->setMark($mark);
```

This is normally the first thing done in the first label's handler. Yes, it will be remembered on every iteration of the loop. However the trick of the arrangement is that the call stack will be returned to the same state before each iteration, so on the second and following iterations this call will become a no-op.

The `makeLoop*()` methods just do this for you, their implementation is fairly simple:

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

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

  my $label = $self->makeLabel($rt, $name, $clear, sub {
    $self->setMark($mark);
    &$exec(@_);
  }, @args);

  return ($label, $mark);
}

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

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

  my $lbWrap = $self->makeLabel($rt, $name, undef, sub {
    $self->setMark($mark);
  });
  $lbWrap->chain($lbFirst);

  return ($lbWrap, $mark);
}
```

7.8. The main loop

The examples above had already shown the "main loop", now let's look at it up close and discuss, what and why is it doing. The point of the main loop is to get the execution of the model going: accept some rowops from the outside world, shovel them into the Triceps model and process them, sending some result rowops back into the outside world. The sending back is done from inside the label handlers, so as long as the model runs, nothing else is needed for them.

By the time the program enters the main loop, the model should be all constructed and ready to run. The simplest main loop may look like this:

```
while ($rowop = &readRowop()) { # reads with some user-defined function
  $unit->call($rowop);
}
```

This loop will read the incoming rowops as long as they're available, and call them. When `$unit->call()` returns, the processing of the rowop in the model is done, including all the nested calls it caused.

However there is also a way to request the post-processing. It's somewhat similar to the Tcl concept of "idletasks". An example of post-processing might be the flushing of the output buffer: the normal processing may collect a number of the output rowops in the buffer, and after everything is done, the buffer would be serialized and sent out. This post-processing needs to happen after the initial call returns.

The rowops are scheduled for post-processing with the method:

```
$unit->schedule(@rowops_or_trays);
```

The model keeps a queue of the post-processing requests, and `schedule()` adds to this queue.

However the simplest main loop shown above won't run the postprocessing. The queue would just keep growing. The postprocessing is done by the method

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

It calls all the collected post-processing rowops in order. Their handling may keep scheduling more rowops, and the draining won't stop until all of them are processed. So they should not keep schduling more rowops forever, or the draining will never end. To handle the postprocessing properly, the main loop should be:

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

You can even write it in a slightly different form:

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

In this version the incoming rowop gets added to the queue, and then `drainFrame()` calls it and any of its after-effects. Historically, this has been the intended way but then it had turned out that there is no point in first placing the incoming rowop onto the queue and then reading it from the queue, so calling it directly is slightly more efficient.

What if you decide in some label handler deep in the call tree that now is the good time to run the schduled rowops, similar to Tcl's "update idletasks" and call `drainFrame()`? First of all, this is a very bad idea. The CEP models are usually very sensitive to the particular execution order, and inserting some random rowops in the middle tends to break things. Second, it won't work. It might execute *some* rowops (which ones exactly is a long story, described in Section 7.11: "The gritty details of Triceps scheduling" (p. 69)) but none of the scheduled ones. In short, there is a reason to why the method is called `drainFrame()`: the queue is organized in frames that are pushed stack-wise as the labels are called, and popped after the calls complete. DrainFrame() drains the current frame. Schedule() puts the rowops onto the outermost frame that becomes accessible for draining only when the model is idle.

It is possible to find out whether there are the post-processing rowops scheduled and to run them one by one:

```
while ($rowop = &readRowop()) { # reads with some user-defined function
  $unit->call($rowop);
  while (!$unit->empty()) {
    $unit->callNext();
  }
}
```

But of course a Perl loop is less efficient than the C++ loop in `drainFrame()`.

Another straightforward idea is to read and execute the input as it comes in but delay the post-processing until the input becomes idle, exacly like the Tcl "idletasks" do. Somewhat like this:

```
while (1) {
  if (!$unit->empty()) {
    $rowop = &readRowopNoWait();
    if ($rowop) {
      $unit->call($rowop);
    } else {
      $unit->callNext();
    }
  } else {
```

```
    $rowop = &readRowop();
    last if (!$rowop); # no more input
    $unit->call($rowop);
  }
}
```

It might even be useful sometimes but most of the time this turns out to be nothing but pain. The problem is that the exact order of execution becomes dependent on the timing of the data arrival, and the repeatable testing becomes next to impossible. It's another case of the bundling problem.

If the data arrives bundled with multiple rowops per packet, you have a choice whether to drain the frame after each rowop or after each packet. Which approach is better depends on the needs of the application and on whether the bundling of the rowops into packets is predictable and repeatable. If there are no defined boundaries between packets but the grouping is done simply by timeout or buffer size, such bundles are much better off being broken up into the individual rowops.

Now let's look at yet another aspect: the main loop may need to exit not only when there is no more input available but also after processing some requests. This can be done by adding a global stop flag, with label handlers setting it when they need to request the exit:

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

The examples in this manual tend to read the input data as plain text lines, convert them to rowops and execute. They are simple-minded, so they don't do any error checking, they would just fail randomly on the incorrect input. Their main loop usually goes 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.

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();
}
```

Once again, none of these small examples are production-ready. They have no error handling, and 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.

The multithreaded models have their own special needs for the main loops. These will be discussed in Section 16.6: "Dynamic threads and fragments in a socket server" (p. 304) .

7.9. 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, it's only of an example quality, and thus is located in an X-package. 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. You can import this package as Triceps::X::SimpleServer, its source code found in `lib/Triceps/X/SimpleServer.pm`.

```
package Triceps::X::SimpleServer;

sub CLONE_SKIP { 1; }

our $VERSION = 'v2.0.0';

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

our @ISA = qw(Exporter);

our %EXPORT_TAGS = ( 'all' => [ qw(
  outBuf outCurBuf mainLoop startServer makeExitLabel makeServerOutLabel
) ] );

our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

# For whatever reason, Linux signals SIGPIPE when writing on a closed
# socket (and it's not a pipe). So intercept it.
sub interceptSigPipe
{
  if (!$SIG{PIPE}) {
    $SIG{PIPE} = sub {};
  }
}

# and intercept SIGPIPE by default on import
&interceptSigPipe();
```

The package starts with the usual imports and exports. The CLONE_SKIP is required to make sure that the package interacts properly with the multithreading (any objects of this package won't be cloned into the new threads, and since the cloning tends to not work right anyway, I'm not sure why it's not the default).

Then it intercepts and ignores the SIGPIPE signal for the reasons described in the comment. It's very inconvenient to have your server die on a signal when the other side decides to drop the connection. Any server dealing with sockets on Linux must intercept SIGPIPE. Intercepting it with an empty handler looks like a better idea than ignoring it altogether, to make extra-sure that the writer won't be stuck in that write forever, but perhaps ignoring it would be just as good. The interception is placed into a function which gets called on the package import and can be called again later in case if something else resets the handler to default.

```perl
# 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. Will also trigger the polling to
# actually send the output data to the client's socket.
#
# @param id - the client id, as generated on the client connection
#         (if the client already disconnected, this call will
#         have no effect)
# @param string - the string to write
sub outBuf # ($id, $string)
{
  my $id = shift;
  my $line = shift;
  if (exists $clients{$id}) {
    $outbufs{$id} .= $line;
    # If there is anything to write on a buffer, stop reading from it.
    $poll->mask($clients{$id} => POLLOUT);
  }
}

# Write to the output buffer of the current client (as set in $cur_cli
# by the main loop).
#
# @param string - the string to write
sub outCurBuf # ($string)
{
  outBuf($cur_cli, @_);
}

# Close the client connection. This doesn't flush the ouput buffer,
# so it must be called only after the flush is done, or if the flush
# can not be done (such as, if the client has dropped the connection).
# It does delete all the client-related data.
#
# @param id - the client id, as generated on the client connection
# @param h - the socket handle of the client
sub _closeClient # ($id, $h)
{
  my $id = shift;
  my $h = shift;
  $poll->mask($h, 0);
  $h->close();
  delete $clients{$id}; # OK per Perl manual even when iterating
  delete $inbufs{$id};
  delete $outbufs{$id};
}
```

```
# The server main loop. Runs with the specified server socket.
# Accepts the connections from it, then polls the connections for
# input, reads the data in CSV and dispatches it using the labels hash.
#
# XXX Caveats:
# 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.
#
# @param srvsock - the server socket handle
# @param labels - Reference to the label hash, that contains the
#         mappings used to dispatch the input, in either of formats:
#           name => label_object
#           name => code_reference
#         The input from the clients is parsed as CSV with the 1st field
#         containing the label name.  Then if the looked up dispatch is an
#         actual label, the rest of CSV fields are: the 2nd the opcode, and the rest
#         the data fields in the order of the label's row type. If the
#         looked up dispatch is a Perl sub reference, just the whole input
#         line is passed to it as an argument.
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) {
      no warnings; # or in tests prints a lot of warnings about undefs

      $cur_cli = $id;
      $mask = $poll->events($h);
      if (($mask & POLLHUP) && !defined $outbufs{$id}) {
        # print("Lost client $client_id\n");
```

```perl
      _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 in this block
        chomp $line;
      }
      my @data = split(/,/, $line);
      my $lname = shift @data;
      my $label = $labels->{$lname};
      if (defined $label) {
        if (ref($label) eq 'CODE') {
          &$label($line);
        } else {
          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 or functions 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. As you can see, a row must contain at least the label name and opcode, or the execution will print an error message on the server's standard error and return no response to the client in the socket.

If the dispatch table contains not a label but a simple function reference for some name, the rest of the row is not even parsed, the function gets called without any arguments. If the exit is implemented as a function in the dispatch table, the following would also work:

exit

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

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

The `$id` is the unique id of the client. How do you find, what is the id of the client you want to send the data to? When an input line is processed, the main loop knows, from what client it was received. It puts the id of that client in the global variable `$Triceps::X::SimpleServer::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, and the client disconnects before you call `outBuf()`, the call will have no effect. In any case, if a client has disconnected, the further processing of its requests should usually be stopped, and thus checking if the client is still connected is a good idea anyway:

```
if (exists $clients{$id}) {
  # ... prepare the data for it ...
  &outBuf($id, $text);
} else {
  # ... stop sending the data to this client ...
}
```

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 `$Triceps::X::SimpleServer::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. The creation of the socket and forking of a separate server process is wrapped in another function:

```
# The server start function that creates the server socket,
# remembers its port number, then forks and
# starts the main loop in the child process. The parent
# process then returns the pair (port number, child PID).
#
# @param port - the port number to use; 0 will cause a unique free
#        port number to be auto-assigned
# @param labels - reference to the label hash, to be passed to mainLoop()
# @return - pair (port number, child PID) that can then be used to connect
#        to and control the server in the child process
sub startServer # ($port, $%labels)
{
  my $port = shift;
  my $labels = shift;

  my $srvsock = IO::Socket::INET->new(
    Proto => "tcp",
    LocalPort => $port,
    Listen => 10,
  ) or confess "socket failed: $!";
  # Read back the port, since the port 0 will cause a free port
  # to be auto-assigned.
  $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);
}
```

You can specify the server 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, as found in `t/xQuery.t`:

```
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
```

```
  price => "float64",
  size => "float64", # number of shares traded
);

use Triceps::X::SimpleServer qw(:all);

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 {
  $Triceps::X::SimpleServer::srv_exit = 1;
});

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

my ($port, $pid) = &Triceps::X::SimpleServer::startServer(0, \%dispatch);
print STDERR "port=$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.

`$rtTrade` is the row type for the expected data. 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. Here is a small session log from the client side (46651 is the port that got picked at random and printed by the server on the start):

```
$ telnet localhost 46651
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
echo,OP_INSERT,1,a,2,3.4
echo OP_INSERT id="1" symbol="a" price="2" size="3.4"
echo2,OP_INSERT,1,a,2,3.4
echo,OP_INSERT,1,a,2,3.4
exit,OP_NOP
^]
telnet> q
Connection closed.
```

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.

The exit label was created manually in this example but SimpleServer also provides the functions that create an exit label or an exit function, either of which can be placed into a dispatch table:

```
# A dispatch function, sending anything to which will exit the server.
# The server will not flush the outputs before exit.
#
# Use like:
#   $dispatch{"exit"} = \&Triceps::X::SimpleServer::exitFunc;
#
# In this way the input line doesn't have to contain the opcode.
# The alternative way is through makeExitLabel().
```

```
sub exitFunc # ($line)
{
    $srv_exit = 1;
}

# Create a label, sending anything to which will exit the server.
# The server will not flush the outputs before exit.
#
# Use like:
#    $dispatch{"exit"} = &Triceps::X::SimpleServer::makeExitLabel($uTrades, "exit");
#
# In this way the input line has to contain at least the opcode.
# The alternative way is through exitFunc().
#
# @param unit - the unit in which to create the label
# @param name - the label name
# @return - the newly created label object
sub makeExitLabel # ($unit, $name)
{
  my $unit = shift;
  my $name = shift;
  return $unit->makeLabel($unit->getEmptyRowType(), $name, undef, sub {
    $srv_exit = 1;
  });
}
```

makeExitLabel() is quite simple, it creates a label with hardcoded function of setting the flag $srv_exit. Even its row type is hardcoded to the empty rows. exitFunc() sets the same flag directly.

There is also a function for making the labels that output their rows to the client in CSV format (as usual, no commas in the values, the same as is expected by the socket server):

```
# Create a label that will print the data in CSV format to the server output
# (to the current client).
#
# @param fromLabel - the new label will be chained to this one and get the
#          data from it
# @param printName - if present, overrides the label name printed
# @return - the newly created label object
sub makeServerOutLabel # ($fromLabel [, $printName])
{
  no warnings; # or in tests prints a lot of warnings about undefs
  my $fromLabel = shift;
  my $printName = shift;
  my $unit = $fromLabel->getUnit();
  my $fromName = $fromLabel->getName();
  if (!$printName) {
    $printName = $fromName;
  }
  my $lbOut = $unit->makeLabel($fromLabel->getType(),
    $fromName . ".serverOut", undef, sub {
      &outCurBuf(join(",", $printName,
        &Triceps::opcodeString($_[1]->getOpcode()),
        $_[1]->getRow()->toArray()) . "\n");
    });
  $fromLabel->chain($lbOut);
  return $lbOut;
}
```

makeServerOutLabel() finds the unit and row type from the 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.

Running the automated tests of the servers requires the clients to be started automatically too, feed the input, receive the results, and then compare them to the expected results. The package Triceps::X::DumbClient from `lib/Triceps/X/DumbClient.pm` does exactly that. The server code gets created as usual, only instead of starting the server, the dispatch table is given to the DumbClient method that takes care of starting the server, feeding the input, collecting the results, and waiting for the server to stop.

For example, the same echo example is run like this with DumbClient:

```
use Triceps::X::SimpleServer qw(:all);

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 {
  $Triceps::X::SimpleServer::srv_exit = 1;
});

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

my @inputQuery = (
"echo,OP_INSERT,1,a,2,3.4\n",
"echo2,OP_INSERT,1,a,2,3.4\n",
);
my $expectQuery =
'> echo,OP_INSERT,1,a,2,3.4
> echo2,OP_INSERT,1,a,2,3.4
echo OP_INSERT id="1" symbol="a" price="2" size="3.4"
echo,OP_INSERT,1,a,2,3.4
';

Triceps::X::TestFeed::setInputLines(@inputQuery);
Triceps::X::DumbClient::run(\%dispatch);

ok(&Triceps::X::TestFeed::getResultLines(), $expectQuery);
```

DumbClient works in symbiosis with the TestFeed module that handles the recorded inputs and outputs. Note that the "exit" line is not there, DumbClient adds it implicitly at the end of the input.

The input lines are also included by TestFeed in the output with the "> " prepended to them. DumbClient feeds all the inputs first and then reads all the results, relying on the TCP buffering to avoid deadlocking on the flow control. This works only for the small amounts of input but is good enough for the small tests.

And the implementation of DumbClient is fairly small, there is only one method:

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

  my ($port, $pid) = Triceps::X::SimpleServer::startServer(0, $labels);
```

```
  my $sock = IO::Socket::INET->new(
    Proto => "tcp",
    PeerAddr => "localhost",
    PeerPort => $port,
  ) or confess "socket failed: $!";
  while(& readLine) {
    $sock->print($_);
    $sock->flush();
  }
  $sock->print("exit,OP_INSERT\n");
  $sock->flush();
  $sock->shutdown(1); # SHUT_WR
  while(<$sock>) {
    & send($_);
  }
  waitpid($pid, 0);
}
```

As mentioned before in Section 4.8: "The Perl libraries and examples" (p. 25) , the methods `readLine` and `send` are imported from the TestFeed module.

7.10. 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,

- before the chained labels execute,

- after the chained labels execute,

- after the label executes,

- before the label's frame is drained (and thus the forked rowops execute, see the details of that in Section 7.11: "The gritty details of Triceps scheduling" (p. 69)),

- after the frame is drained.

The calls around the chaining and around the draining are done only if there are the chained labels to call or forked rowops to drain accordingly. Otherwise these pairs are skipped.

The tracing calls happen in the order shown above. The call after the label executes goes after the chained calls (if any), enveloping them. However the draining calls happen after that (and no matter how many rowops were forked onto that frame, there will be only one after-draining call per frame, still referring to the original label).

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

```
$tracer = Triceps::UnitTracerStringName(option => $value, ...);
```

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 on 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.

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");
$c_lab2 = $u1->makeDummyLabel($rt1, "lab2");
$c_lab3 = $u1->makeDummyLabel($rt1, "lab3");

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

$c_lab1->chain($c_lab2);
$c_lab1->chain($c_lab3);
$c_lab2->chain($c_lab3);

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

$u1->drainFrame();
```

The trace is:

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

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

The print-out is not indented because the execution of real models tends to involve some quite long call chains, which would result in some extremely wide indenting. Instead the curly braces at the end of each line help to find the matching pair. You can always use the vi command % to jump to the matching brace, or a similar feature in the other editors.

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 non-verbose trace doesn't have the curly braces because there are no matching pairs of lines.

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);
```

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. A source code string may be used instead of the function reference, see Section 4.4: "Code references and snippets" (p. 21) .

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_AFTER, &Triceps::TW_BEFORE_DRAIN, &Triceps::TW_AFTER_DRAIN, &Triceps::TW_BEFORE_CHAINED, &Triceps::TW_AFTER_CHAINED; 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);
$string = &Triceps::tracerWhenStringSafe($value);
$value = &Triceps::stringTracerWhenSafe($string);
```

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

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

Now that the constants have been mentioned, the order of tracing calls for a single executing rowop on a single label is:

```
TW_BEFORE
TW_BEFORE_CHAINED
TW_AFTER_CHAINED
TW_AFTER
TW_BEFORE_DRAIN
TW_AFTER_DRAIN
```

There is also a general way to find, if the `$when` refers to a "before" or "after" situation:

```
$result = &Triceps::tracerWhenIsBefore($when);
$result = &Triceps::tracerWhenIsAfter($when);
```

Their typical usage in a trace function, to append an opening or closing brace, looks like:

```
if (Triceps::tracerWhenIsBefore($when)) {
  $msg .= " {";
} elsif (Triceps::tracerWhenIsAfter($when)) {
  $msg .= " }";
}
```

More trace points that are neither "before" or "after" could get added in the future, so a good practice is to use an elsif with both conditions rather than a simple if/else with one condition.

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() # unit, label, fromLabel, rop, when, extra
{
  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 (Triceps::tracerWhenIsBefore($when)) {
    $msg .= " {";
```

```
  } elsif (Triceps::tracerWhenIsAfter($when)) {
    $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 collects 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 something different let's make the result indented, with two spaces per indenting level. As mentioned before, the indenting is actually not such a great idea. 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.

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

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

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

  if ($verbose) {
    ${$rnest}++ if (Triceps::tracerWhenIsBefore($when));
  }
}
```

```
undef @history;
my $tnest =  0; # 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' 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' 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' after label 'lab3' (chain 'lab2') op lab1 OP_INSERT
            a="123" b="456" c="789" d="3.14" e="text"  }
      unit 'u1' after-chained label 'lab2' (chain 'lab1') 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' after label 'lab3' (chain 'lab1') op lab1 OP_INSERT
        a="123" b="456" c="789" d="3.14" e="text"  }
  unit 'u1' after-chained label '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' 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' 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' after label 'lab3' (chain 'lab2') op lab1 OP_DELETE
            a="123" b="456" c="789" d="3.14" e="text"  }
      unit 'u1' after-chained label 'lab2' (chain 'lab1') 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' after label 'lab3' (chain 'lab1') op lab1 OP_DELETE
        a="123" b="456" c="789" d="3.14" e="text"  }
  unit 'u1' after-chained label '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.

7.11. The gritty details of Triceps scheduling

There are four ways of executing a rowop in Triceps:

Call:

>   Execute the label right now, including all the nested calls. When the call returns, the execution is completed. This is the most typical way, and the only one described in detail so far.

Schedule:

>   Execute the label after everything else is done.

Fork:

>   Execute the label after the current label returns but before its caller gets the control back or anything else is done. Obviously, if multiple labels are forked, they will execute in the order they were forked. The forked labels can be seen as "little siblings" of the current label. Forking is currently not used much, other than for the special case of looping.

Loop:

>   Execute the label as the start of the next iteration of the topological loop, after the current iteration is fully completed. This is a special case of fork, essentially forking at the level of the loop's first label.

The common term encompassing all of them is "enqueue". "Enqueue" 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 meaning is kind of intuitively straightforward 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 not just a single queue but a stack of queues that contain the rowops to be executed. The rowops get into the queues when they are forked or looped or scheduled. 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 arrive from the outside world, they can be added with the method `schedule()` to that stack frame. That's what `schedule()` does: always adds rowops to the outermost stack frame, no matter how many frames might be pushed on top of it. 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 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 by calling the Unit method `fork()`. 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. Now is the time to execute the rowops from it. It's also called "frame draining" but if works somewhat differently in the case of the forked rowops. The first rowop gets picked from the frame and called, but in a special way. It doesn't get its own frame. Instead, it takes over the frame of its parent rowop. The frame that was marked "~1" now changes its marking to "~5" because of that take-over:

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

If the rowop 5 forks rowop 8, the stack becomes:

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

Since the frame was inherited from the parent rowop 1, the rowop 8 just gets appended to the end of it after rowop 6. The rowops forked in the same frame are executed in the order they were forked. Unlike the calls, there is no nesting involved in forking.

When the execution of rowop 5 returns, the execution of the forked rowops from the innermost frame continues. The rowop 6 gets picked from the front of the frame and takes over the frame ownership:

```
[8] ~6
[2, 3]
```

Suppose the rowop 6 doesn't call or fork anything else and returns. Then the rowop 8 starts executing and takes over the frame:

```
[ ] ~8
[2, 3]
```

Suppose rowop 8 calls `schedule()` of rowop 9. Rowop 9 is then added to the outermost queue:

```
[ ] ~8
```

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

Rowop 8 then returns, its queue is empty, so it's popped and its call completes.

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

The 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? They turn out to be similar to a `fork()`. The presence of chaining gets checked after the original label completes its execution but before executing any of the forked labels from its frame. If any chained labels are found, they are called one by one. They take over the frame of the parent, just like the forked labels. Any of the chained labels may also call `fork()`, adding more labels to the frame. The next forked label (if any) gets executed only after all the labels chained from the current one are done.

What would happen if `drainFrame()` is called not from outside the model but from inside some label handler? It will drain the innermost frame. Suppose that the queue stack was in the following state, with rowop 5 executing:

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

If the label handler of the rowop 5 calls `drainFrame()` now, `drainFrame()` will do its usual job: pick the rowops one by one from the innermost frame, create the nested frames for them and execute. So first it will pick up the rowop 6:

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

After the rowop 6 completes, its frame gets popped:

```
[8] ~5
[2, 3]
```

But `drainFrame()` continues running, and now picks the rowop 8:

```
[ ] ~8
[ ] ~5
[2, 3]
```

After the rowop 8 completes, its frame gets also popped:

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

At this point the innermost frame becomes empty and `drainFrame()` returns. The label handler of rowop 5 continues its execution.

If you haven't forked anything, the innermost frame will be empty, and `drainFrame()` will do nothing. If you did fork some rowops, `drainFrame()` looks like a convenient way to call them now and then continue. However note that in this case the semantics is different from the normal forking. The rowops from the frame will be called in the nested frames, not taking over the original frame. So if say rowop 6 refers to the same label as rowop 5, this nested execution will be considered a recursive call of the same label. Thus `drainFrame()` is best used only with the outermost frame.

What if the rowop 1 weren't scheduled and then drained but was just directly called? The outermost frame will remain empty, while a new frame will be pushed for the rowop 1 as usual:

```
[ ] ~1
[ ]
```

If the rowop 1 executed the same code as before, after the call it will leave the rowop 9 scheduled on the outermost frame:

```
[9]
```

To execute the rowop 9, call `drainFrame()`, or it will be stuck there forever.

Note that the execution order differs depending on whether the incoming rowops were scheduled or directly called, and on when the `drainFrame()` is called. If the three rowops were scheduled and then drained, the execution order will be 1, 2, 3, 9. If they were called directly with draining the frame after each one, the order will be 1, 9, 2, 3. And if they were called directly but with draining only after the last one, it would be again 1, 2, 3, 9.

The loop scheduling is a whole big separate subject that will be discussed in the next section.

7.12. The gritty details of Triceps loop scheduling

Now it's time to look at what is really going on when a topological loop gets executed. Let's continue looking at the loop example that was already shown in Figure 7.1 (page 46) .

If the loop were handled simple-mindedly, with all the execution done by calls, it could 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 called, creating its own execution frame (marked as such for clarity):

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

Which then calls A2:

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

Which then continues the calls in sequence. 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++ and Perl) stack.

Which is why this kind of recursive calls is forbidden by default in Triceps. If you try to do it, on the first recursive call the execution will die with an error. You can enable the recursion but this only lets the stack grow and doesn't prevent the growth.

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 inherits the stack frame and executes, forks B3:

```
[B3] ~A2
[ ]
```

On each step the frame will be inherited by the next label, and if Y8 is also eventually forked, at the end the stack will be:

```
[ ] ~Y8
[ ]
```

Problem solved, no matter how many iterations were done by the loop, the stack will stay limited.

The catch though is that *every* operation inside the loop must be done with a `fork()`. If there is even one `call()` occuring in the loop, the stack will grow by a frame for each `call()` and may become quite deep again. The problem is that `call()` is hardcoded in many primitives, such as Tables, and is fairly typically used in the templates as well. The historic solution for that was to specify for each table, how it should handle its results, call them or fork them or even schedule them. And the templates could use a similar approach.

The practice had quickly showed that not only all this explicit choice is quite cumbersome and easy to miss, but also the semantics of `fork()` is different from `call()` in a very annoying way. If some label wants to do something, call some other label, then do something more using the result of the call, doing it with `call()` is simple: just execute all this procedurally in sequence. After `call()` returns, its work is guaranteed to be done and any global state to be updated. Not so with `fork()` that just puts the rowop onto a queue, there just isn't any way to get the second half of the original label's code to execute only after all the effects from the forked rowop had propagated. (Historically `fork()` worked differently in Triceps 1.0 and did allow to reproduce the call semantics through some minor contortions but then it kept growing the stack on every fork, just as the calls do).

The solution, even back in the version 1.0 days, was to add a special method for the loop scheduling.

It starts with the concept of the frame mark. 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 .
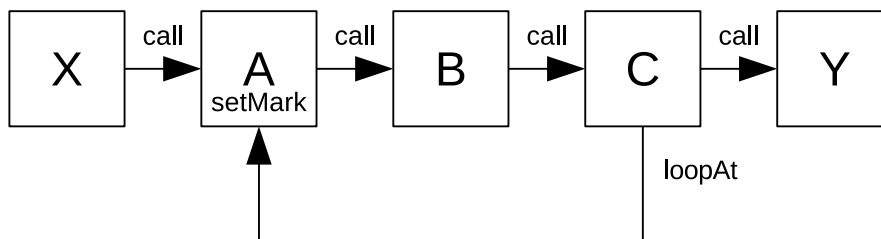
Figure 7.2. Proper calls in a loop.

When the label A executes the rowop A2, first things it does is calling setMark(M). After that the stack will look like this:

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

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

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

The stack then unwinds until A5 starts its execution:

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

When A5 inherits the stack frame from A2, the mark M stays put. The label A would normally call `setMark(M)` again anyway, but it will just put the mark onto the same frame, so effectively it's a no-operation.

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

After Y8 completes, the stack will unroll back, and X1 can continue its execution:

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

To reiterate, when the control returns back to X1, the whole loop is done.

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()`.

It's possible to use this handling of an unset mark to some creative effects. It allows the loops to take 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. In this case the code of label X must be prepared to find that the loop hadn't completed yet after the call of A2 returns. 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 will find that remembered state with B3 and will make 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. When the rowop A5 gets executed, the label A will call `setMark(M)` again, thus setting the mark on its new frame, and making the loop run as far as it can before executing any other scheduled rowops.

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 effects may be quite unexpected.

7.13. Recursion control

Historically, the recursive calls (when a label calls itself, directly or indirectly) have been forbidden in Triceps. Mind you, the recursive calling could still be done even then with the help of trays and forking. And it's probably the best way too from the standpoint of correctness. However it's not the most straightforward way, and the real recursion still comes handy once in a while.

Now the recursion is allowed in its direct way. Especially that it doesn't have to be all-or-nothing, it can be done in a piecemeal and controlled fashion.

It's controlled per-unit. Each unit has two adjustable limits:

Maximal stack depth:
> Limits the total depth of the unit's call stack. That's the maximal length of the call chain, whether it goes straight or in loops.

Maximal recursion depth:
> Limits the number of times each particular label may appear on the call stack. So if you have a recursive code fragment (a simple-minded loop or a recursive streaming function), this is the limit on its recursive reentrances.

Both these limits accept the 0 and negative values to mean "unlimited".

The default is as it has been before: unlimited stack depth, recursion depth of 1 (which means that each label may be called once but it may not call itself). But now you can change them with the calls:

```
$unit->setMaxStackDepth($n);
$unit->setMaxRecursionDepth($n);
```

You can change them at any time, even when the unit is running (but they will be enforced only on the next attempt to execute a rowop).

You can also read the current values:

```
$n = $unit->maxStackDepth();
$n = $unit->maxRecursionDepth();
```

Another thing about the limits is that even if you set them to "unlimited" or to some very large values, there still are the system limits. The calls use the C++ process (or thread) stack and the Perl stack, and if you make too many of them, the stack will overflow and the whole process will crash and possibly dump core. Keeping the call depths within reason is still a good idea.

Now you can do the direct recursion. However as with the procedural code, not all the labels are reentrant. Some of them may work with the static data structures that can't be modified in a nested fashion. Think for example of a table: when you modify a table, it sends rowops to its "pre" and "out" labels. You can connect the other labels there, and react to the table modifications. However these labels can't attempt to modify the same table, because the table is already in the middle of a modification, and it's not reentrant.

The table still has a separate logic to check for non-reentrance, and no matter what is the unit's general recursion depth limit, for the table it always stays at 1. Moreover, the table enforces it across both the input label interface and the procedural interface.

If you make your own non-reentrant labels, Triceps can make this check for you. Just mark the first label of the non-reentrant sequence with

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

It will have its own private recursion limit of 1. Any time it's attempted to execute recursively, it will confess. There is no way to unset this flag: when a label is known to be non-reentrant, it can not suddenly become reentrant until its code is rewritten.

You can read this flag with

```
$val = $label->isNonReentrant();
```

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. 20) . 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. 78) .

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");
my $rtCount = Triceps::RowType->new(
  address => "string",
  count => "int32",
);

my $ttCount = Triceps::TableType->new($rtCount)
  ->addSubIndex("byAddress",
    Triceps::IndexType->newHashed(key => [ "address" ])
  )
;
$ttCount->initialize();

my $tCount = $hwunit->makeTable($ttCount, "tCount");

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,
    );
    $tCount->insert($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, configuring 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, 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");
my $rtCount = Triceps::RowType->new(
  address => "string",
  count => "int32",
);

my $ttCount = Triceps::TableType->new($rtCount)
  ->addSubIndex("byAddress",
    Triceps::IndexType->newHashed(key => [ "address" ])
  )
;
$ttCount->initialize();

my $tCount = $hwunit->makeTable($ttCount, "tCount");

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");
  } );
$tCount->getOutputLabel()->chain($lbPrintCount);

# 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 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. 81)) 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.

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 cause a confession.

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);
$table->insert($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. 101) .

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");
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
);

our $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
      ->addSubIndex("last2",
        Triceps::IndexType->newFifo(limit => 2)
      )
  )
;
$ttWindow->initialize();
our $tWindow = $uTrades->makeTable($ttWindow, "tWindow");

# remember the index type by symbol, for searching on it
our $itSymbol = $ttWindow->findSubIndex("bySymbol");
# remember the FIFO index, for finding the start of the group
our $itLast2 = $itSymbol->findSubIndex("last2");

# 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
  });
$tWindow->getOutputLabel()->chain($lbWindowPrint);

while(<STDIN>) {
  chomp;
  my $rTrade = $rtTrade->makeRowArray(split(/,/));
  my $rhTrade = $tWindow->makeRowHandle($rTrade);
  $tWindow->insert($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);
  }
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2);
  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.

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())
  )
;
```

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");
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
  price => "float64",
  size => "float64", # number of shares traded
);

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)
      )
  )
;
$ttWindow->initialize();
our $tWindow = $uTrades->makeTable($ttWindow, "tWindow");

# remember the index type by symbol, for searching on it
our $itSymbol = $ttWindow->findSubIndex("bySymbol");
# remember the FIFO index, for finding the start of the group
our $itLast2 = $itSymbol->findSubIndex("last2");

# remember, which was the last row modified
our $rLastMod;
our $lbRememberLastMod = $uTrades->makeLabel($rtTrade, "lbRememberLastMod",
  undef, sub { # (label, rowop)
    $rLastMod = $_[1]->getRow();
  });
$tWindow->getOutputLabel()->chain($lbRememberLastMod);

# 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);
  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(/,/);
  $uTrades->makeArrayCall($tWindow->getInputLabel(), @data);
  &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",
);

$it1 = Triceps::IndexType->newPerlSorted("basic", undef, \&compBasic);

$tt1 = Triceps::TableType->new($rt1)
  ->addSubIndex("primary", $it1)
;

$tt2 = Triceps::TableType->new($rt1)
  ->addSubIndex("first", $it1)
;

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` function references (or, as usual, they may be specified as source code strings) 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");
```

This assumes that the row type will have a numeric field "a" and a string field "b". 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. At which point the attempt to get a non-existing field will confess, this error will be caught by the table and set the sticky error in it. The insert operation will confess with this error, and any future operations on the table will also confess with this error (this is the meaning of "sticky"), the table will become unusable.

The $initFunc provides a way to do that check and more up front. 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");
```

The init function can do even better: it can create and set the comparison function. It's done with:

```
$idxt->setComparator($compareFunc);
```

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.

The handling of the fatal errors (as in `die()`) in the initialization and especially comparison functions is an interesting subject. The errors propagate properly through the table, and the table operations confess with the Perl handler's error message. But since an error in the comparison function means that things are going very, very wrong, after that the table becomes inoperative and will die on all the subsequent operations as well. You need to be very careful in writing these functions.

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, ...);
```

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",
    )
  );
```

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");
```

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;

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, '&Triceps::SimpleOrderedIndex::init(@_)', undef, @args
  );
  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 = ""; # 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";

  if (defined $errors) {
    # help with diagnostics, append the row type to the error listing
    $errors .= "the row type is:\n";
    $errors .= $rowt->print();
  } else {
    # set the comparison as source code
    #print STDERR "DEBUG Triceps::SimpleOrderedIndex::init: comparison function:\n$compare
\n";
    $idxt->setComparator($compare);
  }
  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]->get("a") cmp $_[1]->get("a")
  || $_[1]->get("c") <=> $_[0]->get("c")
  || $_[0]->get("b") cmp $_[1]->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 .
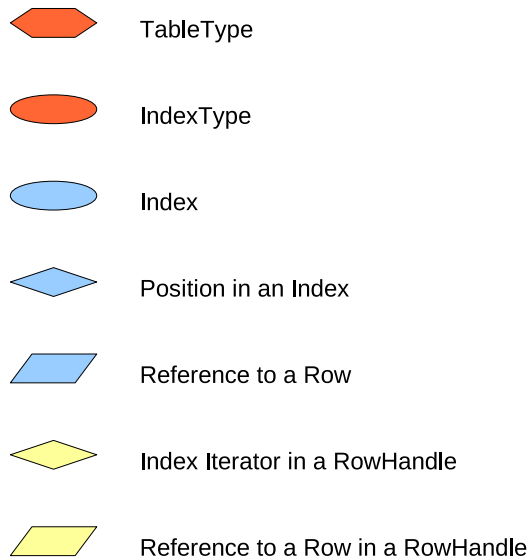
Figure 9.1. Drawings legend.

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.
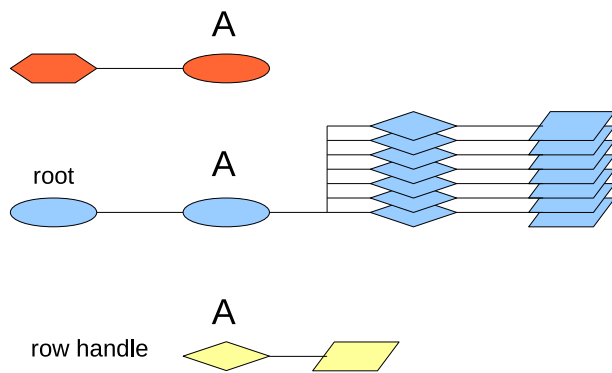
Figure 9.2. One index type.

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"
```
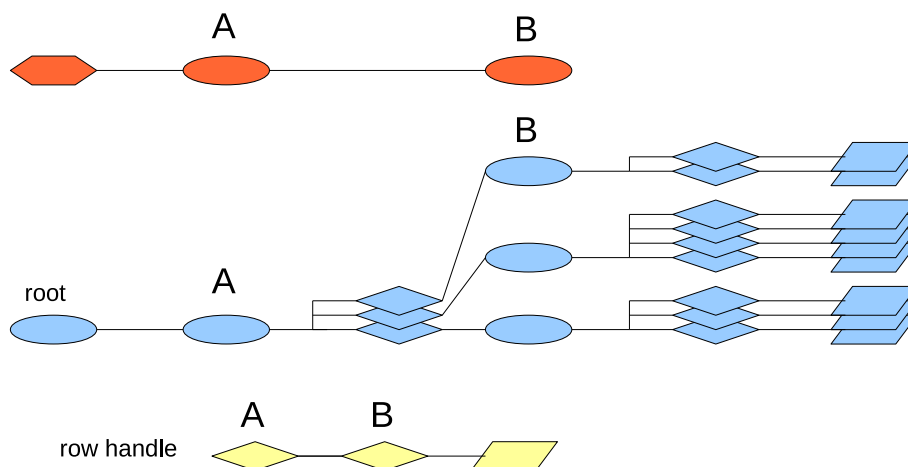
Figure 9.3. Straight nesting.

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.
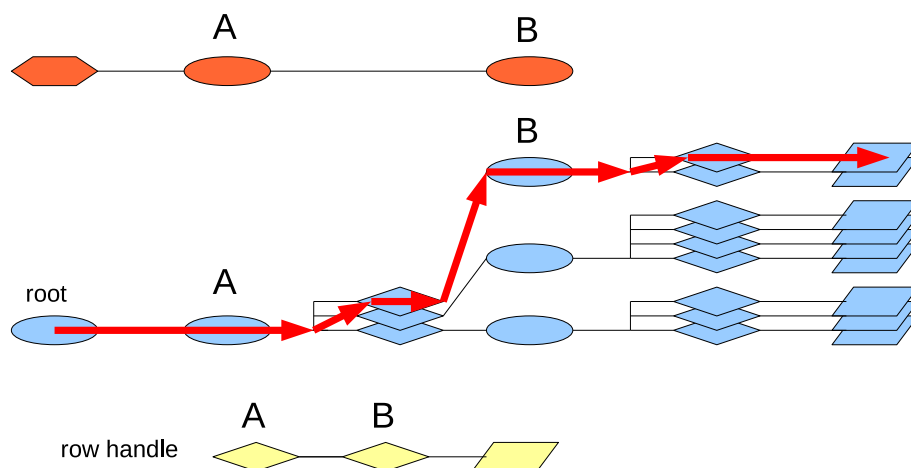
Figure 9.4. `begin()`, `beginIdx($itA)` and `beginIdx($itB)` work the same for this table.

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. 86) 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, $rh)`, on Figure 9.5 it will go through the root index to the index A. There it will try to find the matching position. If none is found, the search ends and returns a NULL row handle. If the position is found, the search progresses towards the first leaf sub-index type. Which is the index type B, and which conveniently sits in this case right under A. The position in the index A determines, which index of type B will be used for the next step. Suppose it's the second position, so the second index of type B is used. Since we're now past the target index A, the logic used is the same as for `beginIdx()`, and the first position in B2 is picked. Which then leads to the first row handle of the second sub-stack of handles.
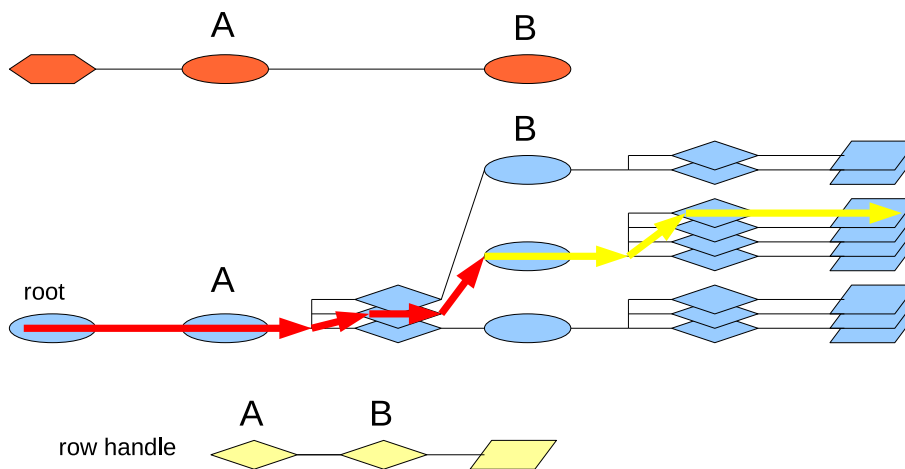
Figure 9.5. `findIdx($itA, $rh)` goes through A and then switches to the `beginIdx()` logic.

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. 88) 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.
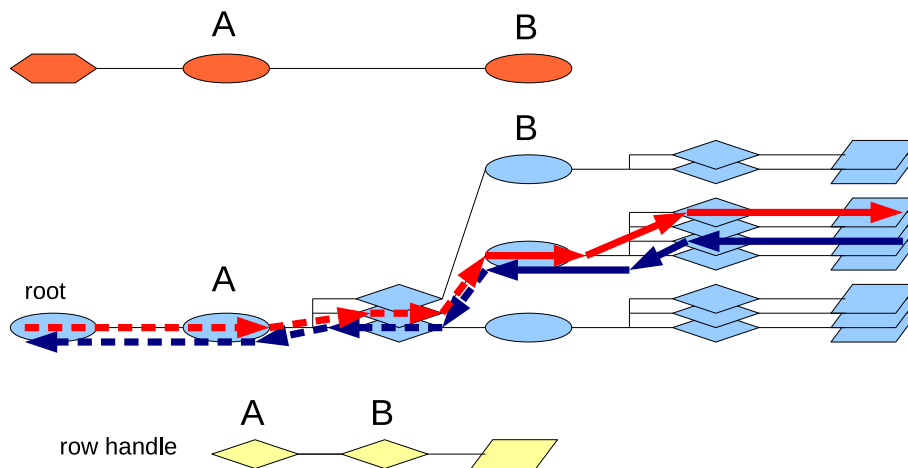
Figure 9.6. `firstOfGroupIdx($itB, $rh)`.

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, $rh)` that starts from the same row handle as in Figure 9.6 , has to logically back up to the index A, go to the next iterator there, and then follow to the first row of B3.
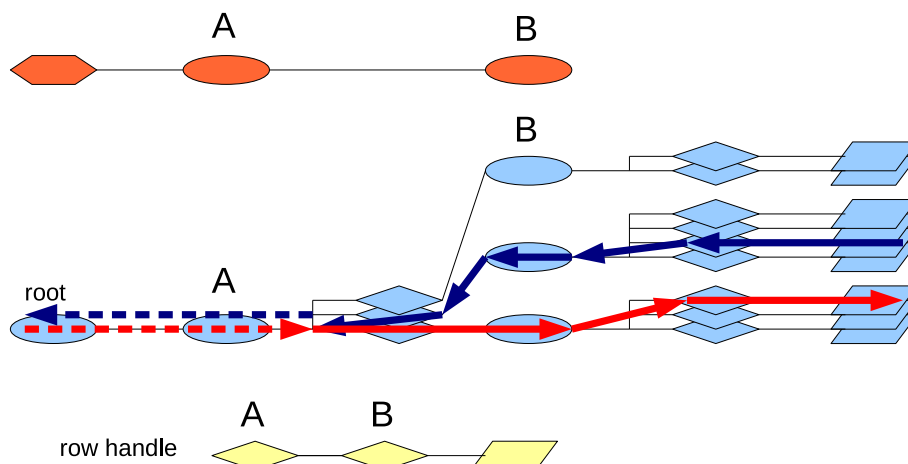
Figure 9.7. `nextGroupIdx($itB, $rh)`.

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 .
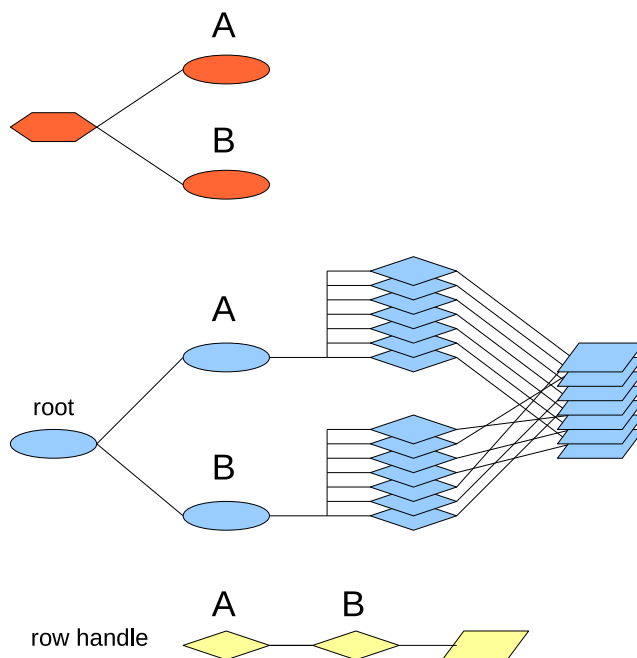
Figure 9.8. Two top-level index types.

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
```
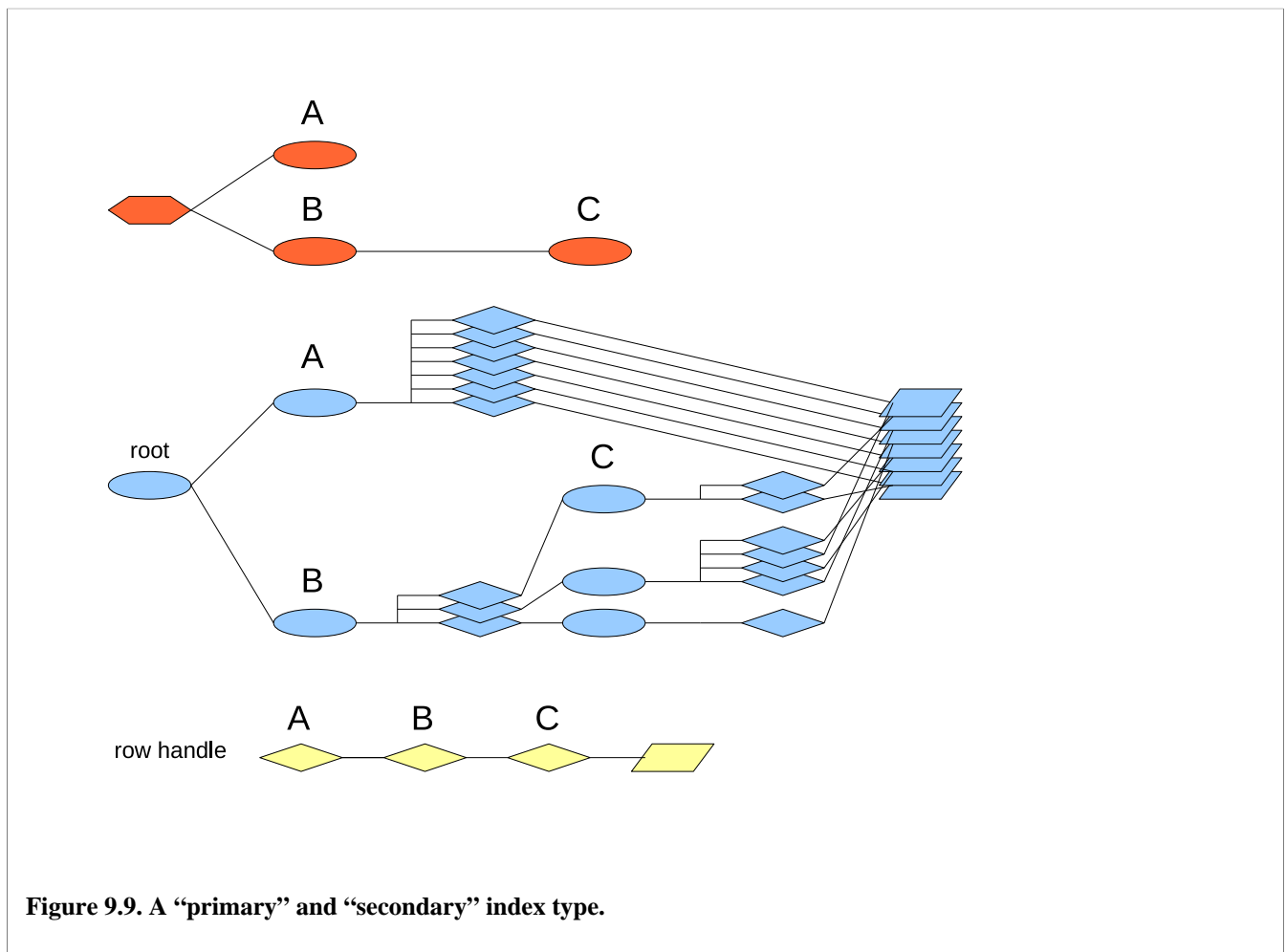

Figure 9.9. A "primary" and "secondary" index type.

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
```
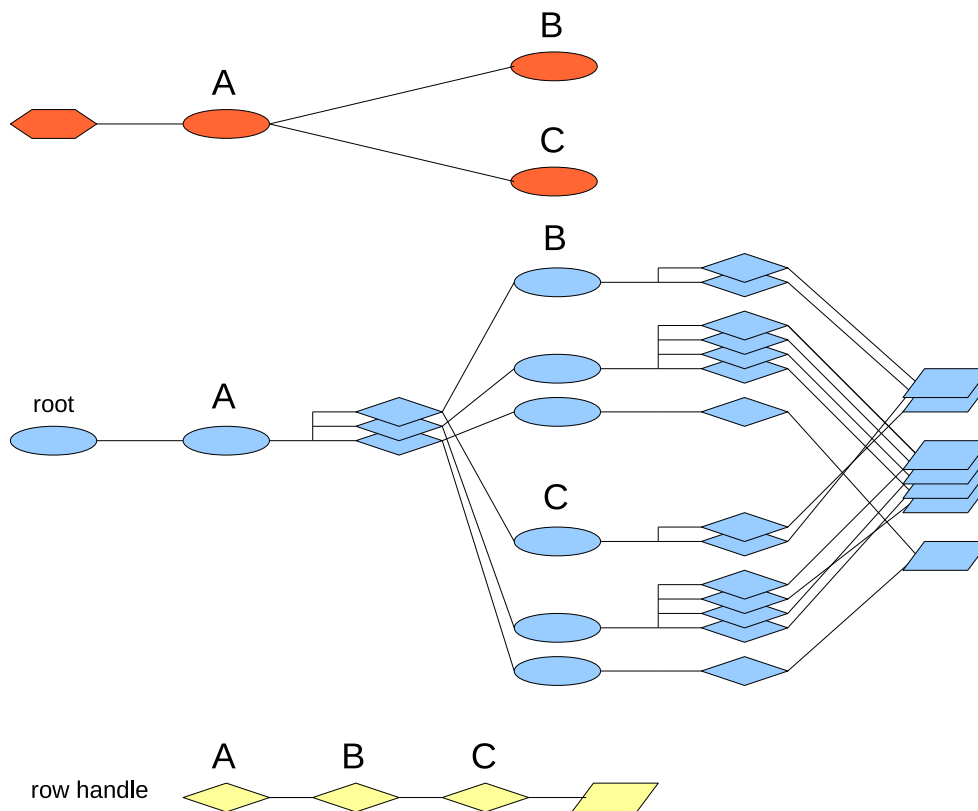
Figure 9.10. Two index types nested under one.

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 record, 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();
$tabType = $idxType->getTabtypeSafe();
```

The difference between these methods is what happens if the index type was not set into a table type yet. `getTabtype()` would confess while `getTabtypeSafe()` would return an `undef`. Which method to use, depends on the circumstances: if this situation is valid and you're ready to check for it and handle it, use `getTabtypeSafe()`, otherwise use `getTabtype()`.

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");
$itSub = $tabType->findSubIndexSafe("indexName");
@itSubs = $tabType->getSubIndexes();

$itSub = $idxType->findSubIndex("indexName");
$itSub = $idxType->findSubIndexSafe("indexName");
@itSubs = $idxType->getSubIndexes();
```

The `findSubIndex()` has been already shown in Section 9.7: "Secondary indexes" (p. 92). 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. The `Safe` versions return `undef` if the index is not found, instead of confessing. `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. 92) 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) = $tabType->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())
  )
;
```

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" ])
    )
  )
;
```

And they would even work fine, with just a little extra overhead from duplication. But `findIndexKeyPath()` will refuse such indexes and confess.

Yet another way to find an index is by the keys. Think of an SQL query: having a WHERE condition, you would want to find if there is an index on the fields in the condition, allowing to find the records quickly. Triceps is not quite up to this level of automatic query planning yet but it does some for the joins. If you know, by which fields you want to join, it's nice to find the correct index automatically. The finding of an index by key is done with the method:

```
@idxPath = $tableType->findIndexPathForKeys(@keyFields);
```

It returns the array that represents the path to an index type that matches these key fields. And then having the path you can find the index type as such. The index type and all the types in the path still have to be of the Hashed variety. If the correct index cannot be found, an empty array is returned. If you specify the fields that aren't present in the row type in the first place, this is simply treated the same as being unable to find an index for these fields. If more that one index would match, the first one found in the direct order of the index tree walk is returned.

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);
```

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. 141) .

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. 141)) 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 11: "*Aggregation*" (p. 141) . The features that take advantage of the streaming functions are described in Section 15.7: "Streaming functions and tables" (p. 264) . 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. 117) , only now in Triceps.

10.3. Simple wrapper templates

The SimpleServer package described in Section 7.9: "Main loop with a socket" (p. 54) contains templates for the repeating tasks. `makeExitLabel()` creates a label that will request the server to exit, `makeServerOutLabel()` creates a label that will send the rows from some label back into the socket.

Rather than copying the code here again, please refer to the description in Section 7.9: "Main loop with a socket" (p. 54) .

Another similar template that is used throughout the following chapters creates a label that prints the rowop contents. It's located in the package that wraps the input (e.g. feeding) and output of the tests:

```
package Triceps::X::TestFeed;

# 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");
    });
  $lbParent->chain($lb);
  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 general running is enclosed in the package Triceps::X::DumbClient:

```
package Triceps::X::DumbClient;

sub run # ($labels)
{
  my $labels = shift;

  my ($port, $pid) = Triceps::X::SimpleServer::startServer(0, $labels);
  my $sock = IO::Socket::INET->new(
```

```
      Proto => "tcp",
      PeerAddr => "localhost",
      PeerPort => $port,
    ) or confess "socket failed: $!";
    while(& readLine) {
      $sock->print($_);
      $sock->flush();
    }
    $sock->print("exit,OP_INSERT\n");
    $sock->flush();
    $sock->shutdown(1); # SHUT_WR
    while(<$sock>) {
      & send($_);
    }
    waitpid($pid, 0);
}
```

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.

Then the actual example makes use of this function:

```
# 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
);

our $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("bySymbol",
    Triceps::SimpleOrderedIndex->new(symbol => "ASC")
      ->addSubIndex("last2",
        Triceps::IndexType->newFifo(limit => 2)
      )
  )
;
$ttWindow->initialize();

my $uTrades = Triceps::Unit->new("uTrades");
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");
my $query = Query1->new($tWindow, "qWindow");
my $srvout = &Triceps::X::SimpleServer::makeServerOutLabel($query->getOutputLabel());

my %dispatch;
$dispatch{$tWindow->getName()} = $tWindow->getInputLabel();
$dispatch{$query->getName()} = $query->getInputLabel();
$dispatch{"exit"} = &Triceps::X::SimpleServer::makeExitLabel($uTrades, "exit");

Triceps::X::DumbClient::run(\%dispatch);
```

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 CLONE_SKIP { 1; }

sub new # ($class, $unit, $tabType, $name)
{
  my $class = shift;
  my $unit = shift;
  my $tabType = shift;
  my $name = shift;

  my $table = $unit->makeTable($tabType, $name);
  my $rt = $table->getRowType();
```

```perl
  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");

  $self->{sendLabel} = &Triceps::X::SimpleServer::makeServerOutLabel($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 `handle-UnitTypeLabel()` 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. 119) . 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;

sub CLONE_SKIP { 1; }

# Sending of rows to the server output.
sub new # ($class, $option => $value, ...)
{
  no warnings;

  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 {
      &Triceps::X::SimpleServer::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);
  }

  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());
```

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;

sub CLONE_SKIP { 1; }

# Sending of rows to the server output.
sub new # ($class, $option => $value, ...)
```

```
{
  no warnings;

  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 {
      &Triceps::X::SimpleServer::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);
  }

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

sub getInputLabel() # ($self)
{
  my $self = shift;
  return $self->{inLabel};
}
```

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 CLONE_SKIP { 1; }

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 CLONE_SKIP { 1; }

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) = @_;
        my $v;';
    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 .= '
        $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 CLONE_SKIP { 1; }

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;';

  # the sorting keeps the key order predictable for the tests;
  # the can also be done with Hash::Util::hash_traversal_mask()
  # but would not be backwards-compatible
  foreach $f (sort keys %qhash) {
    $v = $qhash{$f};
    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
  &Triceps::X::SimpleServer::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. 124) 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 CLONE_SKIP { 1; }

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, "tWindow");
my $query = Query7->new(table => $tWindow, name => "qWindow",
  resultFields => [ '!id', 'size/lot_$&', '.*' ],
);
# print in the tokenized format
my $srvout = $uTrades->makeLabel($query->getOutputLabel()->getType(),
  $query->getOutputLabel()->getName() . ".serverOut", undef, sub {
    &Triceps::X::SimpleServer::outCurBuf($_[1]->printP() . "\n");
  });
$query->getOutputLabel()->chain($srvout);

my %dispatch;
$dispatch{$tWindow->getName()} = $tWindow->getInputLabel();
$dispatch{$query->getName()} = $query->getInputLabel();
$dispatch{"exit"} = &Triceps::X::SimpleServer::makeExitLabel($uTrades, "exit");

Triceps::X::DumbClient::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 make-Translations() 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. 129) .

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 $result_rt = Triceps::RowType->new(@rowdef);
    # XXX extended error "$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
);

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())
  )
;

# 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();
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");

# label to print the result of aggregation
my $lbPrint = $uTrades->makeLabel($rtVwap, "lbPrint",
    undef, sub { # (label, rowop)
        print($_[1]->printP(), "\n");
    });
$tWindow->getAggregatorLabel("aggrVwap")->chain($lbPrint);

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. 92) 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
);

# 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");
    });

# 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);
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2);
  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 $rtAvgPrice = Triceps::RowType->new(
  symbol => "string", # symbol traded
  id => "int32", # last trade's id
  price => "float64", # avg price of the last 2 trades
);

our $ttAvgPrice = Triceps::TableType->new($rtAvgPrice)
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
  )
;
$ttAvgPrice->initialize();
our $tAvgPrice = $uTrades->makeTable($ttAvgPrice, "tAvgPrice");
our $lbAvgPriceHelper = $tAvgPrice->getInputLabel();

# 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);
  my $rhEnd = $rhFirst->nextGroupIdx($itLast2);
  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. 119) .

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");

# 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
);

# 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
);

# 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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  $context->send($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)
      )
    )
  )
;
$ttWindow->initialize();
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");

# 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 or source code snippet,

- 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");
```

The aggregator types are theoretically multithreaded but the way the Perl threads work, the Perl code has to be recompiled from the source code in each thread. So for a table type with aggregators to be exportable to the other threads, the aggregators must have their logic specified as the Perl source code, not a compiled Perl function.

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. 101) for the difference between an index type and an index).

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. 155). This remembered last state can also be used for the other interesting optimizations that will be shown in Section 11.6: "Additive aggregation" (p. 157) .

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.

The handling of the fatal errors (as in `die()`) in the aggregator functions is an interesting subject. The errors propagate properly through the table, and the table operations confess with the Perl handler's error message. But since an error in the aggregator function means that things are going very, very wrong, after that the table becomes inoperative and will die on all the subsequent operations as well. You need to be very careful in writing these functions.

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)
      )
    )
  )
;
```

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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  $context->send($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))
  )
;
```

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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  ${$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)
      )
    )
  )
;
```

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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  $_[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. 147) rewritten in an additive way:

```
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();
  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
  );
  $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. 151) 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)
        )
      )
    )
;
```

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();
  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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  $context->send($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)
      )
    )
  )
;
$ttWindow->initialize();
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");
```

```
# label to print the result of aggregation
my $lbAverage = $uTrades->makeLabel($rtAvgPrice, "lbAverage",
  undef, sub { # (label, rowop)
    printf("%.17g\n", $_[1]->getRow()->get("price"));
  });
$tWindow->getAggregatorLabel("aggrAvgPrice")->chain($lbAverage);
```

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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  $context->send($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())
    )
  )
;
$ttWindow->initialize();
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");

$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();
  my $avg = $sum/$count;

  my $res = $context->resultType()->makeRowHash(
    symbol => $rLast->get("symbol"),
    id => $rLast->get("id"),
    price => $avg
  );
  $context->send($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)
      )
    )
  )
;
$ttWindow->initialize();
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");

$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. 147) 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)
    )
  )
;

# 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();
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");
```

```
# 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. 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 (it gets implicitly wrapped in a `sub { ... }`, like any other source code argument):

```
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. 141) .

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 undefs.

```
  # 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 0s 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. 136) .

```
    ### 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);
}
${$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);

  $idx->setAggregator($agg);

  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 body:

```
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,
  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 that has the key fields `source` and `external`. Such index must be already defined, or the join will refuse to compile.

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,
  rightTable => $tToUsd,
  byLeft => [ "date", "currency" ],
  type => "inner",
);
```

It joins the tables `$tPosition` and `$tToUsd`, with the inner join logic. The join condition is on the fields "date" and "currency" being equal in rows in both tables.

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)
);

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
);
```

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())
  )
;
$ttAccounts->initialize();
```

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, "tAccounts");

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));
  }
});

# 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. 81) 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". This option is optional: if it's missing, the template will automatically find the index that matches the key fields. The index must exist though, if it doesn't exist, LookupJoin can't create it and can't work without it either. 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. 136). 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. If the duplications happen, the most general solution is to 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. There is also another way to solve a special case of duplication that will be shown in a moment.

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.

Now let's take another look at the field duplication problem. The most typical case of duplication is in the key fields, when the key fields on both sides are named the same. If all the fields from both left and right sides were to be included (which is the default), the key fields would be included twice, with the same names, and cause a conflict. LookupJoin provides a solution for this special case, shown in the following example:

```
our $rtTrans = Triceps::RowType->new( # a transaction received
  id => "int32", # the transaction id
  source => "string", # external system that sent us a transaction
  external => "string", # its name of the account of the transaction
  amount => "int32", # the amount of transaction (int is easier to check)
);

our $tAccounts = $uJoin->makeTable($ttAccounts, "tAccounts");

our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtTrans,
  rightTable => $tAccounts,
  byLeft => [ "source", "external" ],
  fieldsDropRightKey => 1,
  isLeft => 1,
); # would confess by itself on an error
```

The example does the exact same thing as the last one, only here the fields in the incoming rows have been named the same as in the table. This made the option "byLeft" the more convenient way to specify the join condition. And this time there are no explicit options to select the result fields, all of them are included. But that would have included the fields "source" and "external" twice, which is illegal. The option "fieldsDropRightKey" set to 1 takes care of that: it automatically removes the key fields on the right side from the result. This example produces the output that is the same as the last one, only with the different field names:

```
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" source="source1" external="999" amount="100"
    internal="1"
trans,OP_INSERT,2,source2,ABCD,200
join.out OP_INSERT id="2" source="source2" external="ABCD"
    amount="200" internal="1"
trans,OP_INSERT,3,source2,QWERTY,200
join.out OP_INSERT id="3" source="source2" external="QWERTY"
    amount="200"
acct,OP_INSERT,source2,QWERTY,2
trans,OP_DELETE,3,source2,QWERTY,200
join.out OP_DELETE id="3" source="source2" external="QWERTY"
    amount="200" internal="2"
acct,OP_DELETE,source1,999,1
```

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. 124) , 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,
  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. The right-side index is found automatically.

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())
      )
    )
;
$ttAccounts2->initialize();

our $tAccounts = $uJoin->makeTable($ttAccounts2, "tAccounts");

our $join = Triceps::LookupJoin->new(
  unit => $uJoin,
  name => "join",
  leftRowType => $rtInTrans,
  rightTable => $tAccounts,
  rightIdxPath => [ "iterateSrc", "lookupSrcExt" ],
  rightFields => [ "internal/acct" ],
  by => [ "acctSrc" => "source", "acctXtrId" => "external" ],
  #saveJoinerTo => \$code,
); # would confess by itself on an error
```

The main loop is unchanged from the first LookupJoin example, so I won't 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. It could also have been found automatically. 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,
  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);
    my @rows = $join->lookup($trans);
    foreach my $r (@rows) {
      $uJoin->call($lbPrint->makeRowop($op, $r));
    }
  }
  $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. There is a way to achieve a similar result using the streaming functions that returns the rowops. It will be described in Section 15.8: "Streaming functions and template results" (p. 266) .

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. 182) 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);

  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";
    $resLabel->getUnit()->call($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);

  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);

  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";
    $resLabel->getUnit()->call($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";
    $resLabel->getUnit()->call($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 whatever 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
);
```

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
);
```

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())
  )
;
```

```
$ttToUsd->initialize();

# 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())
  )
;
$ttPosition->initialize();

our $uJoin = Triceps::Unit->new("uJoin");

our $tToUsd = $uJoin->makeTable($ttToUsd, "tToUsd");
our $tPosition = $uJoin->makeTable($ttPosition, "tPosition");

our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  rightTable => $tToUsd,
  byLeft => [ "date", "currency" ],
  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 of the joined tables must provide a Hashed index for the efficient joining. In this case it will be "currencyLookup" on the left and "primary" on the right, found automatically by the key fields. 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.

There are two ways to provide the join condition: either specify it explicitly in the option "by" or "byLeft", or specify the indexes in both tables and have the key fields in them paired together. Or you can specify both, as long as the information stays consistent. For example, the join in this example could also be written as:

```
our $join = Triceps::JoinTwo->new(
  name => "join",
```

```
  leftTable => $tPosition,
  leftIdxPath => [ "currencyLookup" ],
  rightTable => $tToUsd,
  rightIdxPath => [ "primary" ],
  type => "inner",
); # would confess by itself on an error
```

When the key fields in the indexes are paired up together, it's done 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.

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 for the automatic pairing by index. 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. It would be no problem at all for selecting the pairing explicitly with the "by" options and letting the join find the index, which is the recommended way.

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 field pairing by 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.

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 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". There is no option "fieldsDropRightKey", JoinTwo always excludes the duplicate key fields automatically. The results in this example include all the fields from both sides by default.

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,
  rightTable => $tToUsd,
  byLeft => [ "date", "currency" ],
  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. 235), 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. 231). In it the processing of a row can be forked into multiple parallel paths, each path doing an optional part of the comutation 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,
  rightTable => $tToUsd,
  byLeft => [ "date", "currency" ],
  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. 188)). It's a more high-level template, so it solves this problem automatically, removing the duplicate fields from the result by default.

But the problem in JoinTwo is even worse because the table-to-table outer joins must work with the updates from any side. If a row finds no match in the outer join, the other side and all the fields in that other side will be NULL. If only the fields

from that other side pass through into the result, the result will contain NULLs in the key fields, which would be very wrong. Thus JoinTwo has even more magic built into it: it knows how to have the key fields copied into the result from whatever side happens to be present for a particular row, and does this by default. The other way to think about it is that it makes these fields always available on both sides.

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())
  )
;
$ttToUsd->initialize();

# 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())
  )
;
$ttPosition->initialize();

# remember the indexes for the future use
our $ixtToUsdByDate = $ttToUsd->findSubIndex("byDate");
our $ixtPositionByDate = $ttPosition->findSubIndex("byDate");

# 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. clearBy-Date() 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,
```

```
  rightTable => $tToUsd,
  byLeft => [ "date", "currency" ],
  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. 219). 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");
our $lbPositionFilter = $uJoin->makeLabel($tPosition->getRowType,
  "lbPositionFilter", undef, sub {
    if ($_[1]->getRow()->get("date") >= $businessDay) {
      $uJoin->call($lbPositionCurrent->adopt($_[1]));
    }
  });
$tPosition->getOutputLabel()->chain($lbPositionFilter);

our $lbToUsdCurrent = $uJoin->makeDummyLabel(
  $tToUsd->getRowType, "lbToUsdCurrent");
our $lbToUsdFilter = $uJoin->makeLabel($tToUsd->getRowType,
  "lbToUsdFilter", undef, sub {
    if ($_[1]->getRow()->get("date") >= $businessDay) {
      $uJoin->call($lbToUsdCurrent->adopt($_[1]));
    }
  });
$tToUsd->getOutputLabel()->chain($lbToUsdFilter);

our $join = Triceps::JoinTwo->new(
  name => "join",
  leftTable => $tPosition,
  leftFromLabel => $lbPositionCurrent,
  rightTable => $tToUsd,
  rightFromLabel => $lbToUsdCurrent,
  byLeft => [ "date", "currency" ],
  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
);

# 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())
  )
;
$ttRate->initialize();

our $uArb = Triceps::Unit->new("uArb");

our $tRate = $uArb->makeTable($ttRate, "tRate");

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())
  )
;
$ttJoin1->initialize();
our $tJoin1 = $uArb->makeTable($ttJoin1, "tJoin1");
$join1->getOutputLabel()->chain($tJoin1->getInputLabel());

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",
);
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
  }
});
$join2->getOutputLabel()->chain($lbCompute);

# 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 getOutput-Label().

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",
);
my $ixtCcy1 = $ttRate->findSubIndex("byCcy1");
my $ixtCcy12 = $ixtCcy1->findSubIndex("byCcy12");

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);
  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
    }
  }
});
$tRate->getOutputLabel()->chain($lbCompute);
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",
);
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
  }
});
$join2->getOutputLabel()->chain($lbCompute);
```

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);
```

```perl
  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") {
```

```perl
    $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");

   # 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. 193) .

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
);

# 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
);
```

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);
    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,
  );
  ${$state} = $res;
  $context->send($opcode, $res);
}

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)
        )
      )
    )
  )
;

$ttPackets->initialize();
our $tPackets = $uTraffic->makeTable($ttPackets, "tPackets");

# 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")
  )
;

$ttHourly->initialize();
our $tHourly = $uTraffic->makeTable($ttHourly, "tHourly");

# connect the tables
$tPackets->getAggregatorLabel("aggrHourly")->chain($tHourly->getInputLabel());
```

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);
    }
    &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
);

# 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
);
```

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,
  );
  ${$state} = $res;
  $context->send($opcode, $res);
}
```

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()
    )
  )
;

$ttHourly->initialize();
our $tHourly = $uTraffic->makeTable($ttHourly, "tHourly");

# remember the daily secondary index type
our $idxHourlyByDay = $ttHourly->findSubIndex("byDay");
our $idxHourlyByDayGroup = $idxHourlyByDay->findSubIndex("group");
```

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" ])
  )
;

$ttDaily->initialize();
our $tDaily = $uTraffic->makeTable($ttDaily, "tDaily");

# 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:

```
# the manual aggregation of a day's data
sub computeDay # ($dateStamp)
{
  our $uTraffic;
  my $bytes = 0;

  my $rhFirst = $tHourly->findIdxBy($idxHourlyByDay, day => $_[0]);
  my $rhEnd = $rhFirst->nextGroupIdx($idxHourlyByDayGroup);
  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);
    }
    &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 `flushOld-Packets()` 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).
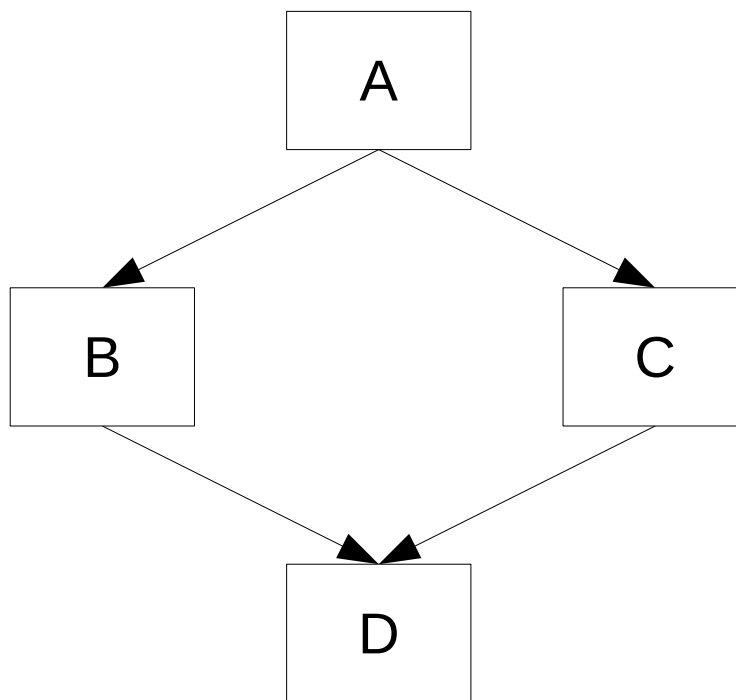
Figure 14.1. The diamond topology.

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 7.6: "No bundling" (p. 45)).

The multithreaded case in Triceps will be discussed separately in Section 16.10: "The threaded dreaded diamond and data reordering" (p. 323) .

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));
  }
});

$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);
});

$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);
});
```

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",
);
```

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.

```
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") } ],
  }, @_);

  # Keeps the names of the datasets in the order they have been defined
  # (since the hash loses the order).
  $self->{dsetnames} = [];

  # parse the data element
  my %data_unparsed = @{$self->{data}};
  my $dataset = {};
  &Triceps::Opt::parse("$class data set (" . ($data_unparsed{name} or 'UNKNOWN') . ")",
 $dataset, {
    name => [ undef, \&Triceps::Opt::ck_mandatory ],
    key => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "ARRAY", "") } ],
    rowType => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::RowType"); } ],
    fromLabel => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Label"); } ],
  }, @{$self->{data}});
```

The options parsing goes as usual. The option "data" is parsed again for the options inside it, and those are placed into the hash %$dataset.

```
  # save the dataset for the future
  push @{$self->{dsetnames}}, $dataset->{name};
  $self->{datasets}{$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. 124) 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();
```

```
  $dataset->{tbInsert} = $self->{unit}->makeTable($dataset->{tt}, $self->{name} . "." .
$dataset->{name} . ".tbInsert");
  $dataset->{tbDelete} = $self->{unit}->makeTable($dataset->{tt}, $self->{name} . "." .
$dataset->{name} . ".tbDelete");
```

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);
  $dataset->{lbOut} = $self->{unit}->makeDummyLabel($dataset->{rowType}, $self->{name} .
"." . $dataset->{name} . ".out");
```

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});
    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 $self = shift;
  return @{$self->{dsetnames}};
}
```

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! Another way to do the collapse will be shown in Section 15.2: "Streaming functions by example, another version of Collapse" (p. 249). And one more piece to it is shown in Section 15.8: "Streaming functions and template results" (p. 266) .

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",
);

# the data is auto-generated by a sequence
our $seq = 0;

our $ttData = Triceps::TableType->new($rtData)
  ->addSubIndex("fifo", Triceps::IndexType->newFifo())
;
$ttData->initialize();
our $tData = $uChunks->makeTable($ttData, "tJoin1");
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",
);

# rowops to run when the model is otherwise idle
our $trayIdle = $uChunks->makeTray();

our $lbReportNote = $uChunks->makeDummyLabel($rtNote, "lbReportNote"
);
makePrintLabel("lbPrintNote", $lbReportNote);

# code that clears the table in small chunks
our $lbClear = $uChunks->makeLabel($rtNote, "lbClear", undef, sub {
  $tData->clear(2); # no more than 2 rows deleted per run
  if ($tData->size() > 0) {
    $trayIdle->push($_[0]->adopt($_[1]));
  } else {
    $uChunks->makeHashCall($lbReportNote, "OP_INSERT",
      text => "done clearing",
    );
  }
});
```

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. 202) . Here for simplicity all the data get wiped out.

But the method clear() 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 and there still are rows left in the table, 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 7.9: "Main loop with a socket" (p. 54) 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. Streaming functions

15.1. Introduction to streaming functions

The streaming functions are a cool and advanced concept. I've never seen it anywhere before, and for all I know I have invented it.

First let's look at the differences between the common functions and macros (or templates and such), shown in Figure 15.1 .
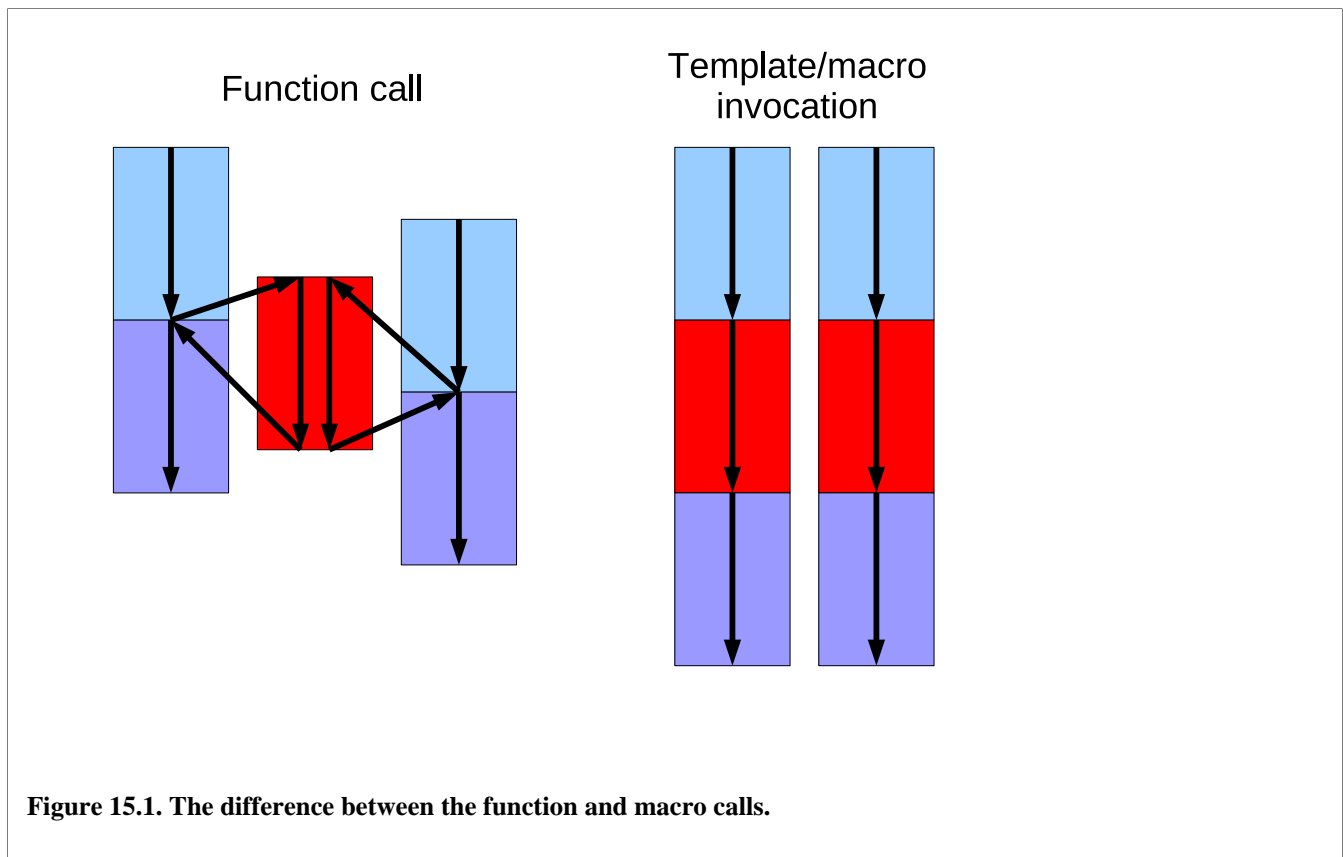
Figure 15.1. The difference between the function and macro calls.

What happens during a function call? Some code (marked with the light bluish color) is happily zooming along when it decides to call a function. It prepares some arguments and jumps to the function code (reddish). The function executes, computes its result and jumps back to the point right after it has been called from. Then the original code continues from there (the slightly darker bluish color).

What happens during a macro (or template) invocation? It starts with some code zooming along in the same way, however when the macro call time comes, it prepares the arguments and then does nothing. It gets away with it because the compiler has done the work: it has placed the macro code right where it's called, so there is no need for jumps. After the macro is done, again it does nothing: the compiler has placed the next code to execute right after it, so it just continues on its way.

So far it's pretty equivalent. An interesting difference happens when the function or macro is called from more than one place. With a macro, another copy of the macro is created, inserted between its call and return points. That's why in the figure the macro is shown twice. But with the function, the same function code is executed every time, and then returns back to the caller. That's why in the figure there are two function callers with their paths through the same function. But how does the function know, where should it jump on return? The caller tells it by pushing the return address onto the stack. When the function is done, it pops this address from the stack and jumps there.

Still, it looks all the same. A macro call is a bit more efficient, except when a large complex macro is called from many places, then it becomes more efficient as a function. However there is another difference if the function or macro holds some context (say, a static variable): each invocation of the macro will get its own context but all the function calls will share the same context. The only way to share the context with a macro is to pass some global context as its argument (or you can use a separately defined global variable if you're willing to dispense with some strict modularity).

Now let's switch to the CEP world. The Sybase or StreamBase modules are essentially macros, and so are the Triceps templates. When such a macro gets instantiated, a whole new copy of it gets created with its tables/windows and streams/labels. Its input and output streams/labels get all connected in a fixed way. The limitation is that if the macro contains any tables, each instantiation gets another copy of them. Well, in Triceps you can use a table as an argument to a template. In the other systems I think you still can't, so if you want to work with a common table in a module, you have to make up the query-response patterns, like the one described in Section 10.1: "Comparative modularity" (p. 117) .

In a query-response pattern there is some common sub-model, with a stream (in Triceps terms, a label, but here we're talking the other systems) for the queries to come in and a stream for the results to come out (both sides might have not only one but multiple streams). There are multiple inputs connected, from all the request sources, and the outputs are connected back to all the request sources. All the request sources (i.e. callers) get back the whole output of the pattern, so they need to identify, what output came from their input, and ignore the rest. They do this by adding the unique ids to their queries, and filter the results. In the end, it looks *almost* like a function but with much pain involved.

To make it look quite like a function, one thing is needed: the selective connection of the result streams (or, returning to the Triceps terminology, labels) to the caller. Connect the output labels, send some input, have it processed and send the result through the connection, disconnect the output labels. And what you get is a streaming function. It's very much like a common function but working on the streaming data arguments and results.

The Figure 15.2 highlights the similarity and differences between the query patterns and the streaming functions.
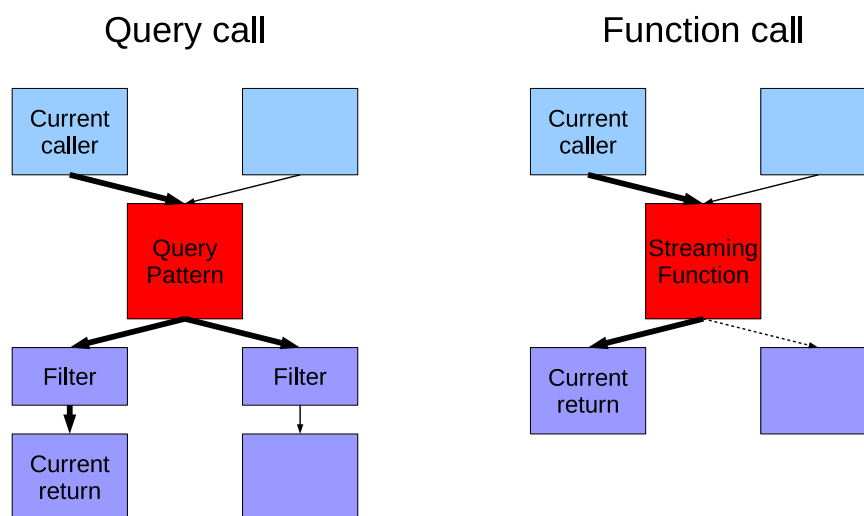
Figure 15.2. The query patterns and streaming functions.

The thick lines show where the data goes during one concrete call. The thin lines show the connections that do exist but without the data going through them at the moment (they will be used during the other calls, from these other callers). The dashed thin line shows the connection that doesn't exist at the moment. It will be created when needed (and at that time the thick arrow from the streaming function to what is now the current return would disappear).

The particular beauty of the streaming functions for Triceps is that the other callers don't even need to exist yet. They can be created and connected dynamically, do their job, call the function, use its result, and then be disposed of. The calling side in Triceps doesn't have to be streaming either: it could as well be procedural.

15.2. Streaming functions by example, another version of Collapse

The streaming functions have proved quite useful in Triceps, in particular the inter-thread communications use an interface derived from them. But the ironic part is that coming up with the good examples of the streaming function usage in Triceps is surprisingly difficult. The flexibility of Triceps is the problem. If all you have is SQL, the streaming functions become pretty much a must. But if you can write the procedural code, most things are easier that way, with the normal procedural functions. For a streaming function to become beneficial, it has to be written in SQLy primitives (such as tables, joins) and not be easily reducible to the procedural code. The streaming function examples that aren't big enough for their own file are collected in `t/xFn.t`.

The most distilled example I've come up with is for the implementation of Collapse. The original implementation of Collapse is described in Section 14.2: "Collapsed updates" (p. 235) . The `flush()` there goes in a loop deleting the all rows from the state tables and sending them as rowops to the output.

The deletion of all the rows can nowadays be done easier with the Table method `clear()`. However by itself it doesn't solve the problem of sending the output. It sends the deleted rows to the table's output label but we can't just connect the output of the state tables to the Collapse output: then it would also pick up all the intermediate changes! The data needs to be picked up from the tables output selectively, only in `flush()`.

This makes it a good streaming function: the body of the function consists of running `clear()` on the state tables, and its result is whatever comes on the output labels of the tables.

Since most of the logic remains unchanged, I've implemented this new version of Collapse in `t/xFn.t` as a subclass that extends and replaces some of the code with its own:

```
package FnCollapse;

sub CLONE_SKIP { 1; }

our @ISA=qw(Triceps::Collapse);

sub new # ($class, $optName => $optValue, ...)
{
  my $class = shift;
  my $self = $class->SUPER::new(@_);
  # Now add an FnReturn to the output of the dataset's tables.
  # One return is enough for both.
  # Also create the bindings for sending the data.
  foreach my $dataset (values %{$self->{datasets}}) {
    my $fret = Triceps::FnReturn->new(
      name => $self->{name} . "." . $dataset->{name} . ".retTbl",
      labels => [
        del => $dataset->{tbDelete}->getOutputLabel(),
        ins => $dataset->{tbInsert}->getOutputLabel(),
      ],
    );
    $dataset->{fret} = $fret;

    # these variables will be compiled into the binding snippets
    my $lbOut = $dataset->{lbOut};
    my $unit = $self->{unit};
    my $OP_INSERT = &Triceps::OP_INSERT;
    my $OP_DELETE = &Triceps::OP_DELETE;
```

```
    my $fbind = Triceps::FnBinding->new(
      name => $self->{name} . "." . $dataset->{name} . ".bndTbl",
      on => $fret,
      unit => $unit,
      labels => [
        del => sub {
          if ($_[1]->isDelete()) {
            $unit->call($lbOut->adopt($_[1]));
          }
        },
        ins => sub {
          if ($_[1]->isDelete()) {
            $unit->call($lbOut->makeRowop($OP_INSERT, $_[1]->getRow()));
          }
        },
      ],
    );
    $dataset->{fbind} = $fbind;
  }
  bless $self, $class;
  return $self;
}

# Override the base-class flush with a different implementation.
sub flush # ($self)
{
  my $self = shift;
  foreach my $dataset (values %{$self->{datasets}}) {
    # The binding takes care of producing and directing
    # the output. AutoFnBind will unbind when the block ends.
    my $ab = Triceps::AutoFnBind->new(
      $dataset->{fret} => $dataset->{fbind}
    );
    $dataset->{tbDelete}->clear();
    $dataset->{tbInsert}->clear();
  }
}
```

new() adds the streaming function elements in each data set. They consist of two parts: FnReturn defines the return value of a streaming function (there is no formal definition of the body or the entry point since they are quite flexible), and FnBinding defines a call of the streaming function. In this case the function is called in only one place, so one FnBinding is defined. If called from multiple places, there would be multiple FnBindings.

When a normal procedural function is called, the return address provides the connection to get the result back from it to the caller. In a streaming function, the FnBinding connects the result labels to the caller's further processing of the returned data. Unlike the procedural functions, the data is not returned in one step (run the function, compute the value, return it). Instead the return value of a streaming function is a stream of rowops. As each of them is sent to a return label, it goes through the binding and to the caller's further processing. Then the streaming function continues, producing the next rowop, and so on.

If this sounds complicated, please realize that here we're dealing with the assembly language equivalent for streaming functions. I expect that over time the more high-level primitives will be developed and it will become easier.

The second source of complexity is that the arguments of a streaming function are not computed in one step either. You don't normally have a full set of rows to send to a streaming function in one go. Instead you set up the streaming call to bind the result, then you pump the argument rowops to the function's input, creating them in whatever way you wish.

Getting back to the definition of a streaming function, FnReturn defines a set of labels, each with a logical name. In this case the names are "del" and "ins". The labels inside FnReturn are a special variety of dummy labels, but they are chained to some real labels that send the result of the function. The snippet

```
del => $dataset->{tbDelete}->getOutputLabel(),
```

says "create a return label named 'del' and chain it from the tbDelete's output label". The FnReturn normally does its chaining with `chainFront()`, unless the option `chainFront => 0` tells it otherwise. But in this particular case the chaining order wouldn't matter. There are more details to the naming and label creation but let's not get bogged in them now.

The FnBinding defines a matching set of labels, with the same logical names. It's like a receptacle and a plug: you put the plug into the receptacle and get the data flowing, you unplug it and the data flow stops. The Perl version of FnBinding provides a convenience: when it gets a code reference instead of a label, it automatically creates a label with that code for its handler.

In this case both binding labels forward the data to the Collapse's output label. Only the one for the Insert table has to change the opcodes to OP_INSERT. The check

```
if ($_[1]->isDelete()) ...
```

is really redundant, to be on the safe side, since we know that when the data will be flowing, all of it will be coming from the table clearing and have the opcodes of OP_DELETE.

The actual call happens in `flush()`: Triceps::AutoFnBind is a constructor of the scope object that does the "plug into receptable" thing, with automatic unplugging when the object returned by it gets destroyed on leaving the block scope. If you want to do things manually, FnReturn has the methods `push()` and `pop()` but the scoped binding is safer and easier. Once the binding is done, the data is sent through the function by calling `clear()` on both tables. And then the block ends, `$ab` get destroyed, AutoFnBind destructor undoes the binding, and thus the streaming function call completes.

The result produced by this version of Collapse is exactly the same as by the original version. And even when we get down to grits, it's produced with the exact same logical sequence: the rows are sent out as they are deleted from the state tables. But it's structured differently: instead of the procedural deletion and sending of the rows, the internal machinery of the tables gets invoked, and the results of that machinery are then converted to the form suitable for the collapse results and propagated to the output.

Philosophically, it could be argued, what is the body of this function? Is it just the internal logic of the table delection, that gets triggered by `clear()` in the caller? Or are the `clear()` calls also a part of the function body? But it practice it just doesn't matter, whatever.

15.3. Collapse with grouping by key with streaming functions

The Collapse as shown before sends all the collected deletes before all the collected inserts. For example, if it has collected the updates for four rows, the output will be (assuming that the Collapse element is named `collapse` and the data set in it is named `idata`):

```
collapse.idata.out OP_DELETE local_ip="3.3.3.3" remote_ip="7.7.7.7"
    bytes="100"
collapse.idata.out OP_DELETE local_ip="2.2.2.2" remote_ip="6.6.6.6"
    bytes="100"
collapse.idata.out OP_DELETE local_ip="4.4.4.4" remote_ip="8.8.8.8"
    bytes="100"
collapse.idata.out OP_DELETE local_ip="1.1.1.1" remote_ip="5.5.5.5"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="3.3.3.3" remote_ip="7.7.7.7"
    bytes="300"
collapse.idata.out OP_INSERT local_ip="2.2.2.2" remote_ip="6.6.6.6"
    bytes="300"
collapse.idata.out OP_INSERT local_ip="4.4.4.4" remote_ip="8.8.8.8"
    bytes="300"
```

```
collapse.idata.out OP_INSERT local_ip="1.1.1.1" remote_ip="5.5.5.5"
    bytes="300"
```

What if you want the updates produced as deletes immediately followed by the matching inserts with the same key? Like this:

```
collapse.idata.out OP_DELETE local_ip="3.3.3.3" remote_ip="7.7.7.7"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="3.3.3.3" remote_ip="7.7.7.7"
    bytes="300"
collapse.idata.out OP_DELETE local_ip="2.2.2.2" remote_ip="6.6.6.6"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="2.2.2.2" remote_ip="6.6.6.6"
    bytes="300"
collapse.idata.out OP_DELETE local_ip="4.4.4.4" remote_ip="8.8.8.8"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="4.4.4.4" remote_ip="8.8.8.8"
    bytes="300"
collapse.idata.out OP_DELETE local_ip="1.1.1.1" remote_ip="5.5.5.5"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="1.1.1.1" remote_ip="5.5.5.5"
    bytes="300"
```

With the procedural version it would have required doing a look-up in the Insert table after processing each row in the Delete table and handling it if found. So I've left it out to avoid complicating that example. But in the streaming function form it becomes easy, just change the binding of the "del" label a little bit:

```
    my $lbInsInput = $dataset->{tbInsert}->getInputLabel();

    my $fbind = Triceps::FnBinding->new(
      name => $self->{name} . "." . $dataset->{name} . ".bndTbl",
      on => $fret,
      unit => $unit,
      labels => [
        del => sub {
          if ($_[1]->isDelete()) {
            $unit->call($lbOut->adopt($_[1]));
            # If the INSERT is available after this DELETE, this
            # will produce it.
            $unit->call($lbInsInput->adopt($_[1]));
          }
        },
        ins => sub {
          if ($_[1]->isDelete()) {
            $unit->call($lbOut->makeRowop($OP_INSERT, $_[1]->getRow()));
          }
        },
      ],
    );
```

The "del" binding first sends the result out as usual and then forwards the DELETE rowop to the Insert table's input. Which then causes the INSERT rowop to be sent if a match is found. Mind you, the look-up and conditional processing still happens. But now it all happens inside the table machinery, all you need to do is add one more line to invoke it.

Let's talk through in a little more detail, what happens when the clearing of the Delete table deletes the row with (`local_ip="3.3.3.3" remote_ip="7.7.7.7"`).

1. The Delete table sends a rowop with this row and OP_DELETE to its output label `collapse.idata.tbDelete.out`.

2. Which then gets forwarded to a chained label in the FnReturn, `collapse.idata.retTbl.del`.

Streaming functions

3. FnReturn has an FnBinding pushed into it, so the rowop passes to the matching label in the binding, `collapse.idata.bndTbl.del`.

4. The Perl handler of that label gets called, first forwards the rowop to the Collapse output label `collapse.idata.out`, and then to the Insert table's input label `collapse.idata.tbInsert.in`.

5. The Insert table looks up the row by the key, finds it, removes it from the table, and sends an OP_DELETE rowop to its output label `collapse.idata.tbInsert.out`.

6. Which then gets forwarded to a chained label in the FnReturn, `collapse.idata.retTbl.ins`.

7. FnReturn has an FnBinding pushed into it, so the rowop passes to the matching label in the binding, `collapse.idata.bndTbl.ins`.

8. The Perl handler of that label gets called and sends the rowop with the opcode changed to OP_INSERT to the Collapse output label `collapse.idata.out`.

It's a fairly complicated sequence but all you needed to do was to add one line of code. The downside of course is that if something goes not the way you expected, you'd have to trace and understand the whole long sequence (that's the typical trouble with the SQL-based systems).

When the INSERTs are sent after DELETEs, their rows are removed from the Insert table too, so the following `clear()` of the Insert table won't find them any more and won't send any duplicates; it will send only the inserts for which there were no matching deletes.

And of course if there is only a DELETE collected for a certain key, not an update, there will be no matching row in the Insert table, so the forwarded DELETE request will have no effect and produce no output from the Insert table.

You may notice that the code in the "del" handler only forwards the rows around, and that can be replaced by a chaining:

```
my $lbDel = $unit->makeDummyLabel(
  $dataset->{tbDelete}->getOutputLabel()->getRowType(),
  $self->{name} . "." . $dataset->{name} . ".lbDel");
$lbDel->chain($lbOut);
$lbDel->chain($lbInsInput);

my $fbind = Triceps::FnBinding->new(
  name => $self->{name} . "." . $dataset->{name} . ".bndTbl",
  on => $fret,
  unit => $unit,
  labels => [
    del => $lbDel,
    ins => sub {
      $unit->call($lbOut->makeRowop($OP_INSERT, $_[1]->getRow()));
    },
  ],
);
```

This shows another way of label definition in FnBinding: an actual label is created first and then given to the FnBinding, instead of letting it automatically create a label from the code. The condition "`if ($_[1]->isDelete())`" has been removed from the "ins" part, since it's really redundant and the "del" part with its chaining doesn't do this check anyway.

This code works just as well and even more efficiently than the previous version, since no Perl code needs to be invoked for "del", it all propagates internally through the chaining. However the price is that the DELETE rowops coming out of the output label will have the head-of-the-chain label in them:

```
collapse.idata.lbDel OP_DELETE local_ip="3.3.3.3" remote_ip="7.7.7.7"
```

```
    bytes="100"
collapse.idata.out OP_INSERT local_ip="3.3.3.3" remote_ip="7.7.7.7"
    bytes="300"
collapse.idata.lbDel OP_DELETE local_ip="2.2.2.2" remote_ip="6.6.6.6"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="2.2.2.2" remote_ip="6.6.6.6"
    bytes="300"
collapse.idata.lbDel OP_DELETE local_ip="4.4.4.4" remote_ip="8.8.8.8"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="4.4.4.4" remote_ip="8.8.8.8"
    bytes="300"
collapse.idata.lbDel OP_DELETE local_ip="1.1.1.1" remote_ip="5.5.5.5"
    bytes="100"
collapse.idata.out OP_INSERT local_ip="1.1.1.1" remote_ip="5.5.5.5"
    bytes="300"
```

The "ins" side can't be handled just by chaining because it has to replace the opcode in the rowops. Another potential way to handle this would be to define various preprogrammed label types in C++ for many primitive operations, like replacing the opcode, and then build the models by combining them.

The final item is that the code shown in this section involved a recursive call of the streaming function. Its output from the "del" label got fed back to the function, producing more output on the "ins" label. This worked because it invoked a different code path in the streaming function than the one that produced the "del" data. If it were to form a topological loop back to the same path with the same labels, that would have been an error. The more advanced use of recursion is possible and will be discussed in more detail later.

15.4. Table-based translation with streaming functions

Next I want to show an example that is in its essence kind of dumb. The same thing is easier to do in Triceps with templates. And the whole premise is not exactly great either. But it provides an opportunity to show more of the streaming functions, in a set-up that is closer to the SQL-based systems.

The background is as follows: There happen to be multiple ways to identify the securities (stock shares and such). *RIC* is the identifier used by Reuters (and quite often by the other data suppliers too), consisting of the ticker symbol on an exchange, a dot, and the coded name of the exchange (such as "L" for the London stock exchange or "N" for the New York stock exchange). *ISIN* is the international standard alphanumeric identifier. A security (and some of its creative equivalents) might happen to be listed on multiple exchanges, each listing having its own RIC. And if you wonder, the ticker names are allocated separately by each exchange and may differ. But all of these RICs refer to the same security, thus translating to the same ISIN (there might be multiple ISINs too but that's another story). A large financial company would want to track a security all around the world. To aggregate the data on the security worldwide, it has to identify it by ISIN, but the data feed might be coming in as RIC only. The translation of RIC to ISIN is then done by the table during processing. The RIC is not thrown away either, it shows the detail of what and where had happened. But ISIN is added for the aggregation on it.

The data might be coming from multiple feeds, and there are multiple kinds of data: trades, quotes, lending quotes and so on, each with its own schema and its own aggregations. However the step of RIC-to-ISIN translation is the same for all of them, is done by the same table, and can be done in one place. Of course, multithreading can add more twists here but for now we're talking about a simple single-threaded example.

An extra complexity is that in the real world the translation table might be incomplete. However some feeds might provide both RICs and ISINs in their records, so the pairs that aren't in the reference table yet, can be inserted there and used for the following translations. This is actually not such a great idea, because it means that there might be previous records that went through before the translation became available. A much better way would be to do the translation as a join, where the update to a reference table would update any previous records as well. But then there would not be much use for a streaming function in it. As I've said before, it's a rather dumb example.

The streaming function will work like this: It will get an argument pair of (RIC, ISIN) from an incoming record. Either component of this pair might be empty. Since the rest of the record is wildly different for different feeds, the rest of the record is left off at this point, and the uniform argument of (RIC, ISIN) is given to the function. The function will consult its table, see if it can add more information from there, or add more information from the argument into the table, and return the hopefully enriched pair (RIC, ISIN) with an empty ISIN field replaced by the right value, to the caller.

The function is defined like this:

```
my $rtIsin = Triceps::RowType->new(
  ric => "string",
  isin => "string",
);

my $ttIsin = Triceps::TableType->new($rtIsin)
  ->addSubIndex("byRic", Triceps::IndexType->newHashed(key => [ "ric" ])
);
$ttIsin->initialize();

my $tIsin = $unit->makeTable($ttIsin, "tIsin");

# the results will come from here
my $fretLookupIsin = Triceps::FnReturn->new(
  name => "fretLookupIsin",
  unit => $unit,
  labels => [
    result => $rtIsin,
  ],
);

# The function argument: the input data will be sent here.
my $lbLookupIsin = $unit->makeLabel($rtIsin, "lbLookupIsin", undef, sub {
  my $row = $_[1]->getRow();
  if ($row->get("ric")) {
    my $argrh = $tIsin->makeRowHandle($row);
    my $rh = $tIsin->find($argrh);
    if ($rh->isNull()) {
      if ($row->get("isin")) {
        $tIsin->insert($argrh);
      }
    } else {
      $row = $rh->getRow();
    }
  }
  $unit->call($fretLookupIsin->getLabel("result")->makeRowop("OP_INSERT", $row));
});
```

The `$fretLookupIsin` is the function result, `$lbLookupIsin` is the function input. In this example the result label in FnReturn is defined differently than in the previous ones: not by a source label but by a row type. This label doesn't get chained to anything, instead the procedural code in the function finds it as `$fretLookupIsin->getLabel("result")` and calls it directly.

Then the ISIN translation code for some trades feed would look as follows (remember, supposedly there would be many feeds, each one with its own schema, but for the example I show only one):

```
my $rtTrade = Triceps::RowType->new(
  ric => "string",
  isin => "string",
  size => "float64",
  price => "float64",
);
```

```
my $lbTradeEnriched = $unit->makeDummyLabel($rtTrade, "lbTradeEnriched");
my $lbTrade = $unit->makeLabel($rtTrade, "lbTrade", undef, sub {
  my $rowop = $_[1];
  my $row = $rowop->getRow();
  Triceps::FnBinding::call(
    name => "callTradeLookupIsin",
    on => $fretLookupIsin,
    unit => $unit,
    rowop => $lbLookupIsin->makeRowopHash("OP_INSERT",
      ric => $row->get("ric"),
      isin => $row->get("isin"),
    ),
    labels => [
      result => sub { # a label will be created from this sub
        $unit->call($lbTradeEnriched->makeRowop($rowop->getOpcode(),
          $row->copymod(
            isin => $_[1]->getRow()->get("isin")
          )
        ));
      },
    ],
  );
});
```

The label `$lbTrade` receives the incoming trades, calls the streaming function to enrich them with the ISIN data, and forwards the enriched data to the label `$lbTradeEnriched`. The function call is done differently in this example. Rather than create a FnBinding object and then use it with a scoped AutoFnBind, it uses the convenience function `FnBinding::call()` that wraps all that logic. It's simpler to use, without all these extra objects, but the price is the efficiency: it ends up creating a new FnBinding object for every call. That's where a compiler would be very useful, it could take a call like this, translate it to the internal objects once, and then keep reusing them.

The `FnBinding::call()` option "name" gives a name that is used for the error messages and also to produce the names of the temporary objects it creates. The option "on" tells, which streaming function is being called (by specifying its FnReturn). The option "rowop" gives the arguments of the streaming functions. There are multiple ways to do that: option "rowop" for a single rowop, "rowops" for an array of rowops, "tray" for a tray, and "code" for a procedural code snippet that would send the inputs to the streaming function. And "labels" as usual connects the results of the function, either to the existing labels, or by creating labels automatically from the snippets of code.

The result handling in this example demonstrates the technique that I call the "implicit join": The function gets a portion of data from an original row, does some transformation and returns the data back. This data is then joined with the original row. The code knows, what this original row was, it gets remembered in the variable `$row`. The semantics of the call guarantees that nothing else has happened during the function call, and that `$row` is still the current row. Then the function result gets joined with `$row`, and the produced data is sent further on its way. The variable `$row` could be either a global one, or as shown here a scoped variable that gets embedded into a closure function.

The rest of the example, the dispatcher part, is:

```
# print what is going on
my $lbPrintIsin = makePrintLabel("printIsin", $tIsin->getOutputLabel());
my $lbPrintTrade = makePrintLabel("printTrade", $lbTradeEnriched);

# the main loop
my %dispatch = (
  isin => $tIsin->getInputLabel(),
  trade => $lbTrade,
);

while(<STDIN>) {
  chomp;
```

```
  my @data = split(/,/); # starts with a command, then string opcode
  my $type = shift @data;
  my $lb = $dispatch{$type};
  my $rowop = $lb->makeRowopArray(@data);
  $unit->call($rowop);
  $unit->drainFrame(); # just in case, for completeness
}
```

And an example of running, with the input lines shown in bold:

isin,OP_INSERT,ABC.L,US0000012345
tIsin.out OP_INSERT ric="ABC.L" isin="US0000012345"
isin,OP_INSERT,ABC.N,US0000012345
tIsin.out OP_INSERT ric="ABC.N" isin="US0000012345"
isin,OP_INSERT,DEF.N,US0000054321
tIsin.out OP_INSERT ric="DEF.N" isin="US0000054321"
trade,OP_INSERT,ABC.L,,100,10.5
lbTradeEnriched OP_INSERT ric="ABC.L" isin="US0000012345" size="100"
    price="10.5"
trade,OP_DELETE,ABC.N,,200,10.5
lbTradeEnriched OP_DELETE ric="ABC.N" isin="US0000012345" size="200"
    price="10.5"
trade,OP_INSERT,GHI.N,,300,10.5
lbTradeEnriched OP_INSERT ric="GHI.N" isin="" size="300" price="10.5"
trade,OP_INSERT,,XX0000012345,400,10.5
lbTradeEnriched OP_INSERT ric="" isin="XX0000012345" size="400"
    price="10.5"
trade,OP_INSERT,GHI.N,XX0000012345,500,10.5
tIsin.out OP_INSERT ric="GHI.N" isin="XX0000012345"
lbTradeEnriched OP_INSERT ric="GHI.N" isin="XX0000012345" size="500"
    price="10.5"
trade,OP_INSERT,GHI.N,,600,10.5
lbTradeEnriched OP_INSERT ric="GHI.N" isin="XX0000012345" size="600"
    price="10.5"

The table gets pre-populated with a few translations, and the first few trades use them. Then goes the example of a non-existing translation, which gets eventually added from the incoming data (see that the trade with (GHI.N, XX0000012345) both updates the ISIN table and sends through the trade record), and the following trades can then use this newly added translation but obviously the older ones do not get updated.

15.5. Streaming functions and loops

The streaming functions can be used to replace the topological loops (where the connection between the labels go in circles) with the procedural ones. Just make the body of the loop into a streaming function and connect its output with its own input (and of course also to the loop results). Then call this function in a procedural while-loop until the data stop circulating.

The way the streaming functions have been described so far, there is a catch, even two of them: First, with such a connection, the output of the streaming function would immediately circulate to its input, and would try to keep circulating until the loop is done, with no need for a while-loop. Second, as soon as it attempts to circulate, the scheduler will detect a recursive call and die (unless you change the recursion settings, however this is not a good reason to change them).

But there is also a solution that has not been described yet: an FnBinding can collect the incoming rowops in a tray instead of immediately forwarding them. This tray can be called later, after the original function call completes. This way the iteration has its data collected, the function completes, and then the next iteration of the while-loop starts, sending the data from the previous iteration. When there is nothing to send any more, the loop completes.

Using this logic, let's rewrite the Fibonacci example with the streaming function loops. Its original version and description of the logic can be found in Section 7.7: "Topological loops" (p. 46) .

The new version is:

```
my $uFib = Triceps::Unit->new("uFib");

###
# A streaming function that computes one step of a
# Fibonacci number, will be called repeatedly.

# Type of its input and output.
my $rtFib = Triceps::RowType->new(
  iter => "int32", # number of iterations left to do
  cur => "int64", # current number
  prev => "int64", # previous number
);

# Input:
#   $lbFibCompute: request to do a step. iter will be decremented,
#     cur moved to prev, new value of cur computed.
# Output (by FnReturn labels):
#   "next": data to send to the next step, if the iteration
#     is not finished yet (iter in the produced row is >0).
#   "result": the result data if the iretaion is finished
#     (iter in the produced row is 0).
# The opcode is preserved through the computation.

my $frFib = Triceps::FnReturn->new(
  name => "Fib",
  unit => $uFib,
  labels => [
    next => $rtFib,
    result => $rtFib,
  ],
);

my $lbFibCompute = $uFib->makeLabel($rtFib, "FibCompute", undef, sub {
  my $row = $_[1]->getRow();
  my $prev = $row->get("cur");
  my $cur = $prev + $row->get("prev");
  my $iter = $row->get("iter") - 1;
  $uFib->makeHashCall($frFib->getLabel($iter > 0? "next" : "result"), $_[1]->getOpcode(),
    iter => $iter,
    cur => $cur,
    prev => $prev,
  );
});

# End of streaming function
###

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

# binding to run the Triceps steps in a loop
my $fbFibLoop = Triceps::FnBinding->new(
  name => "FibLoop",
  on => $frFib,
  withTray => 1,
  labels => [
    next => $lbFibCompute,
    result => $lbPrint,
```

```
  ],
);

my $lbMain = $uFib->makeLabel($rtFib, "Main", undef, sub {
  my $row = $_[1]->getRow();
  {
    my $ab = Triceps::AutoFnBind->new($frFib, $fbFibLoop);

    # send the request into the loop
    $uFib->makeHashCall($lbFibCompute, $_[1]->getOpcode(),
      iter => $row->get("iter"),
      cur => 0, # the "0-th" number
      prev => 1,
    );

    # now keep cycling the loop until it's all done
    while (!$fbFibLoop->trayEmpty()) {
      $fbFibLoop->callTray();
    }
  }
  print(" is Fibonacci number ", $row->get("iter"), "\n");
});

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

It produces the same output as before (as usual, the lines in bold are the input lines):

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

The option "withTray" of FnBind is what makes it collect the rowops in a tray. The rowops are not the original incoming ones but already translated to call the FnBinding's output labels. The method `callTray()` swaps the tray with a fresh one and then calls the original tray with the collected rowops. There are more methods for the tray control: `swapTray()` swaps the tray with a fresh one and returns the original one, which can then be read or called; `traySize()` returns not just the emptiness condition but the whole size of the tray.

The whole loop runs in one binding scope, because it doesn't change with the iterations. The first row primes the loop, and then it continues while there is anything to circulate.

This example sent both the next iteration rows and the result rows through the binding. But for the result rows it doesn't have to. They can be sent directly out of the loop:

```
my $lbFibCompute = $uFib->makeLabel($rtFib, "FibCompute", undef, sub {
  my $row = $_[1]->getRow();
  my $prev = $row->get("cur");
  my $cur = $prev + $row->get("prev");
  my $iter = $row->get("iter") - 1;
  $uFib->makeHashCall($iter > 0? $frFib->getLabel("next") : $lbPrint, $_[1]->getOpcode(),
    iter => $iter,
    cur => $cur,
```

```
    prev => $prev,
  );
});
```

The printed result is exactly the same as in the previous example.

15.6. Streaming functions and pipelines

The streaming functions can be arranged into a pipeline by binding the result of one function to the input of another one. Fundamentally, the pipelines in the world of streaming functions are analogs of the nested calls with the common functions. For example, a pipeline (written for shortness in the Unix way)

```
a | b | c
```

is an analog of the common function calls

```
c(b(a()))
```

Of course, if the pipeline is fixed, it can as well be connected directly with the label chaining and then stay like this. A more interesting case is when the pipeline needs to be reconfigured dynamically based on the user requests. An interesting example of pipeline usage comes from the data security. A client may connect to a CEP model element in a clear-text or encrypted way. In the encrypted way the data received from the client needs to be decrypted, then processed, and then the results encrypted before sending them back:

```
receive | decrypt | process | encrypt | send
```

In the clear-text mode the pipeline becomes shorter:

```
receive | process | send
```

Let's make an example around this idea: To highlight the flexibility, the configuration will be selectable for each input line. If the input starts with a "+", it will be considered encrypted, otherwise clear-text. Since the actual security is not important for the example, it will be simulated by encoding the text in hex (each byte of data becomes two hexadecimal digits). The real encryption, such as SSL, would of course require the key negotiation, but this little example just skips over this part, since it has no key. First, define the input and output (receive and send) endpoints:

```
# All the input and output gets converted through an intermediate
# format of a row with one string field.
my $rtString = Triceps::RowType->new(
  s => "string"
);

# All the input gets sent here.
my $lbReceive = $unit->makeDummyLabel($rtString, "lbReceive");
my $retReceive = Triceps::FnReturn->new(
  name => "retReceive",
  labels => [
    data => $lbReceive,
  ],
);

# The binding that actually prints the output.
my $bindSend = Triceps::FnBinding->new(
  name => "bindSend",
  on => $retReceive, # any matching return will do
  unit => $unit,
  labels => [
    data => sub {
```

```
        print($_[1]->getRow()->get("s"), "\n");
      },
    ],
);
```

The same row type $rtString will be used for the whole pipeline, sending through the arbitrary strings of text. The binding $bindSend is defined on $retReceive, so they can actually be short-circuited together. But they don't have to. $bindSend can be bound to any matching return. The matching return is defined as having the same number of labels in it, with matching row types. The names of the labels don't matter but their order does. It's a bit tricky: when a binding is created, the labels in it get connected to the return on which it's defined by name. But at this point each of them gets assigned a number, in order the labels went in that original return. After that only this number matters: if this binding gets connected to another matching return, it will get the data from the return's label with the same number, not the same name.

Next step, define the endpoints for the processing: the dispatcher and the output label. All of them use the same row type and matching returns. The actual processing will eventually be hard-connected between these endpoints.

```
my %dispatch; # the dispatch table will be set here

# The binding that dispatches the input data
my $bindDispatch = Triceps::FnBinding->new(
  name => "bindDispatch",
  on => $retReceive,
  unit => $unit,
  labels => [
    data => sub {
      my @data = split(/,/, $_[1]->getRow()->get("s")); # starts with a command, then
 string opcode
      my $type = shift @data;
      my $lb = $dispatch{$type};
      my $rowop = $lb->makeRowopArray(@data);
      $unit->call($rowop);
    },
  ],
);

# All the output gets converted to rtString and sent here.
my $lbOutput = $unit->makeDummyLabel($rtString, "lbOutput");
my $retOutput = Triceps::FnReturn->new(
  name => "retOutput",
  labels => [
    data => $lbOutput,
  ],
);
```

And now the filters for encryption and decryption. Each of them has a binding for its input and a return for its output. The actual pseudo-encryption transformation is done with Perl functions unpack() and pack().

```
# The encryption pipeline element.
my $retEncrypt = Triceps::FnReturn->new(
  name => "retEncrypt",
  unit => $unit,
  labels => [
    data => $rtString,
  ],
);
my $lbEncrypt = $retEncrypt->getLabel("data");
my $bindEncrypt = Triceps::FnBinding->new(
  name => "bindEncrypt",
  on => $retReceive,
  unit => $unit,
```

```
    labels => [
      data => sub {
        my $s = $_[1]->getRow()->get("s");
        $unit->makeArrayCall($lbEncrypt, "OP_INSERT", unpack("H*", $s));
      },
    ],
);

# The decryption pipeline element.
my $retDecrypt = Triceps::FnReturn->new(
  name => "retDecrypt",
  unit => $unit,
  labels => [
    data => $rtString,
  ],
);
my $lbDecrypt = $retDecrypt->getLabel("data");
my $bindDecrypt = Triceps::FnBinding->new(
  name => "bindDecrypt",
  on => $retReceive,
  unit => $unit,
  labels => [
    data => sub {
      my $s = $_[1]->getRow()->get("s");
      $unit->makeArrayCall($lbDecrypt, "OP_INSERT", pack("H*", $s));
    },
  ],
);
```

Then goes the body of the model. It defines the actual row types for the data that gets parsed from strings and the business logic (which is pretty simple, increasing an integer field). The dispatch table connects the dispatcher with the business logic, and the conversion from the data rows to the plain text rows is done with template makePipePrintLabel(). This template is very similar to the template makePrintLabel() that was shown in Section 10.3: "Simple wrapper templates" (p. 119) .

```
sub makePipePrintLabel($$$) # ($print_label_name, $parent_label, $out_label)
{
  my $name = shift;
  my $lbParent = shift;
  my $lbOutput = shift;
  my $unit = $lbOutput->getUnit();
  my $lb = $lbParent->getUnit()->makeLabel($lbParent->getType(), $name,
    undef, sub { # (label, rowop)
      $unit->makeArrayCall(
        $lbOutput, "OP_INSERT", $_[1]->printP());
    });
  $lbParent->chain($lb);
  return $lb;
}

# The body of the model: pass through the name, increase the count.
my $rtData = Triceps::RowType->new(
  name => "string",
  count => "int32",
);

my $lbIncResult = $unit->makeDummyLabel($rtData, "result");
my $lbInc = $unit->makeLabel($rtData, "inc", undef, sub {
  my $row = $_[1]->getRow();
  $unit->makeHashCall($lbIncResult, $_[1]->getOpcode(),
    name  => $row->get("name"),
```

```
      count => $row->get("count") + 1,
  );
});
makePipePrintLabel("printResult", $lbIncResult, $lbOutput);

%dispatch = (
  inc => $lbInc,
);
```

Finally, the main loop. It will check the input lines for the leading "+" and construct one or the other pipeline for processing. Of course, the pipelines don't have to be constructed in the main loop. They could have been constructed in the handler of `$lbReceive` just as well (then it would need a separate label to send its result to, and to include into `$retReceive`).

```
while(<STDIN>) {
  my $ab;
  chomp;
  if (/^\+/) {
    $ab = Triceps::AutoFnBind->new(
      $retReceive => $bindDecrypt,
      $retDecrypt => $bindDispatch,
      $retOutput => $bindEncrypt,
      $retEncrypt => $bindSend,
    );
    $_ = substr($_, 1);
  } else {
    $ab = Triceps::AutoFnBind->new(
      $retReceive => $bindDispatch,
      $retOutput => $bindSend,
    );
  };
  $unit->makeArrayCall($lbReceive, "OP_INSERT", $_);
  $unit->drainFrame();
}
```

The constructor of AutoFnBind can accept multiple return-binding pairs. It will bind them all, and unbind them back on its object destruction. It's the same thing as creating multiple AutoFnBind objects, one for each pair, only more efficient. And here is an example of a run (as usual the input lines are in bold, and the long lines get wrapped):

```
inc,OP_INSERT,abc,1
result OP_INSERT name="abc" count="2"
inc,OP_DELETE,def,100
result OP_DELETE name="def" count="101"
+696e632c4f505f494e534552542c6162632c32
726573756c74204f505f494e53455254206e616d653d226162632220636f756e743d2
  2332220
+696e632c4f505f44454c4554452c6465662c313031
726573756c74204f505f44454c455445206e616d653d226465662220636f756e743d2
  23130322220
```

What is in the encrypted data? The input lines have been produced by running a Perl expression manually:

```
$ perl -e 'print((unpack "H*", "inc,OP_INSERT,abc,2"), "\n");'
696e632c4f505f494e534552542c6162632c32
$ perl -e 'print((unpack "H*", "inc,OP_DELETE,def,101"), "\n");'
696e632c4f505f44454c4554452c6465662c313031
```

They and their results can be decoded by running another Perl expression:

```
$ perl -e 'print((pack "H*", "726573756c74204f505f494e53455254206e616
  d653d226162632220636f756e743d22332220"), "\n");'
```

```
result OP_INSERT name="abc" count="3"
$ perl -e 'print((pack "H*", "726573756c74204f505f44454c455445206e616
  d653d226465662220636f756e743d223130322220"), "\n");'
result OP_DELETE name="def" count="102"
```

15.7. Streaming functions and tables

Sometimes you might want to collect a table's reaction to an operation on it and process it manually afterwards. Triceps 1.0 had a special feature called "copy tray" to support that but starting with the version 2.0 the streaming functions solve this problem much better, replacing the copy trays.

If you connect the table's output to a FnReturn and then push a binding with a tray onto it, the table's output will be collected on that tray. There is even a Table method that creates this FnReturn:

```
$fret = $table->fnReturn();
```

The return contains the labels "pre", "out", "dump" (more on that one below) and the named labels for all aggregators. The FnReturn object is created on the first call of this method and is kept in the table. All the following calls return the same object. This has some interesting consequences for the "pre" label: the rowop for the "pre" label doesn't get created at all if there is nothing chained from that label. But when the FnReturn gets created, one of its labels gets chained from the "pre" label. Which means that once you call `$table->fnReturn()` for the first time, you will see that table's "pre" label called in all the traces. It's not a huge extra overhead, but still something to keep in mind and not be surprised when calling `fnReturn()` changes all your traces.

The following code demonstrates the use of an FnReturn to collect the changes done to a table on insert:

```
my $fret1 = $t1->fnReturn();
my $fbind1 = Triceps::FnBinding->new(
    unit => $unit,
    name => "fbind1",
    on => $fret1,
    withTray => 1,
    labels => [
        out => sub { }, # another way to make a dummy
    ],
);

$fret1->push($fbind1);
$t1->insert($row1);
$fret1->pop($fbind1);

# $tray contains the rowops produced by the update
my $tray = $fbind1->swapTray(); # get the updates on an insert
my @rowops = $tray->toArray();
```

And then you could for example check if any rowop has the DELETE opcode, this meaning that an old row was displaced by this insert. Of course, this is not the most efficient way. Placing the check into the label handler would be a better approach. And you don't even have to collect the rowops in a tray, you can as well compute the result on the fly:

```
my $seenDelete;

my $fret1 = $t1->fnReturn();
my $fbind1 = Triceps::FnBinding->new(
    unit => $unit,
    name => "fbind1",
    on => $fret1,
    labels => [
        out => sub {
      $seenDelete = 1 if ($_[1]->isDelete());
```

```
    }
    ],
);

$fret1->push($fbind1);
$seenDelete = 0;
$t1->insert($row1);
$fret1->pop($fbind1);

if ($seenDelete) {
  # there was a displacement
}
```

The variable `$seenDelete` is remembered in the closure function that handles the "out" label and sets it accordingly.

In both examples the binding doesn't have to be created from scratch each time. Creating it once and then reusing as needed would be more efficient.

And of course the use of an FnReturn doesn't preclude you from connecting the table outputs as usual.

Another feature where the tables and streaming functions intersect is the table dumping. It allows to iterate on a table in a functional manner.

The label "dump" is present in the table and its FnReturn. Whenever the method `Table::dumpAll()` is called, it sends the whole contents of the table to that label. Then you can set a binding on the table's FnReturn, call `dumpAll()`, and the binding will iterate through the whole table's contents.

If you want to get the dump label explicitly, you can do it with

```
my $dlab = $table->getDumpLabel();
```

Normally the only reason to do that would be to add it to another FnReturn (besides the table's FnReturn). Chaining anything else directly to this label would not make much sense, because the dump of the table can be called from many places, and the directly chained label will receive data every time the dump is called.

The grand plan is also to add the dumping by a a condition that selects a sub-index, but it's not implemented yet. You can select an index for an alternative ordering but all the rows get dumped in any case.

The method `dumpAllIdx()` is the one that sends the rows in the order of a chosen index, rather than the default first leaf index:

```
$table->dumpAll();
$table->dumpAllIdx($indexType);
```

As usual, the index type must belong to the exact type of this table. For example:

```
$table->dumpAllIdx($table->getType()->findIndexPath("cb"), "OP_NOP");
```

The typical usage looks like this:

```
Triceps::FnBinding::call(
  name => "iterate",
  on => $table->fnReturn(),
  unit => $unit,
  labels => [
    dump => sub { ... },
  ],
  code => sub {
    $table->dumpAll();
```

```
  },
);
```

It's less efficient than the normal iteration but sometimes comes handy.

Normally the rowops are sent with the opcode OP_INSERT. But the opcode can also be specified explicitly:

```
$table->dumpAll($opcode);
$table->dumpAllIdx($indexType, $opcode);
```

And some more interesting examples will be forthcoming in Section 15.11: "Streaming functions and unit boundaries" (p. 281) and Section 17.6: "Internals of a TQL join" (p. 345) .

15.8. Streaming functions and template results

The same way as the FnReturns can be used to get back the direct results of the operations on the tables, they can be also used on the templates in general. Indeed, it's a good idea to have a method that would create an FnReturn in all the templates. So I went ahead and added it to the LookupJoin, JoinTwo and Collapse.

For the joins, the resulting FnReturn has one label "out". It's created similarly to the table's:

```
my $fret = $join->fnReturn();
```

And then it can be used as usual. The implementation of this method is fairly simple:

```
sub fnReturn # (self)
{
  my $self = shift;
  if (!defined $self->{fret}) {
    $self->{fret} = Triceps::FnReturn->new(
      name => $self->{name} . ".fret",
      labels => [
        out => $self->{outputLabel},
      ],
    );
  }
  return $self->{fret};
}
```

All this makes the method `lookup()` of LookupJoin essentially redundant, since now pretty much all the same can be done with the streaming function API, and even better, because it provides the opcodes on rowops, can handle the full processing, and calls the rowops one by one without necessarily creating an array. But it could happen yet that the `lookup()` has some more convenient uses too, so I didn't remove it yet.

For Collapse the interface is a little more complicated: the FnReturn contains a label for each data set, named the same as the data set. The order of labels follows the order of the data set definitions (though right now it's kind of moot, because only one data set is supported). The implementation is:

```
sub fnReturn # (self)
{
  my $self = shift;
  if (!defined $self->{fret}) {
    my @labels;
    for my $n (@{$self->{dsetnames}}) {
      push @labels, $n, $self->{datasets}{$n}{lbOut};
    }
    $self->{fret} = Triceps::FnReturn->new(
      name => $self->{name} . ".fret",
```

```
      labels => \@labels,
    );
  }
  return $self->{fret};
}
```

Use these examples to write the `fnReturn()` in your templates.

15.9. Streaming functions and recursion

Let's look again at the pipeline example. Suppose we want to do the encryption twice (you know, maybe we have a secure channel to a semi-trusted intermediary who can can read the envelopes and forward the encrypted messages he can't read to the final destination). The pipeline becomes

```
decrypt | decrypt | process | encrypt | encrypt
```

Or if you want to think about it in a more function-like notation, rather than a pipeline, the logic can also be expressed as:

```
encrypt(encrypt(process(decrypt(decrypt(data)))))
```

However it would not work directly: a decrypt function has only one output and it can not have two bindings at the same time, it would not know which one to use at any particular time.

Instead you can make decrypt into a template, instantiate it twice, and connect into a pipeline. It's very much like what the Unix shell does: it instantiates a new process for each part of its pipeline.

But there is also another possibility: instead of assembling the whole pipeline in advance, do it in steps.

Start by adding this option in every binding:

```
withTray => 1,
```

This will make all the bindings collect the result on a tray instead of sending it on immediately. Then modify the main loop:

```
while(<STDIN>) {
  chomp;

  # receive
  my $abReceive = Triceps::AutoFnBind->new(
    $retReceive => $bindDecrypt,
  );
  $unit->makeArrayCall($lbReceive, "OP_INSERT", $_);

  # 1st decrypt
  my $abDecrypt1 = Triceps::AutoFnBind->new(
    $retDecrypt => $bindDecrypt,
  );
  $bindDecrypt->callTray();

  # 2nd decrypt
  my $abDecrypt2 = Triceps::AutoFnBind->new(
    $retDecrypt => $bindDispatch,
  );
  $bindDecrypt->callTray();

  # processing
  my $abProcess = Triceps::AutoFnBind->new(
    $retOutput => $bindEncrypt,
  );
```

```
  $bindDispatch->callTray();

  # 1st encrypt
  my $abEncrypt1 = Triceps::AutoFnBind->new(
    $retEncrypt => $bindEncrypt,
  );
  $bindEncrypt->callTray();

  # 2nd encrypt
  my $abEncrypt2 = Triceps::AutoFnBind->new(
    $retEncrypt => $bindSend,
  );
  $bindEncrypt->callTray();

  # send
  $bindSend->callTray();
}
```

Here I've dropped the encrypted-or-unencrypted choice to save the space, the data is always encrypted twice. The `drain-Frame()` call has been dropped because with the way the function calls work here there is no chance that it could be useful. The rest of the code stays the same.

The bindings have been split in stages. The next binding is set in each stage, and the data from the previous binding gets sent into it. The binding method `callTray()` replaces the tray in the binding with an empty one, and then calls all the rowops collected on the old tray (and if you wonder what then happens to the old tray, it gets discarded). Because of this the first decryption stage with binding

```
my $abDecrypt1 = Triceps::AutoFnBind->new(
  $retDecrypt => $bindDecrypt,
);
```

doesn't send the data circling forever. It just does one pass through the decryption and prepares for the second pass.

Every time `AutoFnBind->new()` runs, it doesn't replace the binding of the FnReturn but pushes a new binding onto the FnReturn's stack. Each FnReturn has its own stack of bindings (this way it's easier to manage than a single stack). When an AutoFnBind gets destroyed, it pops the binding from the return's stack. And yes, if you specify multiple bindings in one AutoFnBind, all of them get pushed on construction and popped on destruction. In this case all the auto-binds are in the same block, so they will all be destroyed at the end of block in the opposite order. Which means that in effect the code is equivalent to the nested blocks. And the version with explicit nexted blocks might be easier for you to think of:

```
while(<STDIN>) {
  chomp;

  # receive
  my $abReceive = Triceps::AutoFnBind->new(
    $retReceive => $bindDecrypt,
  );
  $unit->makeArrayCall($lbReceive, "OP_INSERT", $_);

  {
    # 1st decrypt
    my $abDecrypt1 = Triceps::AutoFnBind->new(
      $retDecrypt => $bindDecrypt,
    );
    $bindDecrypt->callTray();

    {
      # 2nd decrypt
      my $abDecrypt1 = Triceps::AutoFnBind->new(
        $retDecrypt => $bindDispatch,
```

```
      );
      $bindDecrypt->callTray();

      {
        # processing
        my $abProcess = Triceps::AutoFnBind->new(
          $retOutput => $bindEncrypt,
        );
        $bindDispatch->callTray();

        {
          # 1st encrypt
          my $abEncrypt1 = Triceps::AutoFnBind->new(
            $retEncrypt => $bindEncrypt,
          );
          $bindEncrypt->callTray();

          {
            # 2nd encrypt
            my $abEncrypt1 = Triceps::AutoFnBind->new(
              $retEncrypt => $bindSend,
            );
            $bindEncrypt->callTray();

            # send
            $bindSend->callTray();
          }
        }
      }
    }
  }
}
```

An interesting consequence of all this nesting, pushing and popping is that you can put the inner calls into the procedural loops if you wish. For example, if you want to process every input line thrice:

```
while(<STDIN>) {
  chomp;

  # receive
  my $abReceive = Triceps::AutoFnBind->new(
    $retReceive => $bindDecrypt,
  );

  for (my $i = 0; $i < 3; $i++) {
    $unit->makeArrayCall($lbReceive, "OP_INSERT", $_);

    {
      # 1st decrypt
      my $abDecrypt1 = Triceps::AutoFnBind->new(
        $retDecrypt => $bindDecrypt,
      );
      $bindDecrypt->callTray();

      {
        # 2nd decrypt
        my $abDecrypt1 = Triceps::AutoFnBind->new(
          $retDecrypt => $bindDispatch,
        );
        $bindDecrypt->callTray();

        {
```

```
          # processing
          my $abProcess = Triceps::AutoFnBind->new(
            $retOutput => $bindEncrypt,
          );
          $bindDispatch->callTray();

          {
            # 1st encrypt
            my $abEncrypt1 = Triceps::AutoFnBind->new(
              $retEncrypt => $bindEncrypt,
            );
            $bindEncrypt->callTray();

            {
              # 2nd encrypt
              my $abEncrypt1 = Triceps::AutoFnBind->new(
                $retEncrypt => $bindSend,
              );
              $bindEncrypt->callTray();

              # send
              $bindSend->callTray();
            }
          }
        }
      }
    }
  }
}
```

This code will run the whole pipeline three times for each input line, and print out three output lines. The following example of the output has both the input and the output lines wrapped, since they are hugely long:

**363936653633326334663530356634393465353334353532353432633631363236
3332633332**
3732363537333735366337343230346635303566343934653533343535323534323
0366536313664363533643232363136323633332323230363336663735366537
343364323233333323230
3732363537333735366337343230346635303566343934653533343535323534323
0366536313664363533643232363136323633332323230363336663735366537
343364323233333323230
3732363537333735366337343230346635303566343934653533343535323534323
0366536313664363533643232363136323633332323230363336663735366537
343364323233333323230

If you wonder, what is the meaning of these lines, they are the same as before. The input is :

```
inc,OP_INSERT,abc,2
```

And each line of output is:

```
result OP_INSERT name="abc" count="3"
```

I suppose, it would be more entertaining if the processing weren't just incrementing a value in the input data but incrementing some static counter, then the three output lines would be different.

However this is not the only way to do the block nesting. The contents of the FnBinding's tray is not affected in any way by the binding being pushed or popped. It stays there throughout, until it's explicitly flushed by `callTray()`. So it could use the blocks formed in a more pipeline-like fashion (as opposed to the more function-call-like fashion shown before):

```
while(<STDIN>) {
```

```
  chomp;

  # receive
  {
    my $abReceive = Triceps::AutoFnBind->new(
      $retReceive => $bindDecrypt,
    );
    $unit->makeArrayCall($lbReceive, "OP_INSERT", $_);
  }

  # 1st decrypt
  {
    my $abDecrypt1 = Triceps::AutoFnBind->new(
      $retDecrypt => $bindDecrypt,
    );
    $bindDecrypt->callTray();
  }

  # 2nd decrypt
  {
    my $abDecrypt1 = Triceps::AutoFnBind->new(
      $retDecrypt => $bindDispatch,
    );
    $bindDecrypt->callTray();
  }

  # processing
  {
    my $abProcess = Triceps::AutoFnBind->new(
      $retOutput => $bindEncrypt,
    );
    $bindDispatch->callTray();
  }

  # 1st encrypt
  {
    my $abEncrypt1 = Triceps::AutoFnBind->new(
      $retEncrypt => $bindEncrypt,
    );
    $bindEncrypt->callTray();
  }

  # 2nd encrypt
  {
    my $abEncrypt1 = Triceps::AutoFnBind->new(
      $retEncrypt => $bindSend,
    );
    $bindEncrypt->callTray();
  }

  # send
  $bindSend->callTray();
}
```

After each stage, its binding is popped but the tray is carried through to the next stage.

Which way of blocking is better? I'd say they're pretty equivalent in functionality, and your preference would depend on what style you prefer to express.

15.10. Streaming functions and more recursion

There are great many slightly different ways to use recursion with the streaming functions. This section goes through them with examples of the Fibonacci numbers computed in all these ways. You can as well skip over this section if you're not particularly interested in the details of recursive execution.

All the examples from this section (and most of others from this chapter) are locates in `t/xFn.t`. The first example uses the dumb recursive calls. It's a real dumb recursive way, with two recursive calls and thus the exponential execution time, just to show how they can be done. This simplest and most straightforward way goes as follows:

```
my $uFib = Triceps::Unit->new("uFib");
$uFib->setMaxRecursionDepth(100);

# Type the data going into the function
my $rtFibArg = Triceps::RowType->new(
  idx => "int32", # the index of Fibonacci number to generate
);

# Type of the function result
my $rtFibRes = Triceps::RowType->new(
  idx => "int32", # the index of Fibonacci number
  fib => "int64", # the generated Fibonacci number
);

###
# A streaming function that computes a Fibonacci number.

# Input:
#   $lbFibCompute: request to compute the number.
# Output (by FnReturn labels):
#   "result": the computed value.
# The opcode is preserved through the computation.

my $frFib = Triceps::FnReturn->new(
  name => "Fib",
  unit => $uFib,
  labels => [
    result => $rtFibRes,
  ],
);

my $lbFibResult = $frFib->getLabel("result");

my $lbFibCompute; # must be defined before assignment, for recursion
$lbFibCompute = $uFib->makeLabel($rtFibArg, "FibCompute", undef, sub {
  my $row = $_[1]->getRow();
  my $op = $_[1]->getOpcode();
  my $idx = $row->get("idx");
  my $res;

  if ($idx < 1) {
    $res = 0;
  } elsif($idx == 1) {
    $res = 1;
  } else {
    my ($prev1, $prev2);
    Triceps::FnBinding::call(
      name => "FibCompute.call1",
      on => $frFib,
      unit => $uFib,
      labels => [
        result => sub {
```

```
          $prev1 = $_[1]->getRow()->get("fib");
        }
      ],
      rowop => $lbFibCompute->makeRowopHash($op,
        idx => $idx - 1,
      ),
    );
    Triceps::FnBinding::call(
      name => "FibCompute.call2",
      on => $frFib,
      unit => $uFib,
      labels => [
        result => sub {
          $prev2 = $_[1]->getRow()->get("fib");
        }
      ],
      rowop => $lbFibCompute->makeRowopHash($op,
        idx => $idx - 2,
      ),
    );
    $res = $prev1 + $prev2;
  }
  $uFib->makeHashCall($frFib->getLabel("result"), $op,
    idx => $idx,
    fib => $res,
  );
});

# End of streaming function
###

# binding to call the Fibonacci function and print the result
my $fbFibCall = Triceps::FnBinding->new(
  name => "FibCall",
  on => $frFib,
  unit => $uFib,
  labels => [
    result => sub {
      my $row = $_[1]->getRow();
      print($row->get("fib"), " is Fibonacci number ", $row->get("idx"), "\n");
    }
  ],
);

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

The calling sequence had become different than in the looping version but the produced result is exactly the same. The streaming function now receives an argument row and produces a result row. The unit's recursion depth limit had to be adjusted to permit the recursion.

The recursive calls are done through the `FnBinding::call()`, with a closure for the result handling label. That closure can access the scope of its creator and place the result into its local variable. After both intermediate results are computed, the final result computation takes place and sends out the result row.

The `FnBinding::call()` creates a brand new binding for each call. So no matter how deep is the recursion, each function call will get a separate binding that knows how to put the results into the correct place.

If the streaming function were to return more than one rowop, the closure would have to collect them all into a variable. The further processing can not be done until the function completes. The bindings with trays cannot be used because `FnBinding::call()` disposes of the binding before it returns, so there is no chance to extract the tray from the binding. Perhaps this can be improved in the future. But there is another way to use trays that will be shown below.

And just to show yet another technique, the main loop is also different: instead of creating an AutoFnBind manually, it uses the Unit's method `callBound()` that is more compact to write and slightly more efficient. It's a great method if you have all the rowops for the call available upfront. It's first argument is a rowop or a tray or a reference to an array of rowops. The rest are the pairs of FnReturns and FnBindings. The bindings are pushed onto the FnReturns, then the rowops are called, then the bindings are popped. It replaces a whole block that would contain an AutoFnBind and the calls.

`FnBinding:call()` with closures is easy to use but it creates a closure and an FnBinding object on each run. Can things be rearranged to reuse the same objects? With some effort, they can:

```
###
# A streaming function that computes a Fibonacci number.

# Input:
#   $lbFibCompute: request to compute the number.
# Output (by FnReturn labels):
#   "result": the computed value.
# The opcode is preserved through the computation.

my @stackFib; # stack of the function states
my $stateFib; # The current state

my $frFib = Triceps::FnReturn->new(
  name => "Fib",
  unit => $uFib,
  labels => [
    result => $rtFibRes,
  ],
  onPush => sub { push @stackFib, $stateFib; $stateFib = { }; },
  onPop => sub { $stateFib = pop @stackFib; },
);

my $lbFibResult = $frFib->getLabel("result");

# Declare the label & binding variables in advance, to define them sequentially.
my ($lbFibCompute, $fbFibPrev1, $fbFibPrev2);
$lbFibCompute = $uFib->makeLabel($rtFibArg, "FibCompute", undef, sub {
  my $row = $_[1]->getRow();
  my $op = $_[1]->getOpcode();
  my $idx = $row->get("idx");

  if ($idx <= 1) {
    $uFib->makeHashCall($frFib->getLabel("result"), $op,
      idx => $idx,
      fib => $idx < 1 ? 0 : 1,
    );
  } else {
    $stateFib->{op} = $op;
    $stateFib->{idx} = $idx;

    $frFib->push($fbFibPrev1);
    $uFib->makeHashCall($lbFibCompute, $op,
      idx => $idx - 1,
    );
```

```
    }
});
$fbFibPrev1 = Triceps::FnBinding->new(
  unit => $uFib,
  name => "FibPrev1",
  on => $frFib,
  labels => [
    result => sub {
      $frFib->pop($fbFibPrev1);

      $stateFib->{prev1} = $_[1]->getRow()->get("fib");

      # must prepare before pushing new state and with it new $stateFib
      my $rop = $lbFibCompute->makeRowopHash($stateFib->{op},
        idx => $stateFib->{idx} - 2,
      );

      $frFib->push($fbFibPrev2);
      $uFib->call($rop);
    },
  ],
);
$fbFibPrev2 = Triceps::FnBinding->new(
  unit => $uFib,
  on => $frFib,
  name => "FibPrev2",
  labels => [
    result => sub {
      $frFib->pop($fbFibPrev2);

      $stateFib->{prev2} = $_[1]->getRow()->get("fib");
      $uFib->makeHashCall($frFib->getLabel("result"), $stateFib->{op},
        idx => $stateFib->{idx},
        fib => $stateFib->{prev1} + $stateFib->{prev2},
      );
    },
  ],
);

# End of streaming function
###
```

The rest of the code stays the same, so I won't copy it here.

The computation still needs to keep the intermediate results of two recursive calls. With no closures, these results have to be kept in a global object $stateFib (which refers to a hash that keeps multiple values).

But it can't just be a single object! The recursive calls would overwrite it. So it has to be built into a stack of objects, a new one pushed for each call and popped after it. This pushing and popping can be tied to the pushing and popping of the bindings on an FnReturn. When the FnReturn is defined, the options "onPush" and "onPop" define the custom Perl code to execute, which is used here for the management of the state stack.

The whole logic is then split into the sections around the calls:

- before the first call;

- between the first and second call;

- after the second call.

The first section goes as a normal label and the rest are done as bindings.

A tricky moment is that a simple scoped AutoFnBind can't be used here. The pushing of the binding happens in the calling label (such as FibCompute) but then the result is processed in another label (such as FibPrev1.result). The procedural control won't return to FibCompute until after FibPrev1.result has been completed. But FibPrev1.result needs the state popped before it can do its work! So the pushing and popping of the binding is done explicitly in two split steps: `push()` called in FibCompute and `pop()` called in FibPrev1.result. And of course then after FibPrev1.result saves the result, it pushes the next binding, which then gets popped in FibPrev2.result.

The popping can also be done without arguments, as simply `pop()`, but if it's given an argument, it will check that the binding popped is the same as its argument. This is helpful for detecting the call stack corruptions.

Now, can you guess, what depth of the unit call stack is required to compute and print the 2nd Fibonacci number? It's 7. If the tracing is enabled, it will produce this trace:

```
unit 'uFib' before label 'FibCompute' op OP_DELETE {
unit 'uFib' before label 'FibCompute' op OP_DELETE {
unit 'uFib' before label 'Fib.result' op OP_DELETE {
unit 'uFib' before label 'FibPrev1.result' (chain 'Fib.result') op
    OP_DELETE {
unit 'uFib' before label 'FibCompute' op OP_DELETE {
unit 'uFib' before label 'Fib.result' op OP_DELETE {
unit 'uFib' before label 'FibPrev2.result' (chain 'Fib.result') op
    OP_DELETE {
unit 'uFib' before label 'Fib.result' op OP_DELETE {
unit 'uFib' before label 'FibCall.result' (chain 'Fib.result') op
    OP_DELETE {
unit 'uFib' after label 'FibCall.result' (chain 'Fib.result') op
    OP_DELETE }
unit 'uFib' after label 'Fib.result' op OP_DELETE }
unit 'uFib' after label 'FibPrev2.result' (chain 'Fib.result') op
    OP_DELETE }
unit 'uFib' after label 'Fib.result' op OP_DELETE }
unit 'uFib' after label 'FibCompute' op OP_DELETE }
unit 'uFib' after label 'FibPrev1.result' (chain 'Fib.result') op
    OP_DELETE }
unit 'uFib' after label 'Fib.result' op OP_DELETE }
unit 'uFib' after label 'FibCompute' op OP_DELETE }
unit 'uFib' after label 'FibCompute' op OP_DELETE }
```

9 labels get called in a sequence, all the way from the initial call to the result printing. And only then the whole sequence unrolls back. 3 of them are chained through the bindings, so they don't push the stack frames onto the stack, and there is always the outermost stack frame, with the resulting stack depth of 9-3+1 = 7. This number grows fast. For the 6th number the number of labels becomes 75 and the frame count 51.

It happens because all the calls get unrolled into a single sequence, like what I've warned against in Section 7.7: "Topological loops" (p. 46) . The function return does unroll its FnReturn stack but doesn't unroll the unit call stack, it just goes even deeper by calling the label that processes it.

There are ways to improve it. The simplest one is to use the FnBinding with a tray, and call this tray after the function completely returns. This works out quite conveniently in two other ways too: First, AutoFnBind with its scoped approach can be used again. And second, it allows to handle the situations where a function returns not just one row but multiple of them. That will be the next example:

```
###
# A streaming function that computes a Fibonacci number.

# Input:
#   $lbFibCompute: request to compute the number.
# Output (by FnReturn labels):
#   "result": the computed value.
# The opcode is preserved through the computation.
```

```
my @stackFib; # stack of the function states
my $stateFib; # The current state

my $frFib = Triceps::FnReturn->new(
  name => "Fib",
  unit => $uFib,
  labels => [
    result => $rtFibRes,
  ],
  onPush => sub { push @stackFib, $stateFib; $stateFib = { }; },
  onPop => sub { $stateFib = pop @stackFib; },
);

my $lbFibResult = $frFib->getLabel("result");

# Declare the label & binding variables in advance, to define them sequentially.
my ($lbFibCompute, $fbFibPrev1, $fbFibPrev2);
$lbFibCompute = $uFib->makeLabel($rtFibArg, "FibCompute", undef, sub {
  my $row = $_[1]->getRow();
  my $op = $_[1]->getOpcode();
  my $idx = $row->get("idx");

  if ($idx <= 1) {
    $uFib->makeHashCall($frFib->getLabel("result"), $op,
      idx => $idx,
      fib => $idx < 1 ? 0 : 1,
    );
  } else {
    $stateFib->{op} = $op;
    $stateFib->{idx} = $idx;

    {
      my $ab = Triceps::AutoFnBind->new(
        $frFib => $fbFibPrev1
      );
      $uFib->makeHashCall($lbFibCompute, $op,
        idx => $idx - 1,
      );
    }
    $fbFibPrev1->callTray();
  }
});
$fbFibPrev1 = Triceps::FnBinding->new(
  unit => $uFib,
  name => "FibPrev1",
  on => $frFib,
  withTray => 1,
  labels => [
    result => sub {
      $stateFib->{prev1} = $_[1]->getRow()->get("fib");

      # must prepare before pushing new state and with it new $stateFib
      my $rop = $lbFibCompute->makeRowopHash($stateFib->{op},
        idx => $stateFib->{idx} - 2,
      );

      {
        my $ab = Triceps::AutoFnBind->new(
          $frFib => $fbFibPrev2
        );
        $uFib->call($rop);
```

```
      }
      $fbFibPrev2->callTray();
    },
  ],
);
$fbFibPrev2 = Triceps::FnBinding->new(
  unit => $uFib,
  on => $frFib,
  name => "FibPrev2",
  withTray => 1,
  labels => [
    result => sub {
      $stateFib->{prev2} = $_[1]->getRow()->get("fib");
      $uFib->makeHashCall($frFib->getLabel("result"), $stateFib->{op},
        idx => $stateFib->{idx},
        fib => $stateFib->{prev1} + $stateFib->{prev2},
      );
    },
  ],
);

# End of streaming function
###
```

The stack depth is now greatly reduced because the unit stack pops the frames before pushing more of them. For the 2nd Fibonacci number the trace is:

```
unit 'uFib' before label 'FibCompute' op OP_DELETE {
unit 'uFib' before label 'FibCompute' op OP_DELETE {
unit 'uFib' before label 'Fib.result' op OP_DELETE {
unit 'uFib' after label 'Fib.result' op OP_DELETE }
unit 'uFib' after label 'FibCompute' op OP_DELETE }
unit 'uFib' before label 'FibPrev1.result' op OP_DELETE {
unit 'uFib' before label 'FibCompute' op OP_DELETE {
unit 'uFib' before label 'Fib.result' op OP_DELETE {
unit 'uFib' after label 'Fib.result' op OP_DELETE }
unit 'uFib' after label 'FibCompute' op OP_DELETE }
unit 'uFib' before label 'FibPrev2.result' op OP_DELETE {
unit 'uFib' before label 'Fib.result' op OP_DELETE {
unit 'uFib' before label 'FibCall.result' (chain 'Fib.result') op
    OP_DELETE {
unit 'uFib' after label 'FibCall.result' (chain 'Fib.result') op
    OP_DELETE }
unit 'uFib' after label 'Fib.result' op OP_DELETE }
unit 'uFib' after label 'FibPrev2.result' op OP_DELETE }
unit 'uFib' after label 'FibPrev1.result' op OP_DELETE }
unit 'uFib' after label 'FibCompute' op OP_DELETE }
```

The maximal call stack depth is reduced to 5. For the 6th number the maximal required stack depth now gets reduced to only 9 instead of 51.

And there is also a way to run the recursive calls without even the need to increase the recursion depth limit. It can be left at the default 1, without `setMaxRecursionDepth()`. The secret is to fork the argument rowops to the functions instead of calling them.

```
###
# A streaming function that computes a Fibonacci number.

# Input:
#   $lbFibCompute: request to compute the number.
# Output (by FnReturn labels):
#   "result": the computed value.
```

```
# The opcode is preserved through the computation.

my @stackFib; # stack of the function states
my $stateFib; # The current state

my $frFib = Triceps::FnReturn->new(
  name => "Fib",
  unit => $uFib,
  labels => [
    result => $rtFibRes,
  ],
  onPush => sub { push @stackFib, $stateFib; $stateFib = { }; },
  onPop => sub { $stateFib = pop @stackFib; },
);

my $lbFibResult = $frFib->getLabel("result");

# Declare the label & binding variables in advance, to define them sequentially.
my ($lbFibCompute, $fbFibPrev1, $fbFibPrev2);
$lbFibCompute = $uFib->makeLabel($rtFibArg, "FibCompute", undef, sub {
  my $row = $_[1]->getRow();
  my $op = $_[1]->getOpcode();
  my $idx = $row->get("idx");

  if ($idx <= 1) {
    $uFib->fork($frFib->getLabel("result")->makeRowopHash($op,
      idx => $idx,
      fib => $idx < 1 ? 0 : 1,
    ));
  } else {
    $stateFib->{op} = $op;
    $stateFib->{idx} = $idx;

    $frFib->push($fbFibPrev1);
    $uFib->fork($lbFibCompute->makeRowopHash($op,
      idx => $idx - 1,
    ));
  }
});
$fbFibPrev1 = Triceps::FnBinding->new(
  unit => $uFib,
  name => "FibPrev1",
  on => $frFib,
  labels => [
    result => sub {
      $frFib->pop($fbFibPrev1);

      $stateFib->{prev1} = $_[1]->getRow()->get("fib");

      # must prepare before pushing new state and with it new $stateFib
      my $rop = $lbFibCompute->makeRowopHash($stateFib->{op},
        idx => $stateFib->{idx} - 2,
      );

      $frFib->push($fbFibPrev2);
      $uFib->fork($rop);
    },
  ],
);
$fbFibPrev2 = Triceps::FnBinding->new(
  unit => $uFib,
```

```
    on => $frFib,
    name => "FibPrev2",
    labels => [
      result => sub {
        $frFib->pop($fbFibPrev2);

        $stateFib->{prev2} = $_[1]->getRow()->get("fib");
        $uFib->fork($frFib->getLabel("result")->makeRowopHash($stateFib->{op},
          idx => $stateFib->{idx},
          fib => $stateFib->{prev1} + $stateFib->{prev2},
        ));
      },
    ],
);


# End of streaming function
###
```

This is a variation of the pre-previous example, with the split push and pop. The split is required for the fork to work: when the forked rowop executes, the calling label has already returned, so obviously the scoped approach won't work.

In this version the unit stack depth required to compute the 6th (and any) Fibonacci number reduces to 2: it's really only one level on top of the outermost frame.

If you were to attempt taking the advantage of the techniques from both of the last two examples (the one with the trays and the one with the forks) at the same time, that combination won't work. They could be combined but the combination just doesn't work right.

The problem is that the example with trays relies on the recursive function being completed before the tray gets called. But if the recursive functions are forked, things break. Looking at why they break provides another insight into the works of recursion. The example would look approximately like this in pseudo-code:

```
Compute:
  if (idx <= 1) {
    call FrFib Result;
  } else {
    push FibPrev1 to FrFib;
    fork Compute for n-1;
    fork Followup1;
  }

Followup1:
  fork tray of FibPrev1;

FibPrev1.result:
  pop FibPrev1 from FrFib;
  push FibPrev2 to FrFib;
  fork Compute for n-2;
  fork Followup2;

Followup2:
  fork tray of FibPrev2;

FibPrev2.result:
  pop FibPrev2 from FrFib;
  call FrFib Result;
```

The Followup labels are required because the trays with the intermediate results won't call themselves. They need to be called (or in this case, forked) by something else. The FnBinding has no method `forkTray()` but it can be done manually by first swapping the tray and then forking the result. The result FnReturn has to be called, not forked, so that it would immediately deposit the result rowop into the bound tray.

If there were only one recursive call, it would still work because the execution frame after the label `Compute(n)` returns would then look like this:

```
Compute(n-1)
Followup1(n)
```

The rowop `Compute(n-1)` would be the argument for the recursive function call, and `Followup1(n)` would be the follow-up rowop. When the execution time comes, the rowop `Compute(n-1)` executes, places the result into the tray. Then the rowop `Followup1(n)` executes and forks the tray, with the next rowop `FibPrev1.result(n)` then executing in order. So far so good.

Now let's trace the recursion to the depth of two. The first level starts the same:

```
Compute(n-1)
Followup1(n)
```

Then `Compute(n-1)` executes and forks the second level of recursion, the frame becoming:

```
Followup1(n)
Compute(n-1-1)
Followup1(n-1)
```

Do you see what went wrong? The unit execution frames are FIFO. So the second level of recursion got queued after the follow-up of the first level. That rowop `Followup1(n)` executes next, doesn't get any return values, and everything goes downhill from there.

15.11. Streaming functions and unit boundaries

One of the examples-as-future-standard-modules I've come up with, is TQL: the Triceps Trivial Query Language (or should that be TTQL?) along with a server to execute it. A TQL server is kind of like the Sybase or StreamBase CEP server in the way that it encapsulates the CEP logic, handles the client network connections with inputs and outputs, and also lets the clients define the ad-hoc queries against the tables of both the one-time and streaming varieties. The ad-hoc capabilities of TQL are probably better than those of Sybase and StreamBase, at least comparing to the last time I've looked at them up close. TQL will be described in detail in Chapter 17: "*TQL, Triceps Trivial Query Language*" (p. 333) but right now I want to look at only one aspect of its implementation.

When you're building the execution model of an ad-hoc query, you'd obviously need to take it apart after its work is done. The easy way to do so is by building it in its own unit. Then this unit can be disposed of as, well, a unit, and guarantee that nothing will leak. By the way, that is the answer to the question of why would someone want to use multiple units in the same thread: for modular disposal. So far so good, but it means that the data sources in the "main" unit need to be connected with the processing labels in the units of the ad-hoc queries.

But the labels in the main unit and the query unit can't be directly connected. A direct connection would create the stable references, and the disposal won't work. That's where the streaming function interface comes to the rescue: it provides a temporary connection. Build the query unit, build a binding for it, push the binding onto the FnReturn of the main unit, run the query, pop the binding, dispose of the query unit.

And the special capacity (or if you will, superpower) of the streaming functions that allows all that is that the FnReturn and FnBinding don't have to be of the same unit. They may be of the different units and will still work together fine.

TQL was really developed as a showcase of this feature but it has gained a life of its own and I don't want to go into all details here. Instead lets have a high-level overview and then dive straight into the part that uses the streaming functions.

To start a TQL server, you build a Triceps model as usual, and then create an object of class Triceps::X::Tql using the endpoints of that model (inputs, outputs, queryable tables) as arguments. After that the object runs the server and handles the clients until it's asked to stop.

The TQL queries are pipelines. You read the data from a table, then select, project, join (in any order, and possibly repeatedly) and eventually print the result (that is, send it back to the client over the socket).

The reading from a table is done through its dump label. When the Tql object is created, it builds an FnReturn with the dump labels of all the tables given to it. When an ad-hoc query is created, its head of the pipeline gets a matching FnBinding that is then pushed onto the FnReturn, the table gets dumped and flows through the binding into the query.

Now let's take a look at the code. I'll be skipping over the code that is less interesting, you can find the full version in the source code in `lib/Triceps/X/Tql.pm` as always. The constructor is one of these things to be skipped. The initialization part is more interesting. I've cut out the part that supports the multi-threaded logic, and the remaining single-threaded version goes as follows:

```
sub initialize # ($self)
{
  my $myname = "Triceps::X::Tql::initialize";
  my $self = shift;

  return if ($self->{initialized});

  my $owner = $self->{trieadOwner};
  if (defined $owner) {
    # ... multithreaded version ...
  } else {
    my %dispatch;
    my @labels;
    for (my $i = 0; $i <= $#{$self->{tables}}; $i++) {
      my $name = $self->{tableNames}[$i];
      my $table = $self->{tables}[$i];

      confess "$myname: found a duplicate table name '$name', all names are: "
          . join(", ", @{$self->{tableNames}})
        if (exists $dispatch{$name});

      $dispatch{$name} = $table;
      push @labels, $name, $table->getDumpLabel();
    }

    $self->{dispatch} = \%dispatch;
    $self->{fret} = Triceps::FnReturn->new(
      name => $self->{name} . ".fret",
      labels => \@labels,
    );
  }

  $self->{initialized} = 1;
}
```

It creates a dispatch hash of name-to-table and also an FnReturn that contains the dump labels of all the tables.

The method `compileQuery()` then handles the creation of the separate unit with its contents ("facet" is a term from the multithreading support, just ignore it for now):

```
# The common query compilation for the single-threaded and multi-threaded versions.
#
# The options are:
#
# qid => $id
# (optional) The query id that will be used to report any service information
# such as errors, end of dump portion and such.
# Default: ''.
#
# qname => $name
# The query name that will be used as a label name for all the
# produced data, and for the service information too.
#
```

```
# nxprefix => $name
# (optional) Prefix for the created unit name.
# Default: ''.
#
# text => $query_text
# Text of the query, in the braced format.
#
# subError => \&error($id, $qname, $msg, $error_code, $error_val)
# The function that will handle the error reporting. The args are:
#    $id and $qname as received in the options
#    $msg - the full human-readable message
#    $error_code - the string identifying the error
#    $error_val - the particular value that caused the error
#
# tables => { $name => $table, ... }
# The tables list for the single-threaded version.
# Not used with the multithreaded version.
#
# fretDumps => $fnReturn
# The FnReturn object for dumps in the single-threaded version.
# Not used with the multithreaded version.
#
# faOut => $facet
# The facet used to send the data to the Tql thread.
# Not used with the single-threaded version.
#
# faRqDump => $facet
# The facet used to send the table dump requests back to the app core.
# Not used with the single-threaded version.
#
# subPrint => \&print($text)
# The function that prints the text back to the socket.
# Not used with the single-threaded version.
#
# @return - undef on error, the compiled context object on success
#           (see the definition of its contents inside the function)
sub compileQuery # (@opts)
{
  my $myname = "Triceps::X::Tql::compileQuery";
  my $opts = {};
  &Triceps::Opt::parse("chatSockWriteT", $opts, {
    qid => [ '', undef ],
    qname => [ undef, \&Triceps::Opt::ck_mandatory ],
    nxprefix => [ '', undef ],
    text => [ undef, \&Triceps::Opt::ck_mandatory ],
    subError => [ undef, sub { &Triceps::Opt::ck_mandatory; &Triceps::Opt::ck_ref(@_,
 "CODE"); } ],
    tables => [ undef, sub { &Triceps::Opt::ck_ref(@_, "HASH", "Triceps::Table"); } ],
    fretDumps => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::FnReturn"); } ],
    faOut => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Facet"); } ],
    faRqDump => [ undef, sub { &Triceps::Opt::ck_ref(@_, "Triceps::Facet"); } ],
    subPrint => [ undef, sub { &Triceps::Opt::ck_ref(@_, "CODE"); } ],
  }, @_);

  my $q = $opts->{qname}; # the name of the query itself

  my @cmds = split_braced($opts->{text});
  if ($opts->{text} ne '') {
    &{$opts->{subError}}($opts->{qid}, $q, "mismatched braces in the trailing " . $opts-
>{text},
      'query_syntax', $opts->{text});
```

```
      return undef;
  }

  # The context for the commands to build up an execution of a query.
  # Unlike $self, the context is created afresh for every query.
  my $ctx = {};
  $ctx->{qid} = $opts->{qid};
  $ctx->{qname} = $opts->{qname};

  $ctx->{tables} = $opts->{tables};
  $ctx->{fretDumps} = $opts->{fretDumps};
  $ctx->{actions} = []; # code that will run the pipeline

  $ctx->{faOut} = $opts->{faOut};
  $ctx->{faRqDump} = $opts->{faRqDump};
  $ctx->{subPrint} = $opts->{subPrint};
  $ctx->{requests} = []; # dump and subscribe requests that will run the pipeline
  $ctx->{copyTables} = []; # the tables created in this query
    # (have to keep references to the tables or they will disappear)

  # The query will be built in a separate unit
  $ctx->{u} = Triceps::Unit->new($opts->{nxprefix} . "${q}.unit");
  $ctx->{prev} = undef; # will contain the output of the previous command in the pipeline
  $ctx->{id} = 0; # a unique id for auto-generated objects
  # deletion of the context will cause the unit in it to clean
  $ctx->{cleaner} = $ctx->{u}->makeClearingTrigger();

  if (! eval {
    foreach my $cmd (@cmds) {
      my @args = split_braced($cmd);
      my $argv0 = bunescape(shift @args);
      # The rest of @args do not get unquoted here!
      die "No such TQL command '$argv0'\n" unless exists $tqlDispatch{$argv0};
      # do something better with the errors, show the failing command...
      $ctx->{id}++;
      &{$tqlDispatch{$argv0}}($ctx, @args);
      # Each command must set its result label (even if an undef) into
      # $ctx->{next}.
      die "Internal error in the command $argv0: missing result definition\n"
        unless (exists $ctx->{next});
      $ctx->{prev} = $ctx->{next};
      delete $ctx->{next};
    }
    if (defined $ctx->{prev}) {
      # implicitly print the result of the pipeline, no options
      &{$tqlDispatch{"print"}}($ctx);
    }

    1; # means that everything went OK
  }) {
    &{$opts->{subError}}($opts->{qid}, $q, "query error: $@", 'bad_query', '');
    return undef;
  }

  return $ctx;
}
```

Each TQL command is defined as its own method, all of them collected in the `%tqlDispatch`. `compileQuery()` splits the pipeline and then lets each command build its part of the query, connecting them through `$ctx`. A command may also register an action to be run later. After everything is built, the actions run and produce the result.

The TQL syntax uses braces for the grouping in the pipeline. The functions `split_braced()` and `bunescape()` are imported from the package Triceps::Braced that handles the parsing of the braced nested lists. They are described in detail in Section 19.14: "Braced reference" (p. 386) .

The option "subError" defines a function that reports the errors back to the user. Since everything is returned back as a stream, the errors are reported as rowops on the special label "+ERROR".

And the final part of the puzzle, here is the "read" command handler that creates the head of the query pipeline:

```
# "read" command. Defines a table to read from and starts the command pipeline.
# Options:
# table - name of the table to read from.
sub _tqlRead # ($ctx, @args)
{
  my $ctx = shift;
  die "The read command may not be used in the middle of a pipeline.\n"
    if (defined($ctx->{prev}));
  my $opts = {};
  &Triceps::Opt::parse("read", $opts, {
    table => [ undef, \&Triceps::Opt::ck_mandatory ],
  }, @_);

  my $tabname = bunescape($opts->{table});
  my $unit = $ctx->{u};

  if ($ctx->{faOut}) {
    # ... multithreaded version ...
  } else {
    my $fret = $ctx->{fretDumps};

    die ("Read found no such table '$tabname'\n")
      unless (exists $ctx->{tables}{$tabname});
    my $table = $ctx->{tables}{$tabname};
    my $lab = $unit->makeDummyLabel($table->getRowType(), "lb" . $ctx->{id} . "read");
    $ctx->{next} = $lab;

    my $code = sub {
      Triceps::FnBinding::call(
        name => "bind" . $ctx->{id} . "read",
        unit => $unit,
        on => $fret,
        labels => [
          $tabname => $lab,
        ],
        code => sub {
          $table->dumpAll();
        },
      );
    };
    push @{$ctx->{actions}}, $code;
  }
}
```

It's the only command that registers an action, which sends data into the query unit. The rest of commands just add more handlers to the pipeline in the unit, and get the data that flows from "read". The action sets up a binding and calls the table dump, to send the data into that binding.

The reading of the tables could have also been done without the bindings, and without the need to bind the units at all: just iterate through the table procedurally in the action. But this whole example has been built largely to showcase that the bindings can be used in this way, so naturally it uses bindings.

The bindings come more useful when the query logic has to react to the normal logic of the main unit, such as in the subscriptions: set up the query, read its initial state, and then keep reading as the state gets updated. But guess what, the subscriptions can't be done with the FnReturns as shown because the FnReturn only sends its data to the last binding pushed onto it. This means, if multiple subscriptions get set up, only the last one will be getting the data. This problem gets solved only in the multithreaded implementation of Tql that will be discussed in Section 17.6: "Internals of a TQL join" (p. 345) . There each client runs in its own thread, and each of its queries runs in its own unit; the inter-thread communications are used to subscribe to the updates.

15.12. The ways to call a streaming function

The examples in this chapter have shown many ways to call a streaming function. Here is a recap of them all:

- Manually push the FnBinding onto the FnReturn, send the argument rowops to the streaming function, pop the FnBinding.

- Use an AutoFnBind to handle the pushing and popping in a scoped fashion. A single AutoFnBind can control multiple pairs of FnBinding and FnReturn, so it can build in one go not only a single streaming function call but even a whole pipeline. In the C++ API a more low-level object ScopedFnBind can also be used in a similar way.

- Use `Unit::callBound()` that takes care of creating both the AutoFnBind object and a scope around of it in a more efficient way.

- Use `FnBinding::call()` to create an FnBinding object dynamically, do a call with it, and dispose of it.

The most convenient way depends on the situation.

15.13. The gritty details of streaming functions scheduling

If you've read carefully about all the gritty details of scheduling, you might wonder, what exactly happens when a label in an FnBinding gets called through an FnReturn? The answer is, they are executed like the chained labels, reusing the frame of the parent label (that is, of the matching label on the FnReturn side). They even show in the traces as the chained labels. This lets the bound labels to easily fork a rowop to the frame of its parent.

The only exception is when the FnReturn and FnBinding are in the different units. Then the bound label is properly called with its own frame in the unit where it belongs.

And of course the rowops collected in a tray are another exception, since they are not called in the binding, they are only collected. When the tray gets called, they get properly called with their own frames, just as when calling any other tray.

Chapter 16. Multithreading

16.1. Triceps multithreading concepts

When running the CEP models, naturally the threads have to be connected by the queues for the data exchange. The use of queues is extremely popular but also notoriously bug-prone.

The idea of the multithreading support in Triceps is to make writing the multithreaded model easier. To make writing the good code easy and writing the bad code hard. But of course you don't have to use it, if it feels too constraining, you can always make your own.

The diagram in Figure 16.1 shows all the main elements of a multithread Triceps application.
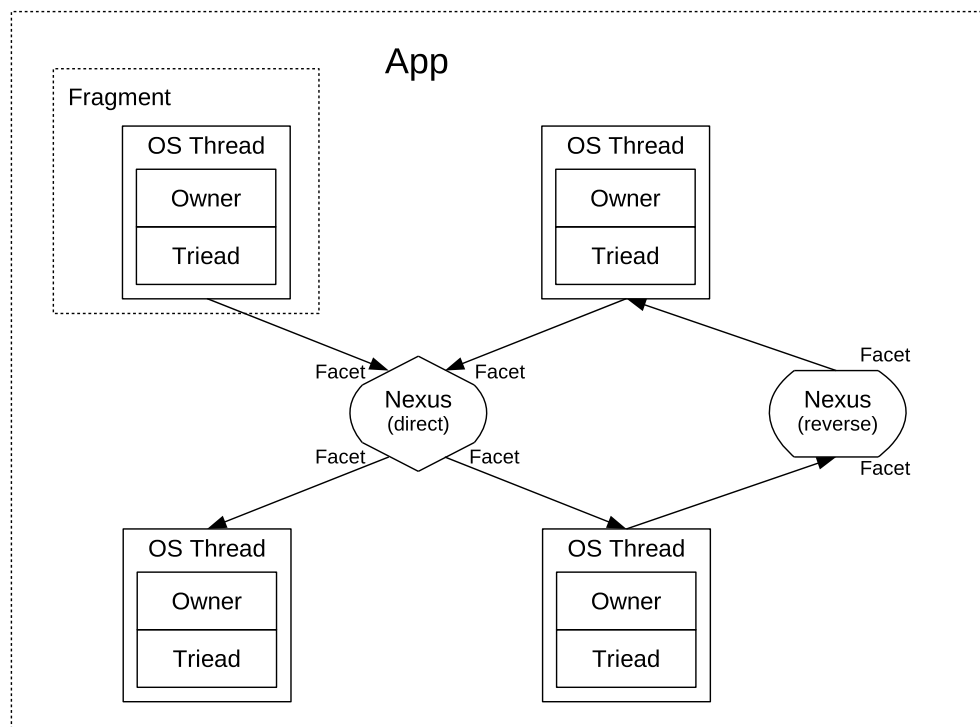
Figure 16.1. Triceps multithreaded application.

The Triceps application is embodied in the class App. It's possible to have multiple Apps in one program.

Each thread has multiple parts to it. First, of course, there is the OS-level (or, technically, library-level, or Perl-level) thread where the code executes. And then there is a class that represents this thread and its place in the App. To reduce the naming conflict, this class is creatively named Triead (pronounced still "thread"). In the discussion I use the word "thread" for both concepts, the OS-level thread and the Triead, and it's usually clear from the context which one I mean. But sometimes it's particularly important to make the distinction, and then I name one or the other explicitly.

The class Triead itself is largely opaque, allowing only a few methods for introspection. But there is a control interface to it, called TrieadOwner. The Triead is visible from the outside, the TrieadOwner object is visible only in the OS thread that owns the Triead. The TrieadOwner manages the thread state and acts as the intermediary in the thread's communications with the App.

The data is passed between the threads through the Nexuses. A Nexus is unidirectional, with data going only one way, however it may have multiple writers and multiple readers. All the readers see the exact same data, with rowops going in the exact same order (well, there will be other policies in the future as well, but for now there is only one policy).

A Nexus passes through the data for multiple labels, very much like an FnReturn does (and indeed there is a special connection between them). A Nexus also allows to export the row types and table types from one thread to another.

A Nexus is created by one thread, and then the other threads connect to it. The thread that creates the Nexus determines what labels will it contain, and what row types and table types to export.

A Nexus gets connected to the Trieads through the Facets (in the diagram, the Facets are shown as flat spots on the round Nexuses). A Facet is a connection point between the Nexus and the Triead. Each Facet is for either reading or writing. And there may be only one Facet between a given Nexus and a given Triead, you can't make multiple connections between them. As a consequence, a thread can't both write and read to the same Nexus, it can do only one thing. This might actually be an overly restrictive limitation and might change in the future but that's how things work now.

Each Nexus also has a direction: either direct ("downwards") or reverse ("upwards"). How does it know, which direction is down and whih is up? It doesn't. You tell it by designating a Nexus one way or the other. And yes, the reverse Nexuses allow to build the models with loops. However the loops consisting of only the direct Nexuses are not allowed, nor of only reverse Nexuses. They would mess up the flow control. The proper loops must contain a mix of direct and reverse Nexuses.

The direct Nexuses have a limited queue size and stop the writers when the queue fills up, until the data gets consumed, thus providing the flow control. The reverse Nexuses have an unlimited queue size, which allows to avoid the circular deadlocks. The reverse Nexuses also have a higher priority: if a thread is reading from a direct Nexus and a reverse one, with both having data available, it will read the data from the reverse Nexus first. This is to prevent the unlimited queues in the reverse Nexuses from the truly unlimited growth.

Normally an App is built once and keeps running in this configuration until it stops. But there is a strong need to have the threads dynamically added and deleted too. For example, if the App running as a server, and clients connect to it, each client needs to have its thread(s) added when the client connects and then deleted when the client disconnects. This is handled through the concept of fragments. There is no Fragment class but when you create a Triead, you can specify a fragment name for it. Then it becomes possible to shut down and dispose the threads in a fragment after the fragment's work is done.

16.2. The Triead lifecycle

Each Triead goes through a few stages in its life:

- declared

- defined

- constructed

- ready

- waited ready

- requested dead

- dead

Note by the way that it's the stages of the Triead object. The OS-level thread as such doesn't know much about them, even though these stages do have some connections to its state.

These stages always go in order and can not be skipped. However for convenience you can request a move directly to a further stage. This will just automatically pass through all the intermediate stages. Although, well, there is one exception: the "waited ready" and "requested dead" stages can get skipped on the way to "dead". Other than that, there is always the

sequence, so if you find out that a Triead is dead, you can be sure that it's also declared, defined, constructed and ready. The attempts to go to a previous stage are silently ignored.

Now, what do these stages mean?

Declared:
> The App knows the name of the thread and that this thread will eventually exist. When an App is asked to find the resources from this thread (such as Nexuses, and by the way, the Nexuses are associated with the threads that created them) it will know to wait until this thread becomes constructed, and then look for the resources. It closes an important race condition: the code that defines the Triead normally runs in a new OS thread but there is no way to tell when exactly will it run and do its work. If you had spawned a new thread and then attempted to get a nexus from it before it actually runs, the App would tell you that there is no such thread and fail. To get around it, you declare the thread first and then start it. Most of the time there is no need to declare explicitly, the library code that wraps the thread creation does it for you.

Defined:
> The Triead object has been created and connected to the App. Since this is normally done from the new OS thread, it also implies that the thread is running and is busy about constructing the nexuses and whatever its own internal resources.

Constructed:
> The Triead had constructed and exported all the nexuses that it planned to. This means that now these nexuses can be imported by the other threads (i.e. connected to the other threads). After this point the thread can not construct any more nexuses. However it can keep importing the nexuses from the other threads. It's actually a good idea to do all your exports, mark the thread constructed, and only then start importing. This order guarantees the absence of initialization deadlocks (which would be detected and will cause the App to be aborted). There are some special cases when you need to import a nexus from a thread that is not fully constructed yet, and it's possible, but requires more attention and a special override of the "immediate" import. This is described in more detail in Section 19.20: "TrieadOwner reference" (p. 405) , with the method `importNexus()`.

Ready:
> The thread had imported all the nexuses it wanted and fully initialized all its internals (for example, if it needs to load data from a file, it might do that before telling that it's ready). After this point no more nexuses can be imported. A fine point is that the other threads may still be created, and they may do their exporting and importing, but once a thread is marked as ready, it's cast in bronze. And in the simple cases you don't need to worry about separating the constructed and ready stages, just initialize everything and mark the thread as ready.

Waited ready:
> Before proceeding further, the thread has to wait for all the threads in App to be ready, or it would lose data when it tries to communicate with them. It's essentially a barrier. Normally both the stages "ready" and "waited ready" are advanced to with a single call `readyReady()`. With it the thread says "I'm ready, and let me continue when everyone is ready". After that the actual work can begin. It's still possible to create more threads after that (normally, parts of the transient fragments), and until they all become ready, the App may temporarily become unready again, but that's a whole separate advanced topic that will be discussed in Section 16.6: "Dynamic threads and fragments in a socket server" (p. 304) .

Requested dead:
> This is the way to request a thread to exit. Normally some control thread will decide that the App needs to exit and will request all its threads to die. The threads will get these requests, perform their last rites and exit. The threads don't have to get this request to exit, they can also always decide to exit on their own. When a thread is requested to die, all the data communication with it stops. No more data will get to it through the nexuses and any data it sends will be discarded. It might churn a little bit through the data in its input buffers but any results produced will be discarded. The good practice is to make sure that all the data is drained before requesting a thread to die. Note that the nexuses created by this thread aren't affected at all, they keep working as usual. It's the data connections between this thread and any nexuses that get broken.

Dead:

The thread had completed its execution and exited. Normally you don't need to mark this explicitly. When the thread's main function returns, the library will do it for you. Marking the thread dead also drives the harvesting of the OS threads: the harvesting logic will perform a `join()` (not to be confused with SQL join) of the thread and thus free the OS resources. The dead Trieads are still visible in the App (except for some special cases with the fragments), and their nexuses continue working as usual (even including the special cases with the fragments), the other threads can keep communicating through them for as long as they want.

16.3. Multithreaded pipeline

The multithreaded models are well suited for running the pipelines, so that is going to be the first example of the threads. The full text of the example can be found in `t/xTrafficAggMt.t` in the class Traffic1. It's a variation of an already shown example, the traffic data aggregation from Section 13.2: "Periodic updates" (p. 225) . The short recap is that it gets the data for each network packet going through and keeps it for some time, aggregates the data by the hour and keeps it for a longer time, and aggregates it by the day and keeps for a longer time yet. This multi-stage computation naturally matches the pipeline approach.

Since this new example highlights different features than the original one, I've changed it logic a little: it updates both the hourly and daily summaries on every packet received. And I didn't bother to implement the part with the automatic cleaning of the old data, it doesn't add anything interesting to the pipeline works.

The pipeline topologies are quite convenient for working with the threads. The parallel computations create a possibility of things happening in an unpredictable order and producing unpredictable results. The pipeline topology allows the parallelism and at the same time also keeps the data in the same predictable order, with no possibility of rows overtaking each other.

The computation in this example is split into the following threads:

- Read the input, parse and send the data into the model.

- Store the recent data and aggregate it by the hour.

- Store the hourly data and aggregate it by the day.

- Store the daily data.

- Get the data at the end of the pipeline and print it.

The result of each aggregation gets stored in a table in the next thread, which then uses the same table for the next stage of aggregation.

Technically, each stage only needs the data from the previous stage, but to get the updates to the printing stage (since we want to print the original updates, daily and hourly), they all go all the way through.

Dumping the contents of the tables also requires some special support. Each table is local to its thread and can't be accessed from the other threads. To dump its contents, the dump request needs to be sent to its thread, which would extract the data and send it through. There are multiple ways to deal with the dump results. One is to have a special label for each table's dump and propagate it to the last stage to print. If all that is needed is text, another way is to have one label that allows to send strings is good enough, all the dumps can send the data converted to text into it, and it would go all the way through the pipeline. For this example I've picked the last approach.

And now is time to show some code. The main part goes like this:

```
Triceps::Triead::startHere(
  app => "traffic",
  thread => "print",
  main => \&printT,
);
```

The `startHere()` creates an App and starts a Triead in the current OS thread. "Here" in the method name stands for "in the current OS thread". "traffic" is the app name, "print" the thread name. This thread will be the end of the pipeline, and it will create the rest of the threads. This is a convenient pattern when the results of the model need to be fed back to the current thread, and it works out very conveniently for the unit tests. `printT()` is the body function of this printing thread:

```
sub printT # (@opts)
{
  my $opts = {};
  Triceps::Opt::parse("traffic main", $opts, {@Triceps::Triead::opts}, @_);
  my $owner = $opts->{owner};
  my $unit = $owner->unit();

  Triceps::Triead::start(
    app => $opts->{app},
    thread => "read",
    main => \&readerT,
  );
  Triceps::Triead::start(
    app => $opts->{app},
    thread => "raw_hour",
    main => \&rawToHourlyT,
    from => "read/data",
  );
  Triceps::Triead::start(
    app => $opts->{app},
    thread => "hour_day",
    main => \&hourlyToDailyT,
    from => "raw_hour/data",
  );
  Triceps::Triead::start(
    app => $opts->{app},
    thread => "day",
    main => \&storeDailyT,
    from => "hour_day/data",
  );

  my $faIn = $owner->importNexus(
    from => "day/data",
    as => "input",
    import => "reader",
  );

  $faIn->getLabel("print")->makeChained("print", undef, sub {
    print($_[1]->getRow()->get("text"));
  });
  for my $tag ("packet", "hourly", "daily") {
    makePrintLabel($tag, $faIn->getLabel($tag));
  }

  $owner->readyReady();
  $owner->mainLoop(); # all driven by the reader
}
```

`startHere()` accepts a number of fixed options plus arbitrary options that it doesn't care about by itself but passes through to the thread's main function, which are then the responsibility of the main function to parse. To reiterate, the main function gets all the options from the call of `startHere()`, both these that `startHere()` parses and these that it simply passes through. `startHere()` also adds one more option on its own: "owner" containing the TrieadOwner object that the thread uses to communicate with the rest of the App.

In this case `printT()` doesn't have any extra options on its own, it's just happy to get `startHere()`'s standard set that it takes all together from `@Triceps::Triead::opts`.

It gets the TrieadOwner object `$owner` from the option appended by `startHere()`. Each TrieadOwner is created with its own Unit, so the unit is obtained from it to create the thread's model in it. Incidentally, the TrieadOwner also acts as a clearing trigger object for the Unit, so when the TrieadOwner is destroyed, it properly clears the Unit.

Then it goes and creates all the threads of the pipeline. The `start()` works very much like `startHere()`, only it actually creates a new thread and starts the main function in it. The main function can be the same whether it runs through `start()` or `startHere()`. The special catch is that the options to `start()` must contain only the plain Perl values, not Triceps objects. It has to do with how Perl works with threads: it makes a copy of every value for the new thread, and it cant's copy the XS objects, so they simply become undefined in the new thread.

All but the first thread in the pipeline have the extra option "from": it specifies the input nexus for this thread, and each thread creates an output nexus "data". A nexus it named relatively to the thread that created it, so when the option "from" says "day/data", it's the nexus "data" created by the thread "day".

So, the pipeline gets all connected sequentially until eventually `printT()` imports the nexus at its tail. `importNexus()` returns a Facet, which is the thread's API to the nexus. A facet looks very much like an FnReturn for most purposes, with a few additions. It even has a real FnReturn in it, and you work with the labels of that FnReturn to get the data out of the nexus (or to send data into the nexus). You could potentially use an FnBinding with that FnReturn but the typical pattern for reading from a facet is different: just get its labels and chain the handling labels directly to them.

The option "as" of `importNexus()` gives the name to the facet and to its same-named FnReturn (without it the facet would be named the same as the short name of the nexus, in this case "data"). The option "import" tells whether this thread will be reading or writing the nexus, and in this case it's reading.

By the time the pipeline gets to the last stage, it has a few labels in its facet:

- `print` - carries the direct text lines to print in its field `text`, and its contents gets printed.

- `dumprq` - carries the dump requests to the tables, and the printing thread doesn't care about it.

- `packet` - carries the raw data about the packets.

- `hourly` - carries the hourly summaries.

- `daily` - carries the daily summaries.

The last three get also printed but this time as whole rows.

And after everything is connected, the thread both tells that it's ready and waits for all the other threads to become ready by calling `readyReady()`. Then its the run time, and `mainLoop()` takes care of it: it keeps reading data from the nexus and processes it until it's told to shutdown. The shutdown will be controlled by the file reading thread at the start of the pipeline. The processing is done by getting the rowops from the nexus and calling them on the appropriate label in the facet, which then calls the the labels chained from it, and that gets all the rest of the thread's model running.

The reader thread drives the pipeline:

```
sub readerT # (@opts)
{
  my $opts = {};
  Triceps::Opt::parse("traffic main", $opts, {@Triceps::Triead::opts}, @_);
  my $owner = $opts->{owner};
  my $unit = $owner->unit();

  my $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
```

```
  );

  my $rtPrint = Triceps::RowType->new(
    text => "string", # the text to print (including \n)
  );

  my $rtDumprq = Triceps::RowType->new(
    what => "string", # identifies, what to dump
  );

  my $faOut = $owner->makeNexus(
    name => "data",
    labels => [
      packet => $rtPacket,
      print => $rtPrint,
      dumprq => $rtDumprq,
    ],
    import => "writer",
  );

  my $lbPacket = $faOut->getLabel("packet");
  my $lbPrint = $faOut->getLabel("print");
  my $lbDumprq = $faOut->getLabel("dumprq");

  $owner->readyReady();

  while(<STDIN>) {
    chomp;
    # print the input line, as a debugging exercise
    $unit->makeArrayCall($lbPrint, "OP_INSERT", "> $_\n");

    my @data = split(/,/); # starts with a command, then string opcode
    my $type = shift @data;
    if ($type eq "new") {
      $unit->makeArrayCall($lbPacket, @data);
    } elsif ($type eq "dump") {
      $unit->makeArrayCall($lbDumprq, "OP_INSERT", $data[0]);
    } else {
      $unit->makeArrayCall($lbPrint, "OP_INSERT", "Unknown command '$type'\n");
    }
    $owner->flushWriters();
  }

  {
    # drain the pipeline before shutting down
    my $ad = Triceps::AutoDrain::makeShared($owner);
    $owner->app()->shutdown();
  }
}
```

It starts by creating the nexus with the initial set of the labels: for the data about the network packets, for the lines to be printed at the end of the pipeline and for the dump requests to the tables in the other threads. It gets exported for the other threads to import, and also imported right back into this thread, for writing. And then the setup is done, `readyReady()` is called, and the processing starts.

It reads the CSV lines, splits them, makes a decision if it's a data line or dump request, and one way or the other sends it into the nexus. The data sent to a facet doesn't get immediately forwarded to the nexus. It's collected internally in a tray, and then `flushWriters()` sends it on. The `mainLoop()` shown in `printT` calls `flushWriters()` automatically after every tray it processes from the input. But when reading from a file you've got to do it yourself. Of course, it's more efficient to send through multiple rows at once, so a smarter implementation would check if multiple lines are available from the file and send them in larger bundles.

The last part is the shutdown. After the end of file is reached, it's time to shut down the application. You can't just shut down it right away because there still might be data in the pipeline, and if you shut it down, that data will be lost. The right way is to drain the pipeline first, and then do the shutdown when the app is drained. `AutoDrain::makeShared()` creates a scoped drain: the drain request for all the threads is started when this object is created, and the object construction completes when the drain succeeds. When the object is destroyed, that releases the drain. So in this case the drain succeeds and then the app gets shut down.

The shutdown causes the `mainLoop()` calls in all the other threads to return, and the threads to exit. Then `startHere()` in the first thread has the special logic in it that joins all the started threads after its own main function returns and before it completes. After that the script continues on its way and is free to exit.

The rest of this example might be easier to understand by looking at an example of a run first. The lines in bold are the copies of the input lines that `readerT()` reads from the input and sends into the pipeline, and `printT()` faithfully prints.

`input.packet` are the rows that reach the `printT` on the `packet` label (remember, "input" is the name with which it imports its input nexus). `input.hourly` is the data aggregated by the hour intervals (and also by the IP addresses, dropping the port information), and `input.daily` further aggregates it per day (and again per the IP addresses). The timestamps in the hourly and daily rows are truncated to the start of the hour or day.

And the lines without any prefixes are the dumps of the table contents that again reach the `printT()` through the "print" label:

```
new,OP_INSERT,1330886011000000,1.2.3.4,5.6.7.8,2000,80,100
input.packet OP_INSERT time="1330886011000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="100"
input.hourly OP_INSERT time="1330884000000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="100"
input.daily OP_INSERT time="1330819200000000" 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
input.packet OP_INSERT time="1330886012000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="50"
input.hourly OP_DELETE time="1330884000000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="100"
input.daily OP_DELETE time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="100"
input.hourly OP_INSERT time="1330884000000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
input.daily OP_INSERT time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
new,OP_INSERT,1330889612000000,1.2.3.4,5.6.7.8,2000,80,150
input.packet OP_INSERT time="1330889612000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="150"
input.hourly OP_INSERT time="1330887600000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
input.daily OP_DELETE time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
input.daily OP_INSERT time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="300"
new,OP_INSERT,1330889811000000,1.2.3.4,5.6.7.8,2000,80,300
input.packet OP_INSERT time="1330889811000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" local_port="2000" remote_port="80" bytes="300"
input.hourly OP_DELETE time="1330887600000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
input.daily OP_DELETE time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="300"
input.daily OP_INSERT time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="150"
input.hourly OP_INSERT time="1330887600000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="450"
input.daily OP_DELETE time="1330819200000000" local_ip="1.2.3.4"
```

```
    remote_ip="5.6.7.8" bytes="150"
input.daily OP_INSERT time="1330819200000000" local_ip="1.2.3.4"
    remote_ip="5.6.7.8" bytes="600"
new,OP_INSERT,1330972411000000,1.2.3.5,5.6.7.9,3000,80,200
input.packet OP_INSERT time="1330972411000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" local_port="3000" remote_port="80" bytes="200"
input.hourly OP_INSERT time="1330970400000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" bytes="200"
input.daily OP_INSERT time="1330905600000000" local_ip="1.2.3.5"
    remote_ip="5.6.7.9" bytes="200"
new,OP_INSERT,1331058811000000
input.packet OP_INSERT time="1331058811000000"
new,OP_INSERT,1331145211000000
input.packet OP_INSERT time="1331145211000000"
dump,packets
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="1330889612000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="150"
time="1330889811000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    local_port="2000" remote_port="80" bytes="300"
time="1330972411000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    local_port="3000" remote_port="80" bytes="200"
dump,hourly
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="450"
time="1330970400000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    bytes="200"
dump,daily
time="1330819200000000" local_ip="1.2.3.4" remote_ip="5.6.7.8"
    bytes="600"
time="1330905600000000" local_ip="1.2.3.5" remote_ip="5.6.7.9"
    bytes="200"
```

Note that the order of the lines is completely nice and predictable, nothing goes out of order. Each nexus preserves the order of the rows put into it, and the fact that there is only one writer per nexus and that every thread is fed from only one nexus, avoids the races.

Let's look at the thread that performs the aggregation by the hour:

```
# compute an hour-rounded timestamp (in microseconds)
sub hourStamp # (time)
{
  return $_[0]  - ($_[0] % (1000*1000*3600));
}

sub rawToHourlyT # (@opts)
{
  my $opts = {};
  Triceps::Opt::parse("traffic main", $opts, {
    @Triceps::Triead::opts,
    from => [ undef, \&Triceps::Opt::ck_mandatory ],
  }, @_);
  my $owner = $opts->{owner};
  my $unit = $owner->unit();

  # The current hour stamp that keeps being updated;
  # any aggregated data will be propagated when it is in the
```

```
    # current hour (to avoid the propagation of the aggregator clearing).
    my $currentHour;

    my $faIn = $owner->importNexus(
      from => $opts->{from},
      as => "input",
      import => "reader",
    );

    # the full stats for the recent time
    my $ttPackets = Triceps::TableType->new($faIn->getLabel("packet")->getRowType())
      ->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()
          )
        )
      )
    ;

    # type for a periodic summary, used for hourly, daily etc. updates
    my $rtSummary;

    Triceps::SimpleAggregator::make(
      tabType => $ttPackets,
      name => "hourly",
      idxPath => [ "byHour", "byIP", "group" ],
      result => [
        # time period's (here hour's) start timestamp, microseconds
        time => "int64", "last", sub {&hourStamp($_[0]->get("time"));},
        local_ip => "string", "last", sub {$_[0]->get("local_ip");},
        remote_ip => "string", "last", sub {$_[0]->get("remote_ip");},
        # bytes sent in a time period, here an hour
        bytes => "int64", "sum", sub {$_[0]->get("bytes");},
      ],
      saveRowTypeTo => \$rtSummary,
    );

    $ttPackets->initialize();
    my $tPackets = $unit->makeTable($ttPackets, "tPackets");

    # Filter the aggregator output to match the current hour.
    my $lbHourlyFiltered = $unit->makeDummyLabel($rtSummary, "hourlyFiltered");
    $tPackets->getAggregatorLabel("hourly")->makeChained("hourlyFilter", undef, sub {
      if ($_[1]->getRow()->get("time") == $currentHour) {
        $unit->call($lbHourlyFiltered->adopt($_[1]));
      }
    });

    # update the notion of the current hour before the table
    $faIn->getLabel("packet")->makeChained("processPackets", undef, sub {
      my $row = $_[1]->getRow();
      $currentHour = &hourStamp($row->get("time"));
      # skip the timestamp updates without data
      if (defined $row->get("bytes")) {
        $unit->call($tPackets->getInputLabel()->adopt($_[1]));
      }
```

```
  });

  # The makeNexus default option chainFront => 1 will make
  # sure that the pass-through data propagates first, before the
  # processed data.
  my $faOut = $owner->makeNexus(
    name => "data",
    labels => [
      $faIn->getFnReturn()->getLabelHash(),
      hourly => $lbHourlyFiltered,
    ],
    import => "writer",
  );

  my $lbPrint = $faOut->getLabel("print");

  # the dump request processing
  $tPackets->getDumpLabel()->makeChained("printDump", undef, sub {
    $unit->makeArrayCall($lbPrint, "OP_INSERT", $_[1]->getRow()->printP() . "\n");
  });
  $faIn->getLabel("dumprq")->makeChained("dump", undef, sub {
    if ($_[1]->getRow()->get("what") eq "packets") {
      $tPackets->dumpAll();
    }
  });

  $owner->readyReady();
  $owner->mainLoop(); # all driven by the reader
}
```

This function inherits the options from `Triead::start()` as usual and adds the option "from" of its own. This option's value is then used as the name of nexus to import for reading. The row types of the labels from that imported facet are then used to create the table and aggregation.

The table and aggregation themselves are the same as in Section 13.2: "Periodic updates" (p. 225) , so I won't go into much detail describing them. The only big change is the use of SimpleAggergator instead of a manually-built one. The filter logic allows to delete the old raw data without propagation of the aggregation changes caused by it.

Then the output nexus is created. The creation passes through all the incoming data, short-circuiting the input and output, and adds the extra label for the aggregated output. The call `$faIn->getFnReturn()->getLabelHash()` pulls all the labels and their names from the input facet, convenient for passing the data directly through to the output. Just like an FnReturn, the Facet construction with `makeNexus()` has the option "chainFront" set to 1 by default, and thus when it chains the labels from the pass-through ones, they are chained on the front. This works very nicely: this way the input data passes through first and only then the input goes to the computational labels and produces the results that follow it into the output facet.

The table dump is implemented after the output facet is defined because it needs the print label from that facet to send the results to. That print label ends up with two sources of datra for it. One is the eponymous label from the input facet, that passes the print requests from the previous stage of the pipeline. Another one is the table dump logic from this thread. Both are fine and can be mixed together.

And after that it's all usual `readyReady()` and `mainLoop()`.

The `hourlyToDailyT()` is very similar, so I won't even show it here, you can find the full text in the sources.

16.4. Object passing between threads

A limitation of the Perl threads is that no variables can be shared between them. Well, there are the special shared variables but they're are very special and have great many limitations on their own. When a new thread gets created, it gets a copy

of all the variables of the parent. That is, of all the plain Perl variables. With the XS extensions your luck may vary: the variables might get copied, might become undefined, or just become broken (if the XS module is not threads-aware). Copying the XS variables requires a quite high overhead at all the other times, so Triceps doesn't do it and all the Triceps object become undefined in the new thread.

This model of behavior for a package is marked by creating the method CLONE_SKIP in it:

```
sub CLONE_SKIP { 1; }
```

All the Triceps packages define it, and it's the best practice to define it in your packages as well.

However the threads are useless without communication, and Triceps provides a way to pass around certain objects through the Nexuses.

First, obviously, the Nexuses are intended to pass through the Rowops. These Rowops coming out of a nexus are not the same Rowop objects that went in. Rowop is a single-threaded object and can not be shared by two threads. Instead it gets converted to an internal form while in the nexus, and gets re-created when it comes out, pointing to the same Row object and to the correct Label in the local Facet.

Then, again obviously, the Facets get imported to the other threads as an interface of the Nexus, together with their row types.

And two more types of objects can be exported through a Nexus: the RowTypes and TableTypes. They get exported through the options as in this example:

```
$fa = $owner->makeNexus(
    name => "nx1",
    labels => [
        one => $rt1,
        two => $lb,
    ],
    rowTypes => [
        one => $rt2,
        two => $rt1,
    ],
    tableTypes => [
        one => $tt1,
        two => $tt2,
    ],
    import => "writer",
);
```

As you can see, the namespaces for the labels, row types and table types are completely independent, and the same names can be reused in each of them for different meaning. All the three sections are optional, so if you want, you can export only the types in the nexus, without any labels.

They can then be extracted from the imported facet as:

```
$rt1 = $fa->impRowType("one");
$tt1 = $fa->impTableType("one");
```

Or the whole set of name-value pairs can be obtained with:

```
@rtset = $fa->impRowTypesHash();
@ttset = $fa->impTableTypesHash();
```

The exact table types and row types (by themselves or in the table types or labels) in the importing thread will be copied. It's technically possible to share the references to the same row type from multiple threads in the C++ code but it's more efficient to make a separate copy for each thread, and thus the Perl API goes along the more efficient way.

The import is smart in the sense that it preserves the sameness of the row types: if in the exporting thread the same row type was referred from multiple places in the labels, rowTypes and tableTypes sections, in the imported facet that

would again be the same row type object (even though of course not the one that has been exported but its copy). This again helps with the efficiency when various objects decide if the rows created by this and that type are matching.

This is all well until you want to export a table type that has an index with a Perl sort condition in it, or an aggregator with the Perl code. The Perl code objects are tricky: they get copied OK when a new thread is created but the attempts to import them through a nexus later cause a terrible memory corruption. So Triceps doesn't allow to export the table types with the function references in them. But it provides an alternative solution: the code snippets can be specified as the source code, as described in Section 4.4: "Code references and snippets" (p. 21) . They get compiled when the table type gets initialized. When a table type gets imported through a nexus, it brings the source code with it. The imported table types are always uninitialized, so at initialization time the source code gets compiled in the new thread and works.

It all works transparently: just specify a string instead of a function reference when creating the index, and it will be recognized and processed. For example:

```
$it= Triceps::IndexType->newPerlSorted("b_c", undef, '
    my $res = ($_[0]->get("b") <=> $_[1]->get("b")
        || $_[0]->get("c") <=> $_[1]->get("c"));
    return $res;
    '
);
```

Before the code gets compiled, it gets wrapped into a sub { ... }, so don't write your own sub in the code string, that would be an error.

To recap the differences between the code references and the source code snippets format:

When you compile a function, it carries with it the lexical context. So you can make the closures that refer to the "my" variables in their lexical scope. With the source code you can't do this. The table type compiles them at initialization time in the context of the main package, and that's all they can see. Remember also that the global variables are not shared between the threads, so if you refer to a global variable in the code snippet and rely on a value in that variable, it won't be present in the other threads (unless the other threads are direct descendants and the value was set before their creation).

There is also the issue of arguments that can be specified for these functions. Triceps is smart enough to handle the arguments that are one of:

- undef

- integer

- floating-point

- string

- Triceps::RowType object

- Triceps::Row object

- reference to an array or hash thereof

It converts the data to an internal C++ representation in the nexus and then converts it back on import. So, if a TableType has all the code in it in the source form, and the arguments for this code within the limits of this format, it can be exported through the nexus. Otherwise an attempt to export it will fail.

The SimpleOrderedIndex uses the source code format for the functions it generates, so they will pass through the nexuses. And if you specify the aggregator functions as code snippets, you can export the table types with them through the nexuses too.

However things didn't work out so well for the SimpleAggregator. I've found that I can't just do it within the current aggregation infrastructure. As mentioned in Section 11.5: "Optimized DELETEs" (p. 155) , the aggregators don't have the

same kind of initialization function as indexes (one that would run at the table type initialization time), and that becomes the deal-breaker.

Fortunately, some thinking had showed that this feature is not really needed. There usually just isn't any need to export a table type with aggregators. So it's a nice feature to have overall but not urgent. Moreover, there is a need to export the table types with many elements stripped.

What is to be stripped and why? The most central part of the table type is its primary index. It defines how the data gets organized. And then the secondary indexes and aggregators perform the computations from the data in the table. The tables can not be shared between threads, and thus the way to copy a table between the threads is to export the table type and send the data, then let the other thread construct a copy of the table from that. But the table created in another thread really needs only the base data organization. If it does any computations on that data, that would be its own computations, different than the ones in the exporting thread. So all it needs to get is the basic table type with the primary index, very rarely some secondary indexes, and pretty much never the aggregators. The importing thread would then add its own secondary indexes and aggregators before initializing its table type and constructing the table from it.

The way to get such a stripped table type with only the fundamentally important parts is:

```
$tabtype_fundamental = $tabtype->copyFundamental();
```

That copies the row type and the primary index (the whole path to the first leaf index type) and leaves alone the rest. All the aggregators on all the indexes, even on the primary one, are not included in the copy. In the context of the full nexus, making it can look like:

```
$facet = $owner->makeNexus(
    name => "data"
    labels => [ @labels ],
    tableTypes => [
        mytable => $mytable->getType()->copyFundamental(),
    ],
    import => "writer",
);
```

In case if more index types need to be included, they can be specified by path in the arguments of copyFundamental():

```
$tabtype_fundamental = $tabtype->copyFundamental(
    [ "byDate", "byAddress", "fifo" ],
    [ "byDate", "byPriority", "fifo" ],
);
```

The paths may overlap, as shown here, and the matching subtrees will be copied correctly, still properly overlapping in the result. There is also a special syntax:

```
$tabtype_fundamental = $tabtype->copyFundamental(
    [ "secondary", "+" ],
);
```

The "+" in the path means "do the path to the first leaf index of that subtree" and saves the necessity to write out the whole path.

Finally, what if you don't want to include the original primary index at all? You can use the string "NO_FIRST_LEAF" as the first argument. That would skip it. You can still include it by using its explicit path, possibly at the other position.

For example, suppose that you have a table type with two top-level indexes, "first" is the primary index and "second" as secondary, and make a copy:

```
$tabtype_fundamental = $tabtype->copyFundamental(
    "NO_FIRST_LEAF",
    [ "second", "+" ],
    [ "first", "+" ],
);
```

In the copied table type the index "second" becomes primary and "first" secondary.

Another example of the table type exporting and of the table copying is shown in Section 17.6: "Internals of a TQL join" (p. 345) .

16.5. Threads and file descriptors

The interaction of the file descriptors (including sockets) with threads tends to be a somewhat thorny issue. The problems lie around the problem of geting a thread to stop if it's stuck reading from a file descriptor. Triceps aims to be an one-stop shop for the threading solutions, so among other things it covers the interaction with the file descriptors.

The overall approach is that whenever a thread opens a file, it should register that file with its TrieadOwner object. Then when the thread is requested to die, that file will be revoked, waking up the thread if it's waiting for that file descriptor. This is an operation that requires finesse, with multiple possibilities for the race conditions, but Triceps covers all the complexity and takes care of everything.

The registration of files with TrieadOwner is done with:

```
$to->track(*FILE);
$to->track($socket);
```

The form of the argument differs depending on whether it's a plain Perl file handle or if it's a file object, such as the one returned by the socket creation methods. The plain perl file handles have to be specified in a glob form, with *.

To unregister the file and close it, use:

```
$to->close(*FILE);
$to->close($socket);
```

If you just want to unregister a file without closing it (not sure why would you want to do that, but why not), there is also a way:

```
$to->forget(*FILE);
$to->forget($socket);
```

And there also are the methods that do the registration with the plain file descriptors:

```
$to->trackFd($fd);
$to->forgetFd($fd);
```

The file descriptor methods underlie the file handle methods, a `$to->trackFd(fileno(FILE));` is really an equivalent of `$to->track(*FILE);`.

But wait, there is more. It's annoying to close the files manually, and easy to miss too (especially if the code might die within an `eval`, with the file close sandwiched between them). The scope-based file closing is much easier and more reliable: when you leave the scope, the file is guaranteed to get closed. Triceps provides the scope-based file handle management. Actuallly, if you use files-as-objects in Perl (in the form like `$socket`), Perl already does the scope-based management, closing the file when the last reference to it disappears. But Triceps adds the automatic forgetting of the file in the TrieadOwner as well. And it's very, very important to unregister the file descriptors before closing them, or the file descriptor corruption will result when the thread exits. A scope-based file is registered like this:

```
$tf = $to->makeTrackedFile($file);
```

`$tf` will contain a TrackedFile object that will control the life of the file tracking. `$tf` contains its own reference to the file handle, so until `$tf` is destroyed, the file handle will not be destroyed and thus will not be automatically closed. When `$tf` goes out of scope and gets destroyed, it will unregister the file from TrieadOwner and then discard its reference to the file handle, letting Perl close it if all the references are gone. To find the file handle from the TrackedFile, use:

```
$file = $tf->get();
```

It's also possible to close the file explicitly before $tf goes out of scope:

```
$tf->close();
```

That will unregister the file and close it. And in this case close really means close: even if there are other references to the file handle, it will still get closed and could not be used through these other references any more either. After that, the final destruction of the TrackedFile object will have nothing to do.

In the C++ API the file registration happens a bit differently, through a FileInterrupt object that keeps track of a set of file descriptors. Each trieadOwner object has a public field `fileInterrupt_` of that class. There is no scoped unregistration in the C++ API yet, it should probably be added in the future. The Perl API actually also uses the FileInterrupt but with the high-level logic on top of it.

In case if you wonder what exactly happens with the file handle during the revocation, let me tell you. Overall it follows the approach described in my book [Babkin10], with the system call `dup2()` copying a descriptor of `/dev/null` opened read-only over the target file descriptor. Perl's file handle keeps owning that file descriptor id but now with a different contents in it. However there is a problem with the plain `dup2()`, it doesn't always interrupt the ongoing system call. I've thought that on Linux it works reliably but then I've found that it works on the sockets but not on the pipes, and even with sockets the `accept()` seems to ignore it. So I've found a better solution: use a `dup2()` but then also send a signal (Triceps uses `SIGUSR2`) to the target thread, which has a dummy handler of that signal to avoid killing the process. Even if `dup2()` gets ignored by the current system call, the signal will get through and either make the ongoing system call return `EINTR` to make the user code retry or cause a system call restart in the OS. In either case the new file descriptor copied by `dup2()` will be discovered on the next attempt and cause the desired interruption. And unlike the signal used by itself, `dup2()` closes the race window around the signal.

By the way, the Perl's `threads::kill()` doesn't send a real signal, it just sets a flag for the interpreter. If you try it on your own, it won't interrupt the system calls, and now you know why. Instead Triceps gets the POSIX thread identity from the Perl thread and calls the honest `ptherad_kill()` from the C++ code.

And another detour into the gritty details, what if the thread gets requested to die after the socket is opened but before it is tracked? The answer is that the tracking enrollment will check whether the death request already happened, and if so, it will revoke the socket right away, before returning. So the reading loop will find the socket revoked right on its first iteration. It's one of these potential race conditions that Triceps prevents.

Now returning back to the high-level Perl API. As it turns out, Perl doesn't allow to pass the file descriptors between the threads. Well, you sort of can pass them as arguments to another thread but then it ends up printing the error messages like these and corrupting the reference counts:

```
Unbalanced string table refcount: (1) for "GEN1" during global destruction.
Unbalanced string table refcount: (1) for "/usr/lib/perl5/5.10.0/Symbol.pm" during global
 destruction.
Scalars leaked: 1
```

If you try to pass a file descriptor through trheads::shared, it honestly won't allow you, while the thread arguments pretend that they can and then fail.

And it's something that is really needed, for more than one reason. If you write a TCP server, you typically have one thread accepting connections and then creating a new thread to handle each accepted socket. And the typical way to avoid polling on a socket is to have two threads handle it, one doing all the reading, another one doing all the writing. None of that works with Perl out of the box. Triceps comes to the rescue again, it gets done as follows:

```
# in one thread
$to->app()->storeCloseFile($name, $socket);

#----------

# in another thread
my ($tf, $socket) = $to->trackGetFile($name, 'r+');
```

The first thing that needs to be clarified here is that even though the variables $to are named the same in both threads, they contain different TrieadOwner objects, each one belonging to its own thread. storeCloseFile() stores a copy of the socket in the App object and then closes it in the local thread (the copy stays open). trackGetFile() pulls the socket out of the App, registers it with the TrieadOwner and creates a TrackedFile scoped object for it, returning both the TrackedFile and the socket handle object.

You can actually get the filehandle directly from the TrackedFile, as $tf->get(), so why return the socket separately? As it turns out, Perl has issues with handling the Perl values stored inside the XS objects if they aren't referred by any Perl variables. Returning the file handle as a separate value prevents that.

The file opening mode still has to be specified as a trackGetFile() argument because it can't be easily pulled out of the original file handle, and also because the other thread might want to use only a subset of the modes from the original. The mode can be specified in either of the fashions: as r/w/a/r+/w+/a+ or </>/>>/+</+>/+>>.

The $name is a unique name by which this socket is known in the App, it has to be generated in some way to guarantee the uniqueness. But then the name is a plain string that can be passed between the threads, either as an argument at the thread creation time or in a Triceps Rowop going through a nexus. The whole procedure is easy, straightforward, and difficult to get wrong.

Let's look at more variations of the same. What if you want to pass a file handle to more than one thread? It's a typical case when a TCP server accepts a connection and wants to start the separate reader and writer threads on that socket. One way is to simply pass it twice, with different names. For example:

```
# in the acceptor thread
$to->app()->storeFile('name_r', $socket);
$to->app()->storeCloseFile('name_w', $socket);


#----------


# in the reader thread
my ($tf, $socket) = $to->trackGetFile('name_r', 'r');


#----------


# in the writer thread
my ($tf, $socket) = $to->trackGetFile('name_w', 'w');
```

The storeFile() in the acceptor stores a copy of the file descriptor but doesn't close the original. Which then gets closed after storing the second copy. Then two threads extract each its own copy. In this example each thread adds its own permission limitation on the extracted file handle, one reading, another one writing, even though the underlying socket is capable of both reading and writing.

Another way is possible if the threads are started sequentially, such as if the acceptor thread starts the reader thread, which in turn starts the writer thread. Then the reader thread can load the file handle without forgetting it in the App before starting the writer thread, and then the writer thread would get it and discard from the App:

```
# in the acceptor thread
$to->app()->storeCloseFile('name', $socket);
// ... start the reader thread


#----------


# in the reader thread
my ($tf, $socket) = $to->trackDupFile('name', 'r');
// ... start the writer thread


#----------


# in the writer thread
my ($tf, $socket) = $to->trackGetFile('name', 'w');
```

Yet another way is to store the file once and load into each thread without forgetting, but then have the storing thread close the copy stored in the App after all the loading threads have completed the initialization. For example:

```
# in the acceptor thread
$to->app()->storeCloseFile('name', $socket);
// ... start the reader thread
// ... start the writer thread

$to->readyReady(); // wait for all threads to initialize
$to->app()->closeFd('name');

#----------

# in the reader thread
my ($tf, $socket) = $to->trackDupFile('name', 'r');
$to->readyReady();

#----------

# in the writer thread
my ($tf, $socket) = $to->trackDupFile('name', 'w');
$to->readyReady();
```

In the acceptor thread, `readyReady()` doesn't mark the acceptor thread as ready, because it already is, but simply waits for all other threads to become ready. And the reader and writer threads report that they are ready only after they have copied the file handle out of the App. This approach is used, for example, in Triceps::X::ThreadedClient. It would not be so great in a server because generally we want the server to accept the connections as fast as possible, without waiting for each connection's threads to initialize. But it's fine for a client.

The App's copy of the file is closed by `closeFd()`. It's an Fd and not a File because the App really stores the OS file descriptors, not the Perl file handles, and the File handling is done as wrappers on top of it. If you really want, you can deal with the raw file descriptors in the App, with the methods described in Section 19.18: "App reference" (p. 395) and Section 20.37: "TrieadOwner reference" (p. 506). But dealing directly with the Perl file handles is much more convenient.

16.6. Dynamic threads and fragments in a socket server

The threads can be used to run a TCP server that accepts the connections and then starts the new client communication thread(s) for each connection. This thread can then communicate with the rest of the model, feeding and receiving data, as usual, through the nexuses.

The challenge here is that there must be a way to create the threads dynamically, and later when the client closes connection, to dispose of them. There are two possible general approaches:

• Dynamically create and delete the threads in the same App;

• Create a new App per connection and connect it to the main App.

Both have their own advantages and difficulties, but the approach with the dynamic creation and deletion of threads ended up looking easier, and that's what Triceps has. The second approach is not particularly well supported yet. You can create multiple Apps in one program, and you can connect them by making two Triceps Trieads run in the same OS thread and ferry the data around. But it's extremely cumbersome. This will be improved in the future, but for now the first approach is the ticket.

The dynamically created threads are grouped into the fragments. This is done by specifying the fragment name option when creating a thread. The threads in a fragment have a few special properties.

One, it's possible to shut down (i.e. request to die) the whole fragment in one fell swoop. There is no user-accessible way to shut down the individual threads, you can shut down either the whole App or a fragment. Shutting down individual threads is dangerous, since it can mess up the application in many non-obvious ways. But shutting down a fragment is OK, since the fragment serves a single logical function, such as servicing one TCP connection, and it's OK to shut down the whole logical function.

Two, when a thread in the fragment exits, it's really gone, and takes all its nexuses with it. Well, technically, the nexuses continue to exist and work as long as there are threads connected to them, but no new connections can be created after this point. Since usually the whole fragment will be gone together, and since the nexuses defined by the fragment's thread are normally used only by the other threads of the same fragment, a fragment shutdown cleans up its state like the fragment had never existed. By contrast, when a normal thread exists, the nexuses defined by it stay present and accessible until the App shuts down.

To show how all this stuff works, I've created an example of a "chat server". It's not really a human-oriented chat, it's more of a machine-oriented publish-subscribe, and specially tilted to work through the running of a socket server with threads.

In this case the core logic is absolutely empty. All there is of it, is a nexus that passes messages through. The clients read from this nexus to get the messages, and write to this nexus to send the messages.

When the App starts, it has only one thread, the listener thread that listens on a socket for the incoming connections. The listener doesn't even care about the common nexus and doesn't import it. When a connection comes in, the listener creates two threads to serve it: the reader reads the socket and sends to the nexus, and the writer receives from the nexus and writes to the socket. These two threads constitute a fragment for this client. They also create their own private nexus, allowing the reader to send control messages to the writer. That could also have been done through the central common nexus, but I wanted to show that there are different ways of doing things.

With a couple of clients connected, threads and sockets start looking as shown in Figure 16.2 . And the listener thread still stays on the side.
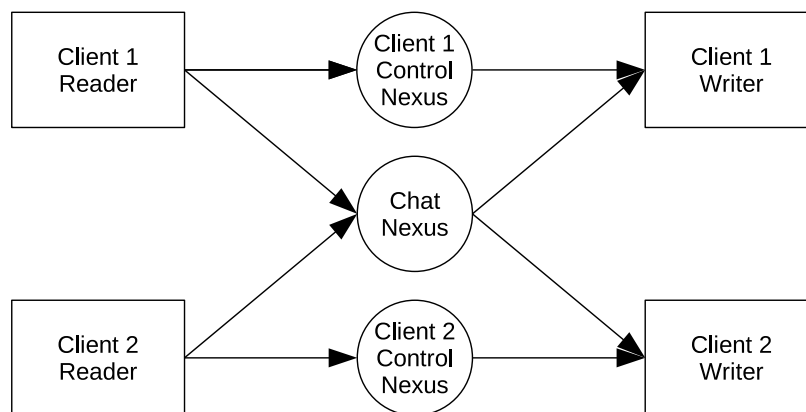
Figure 16.2. Chat server internal structure.

Now let's look at the code, located in `t/xChatMt.t`. Let's start with the top-level: how the server gets started. It's really the last part of the code, that brings everything together.

It uses the ThreadedServer infrastructure:

```
use Triceps::X::ThreadedServer qw(printOrShut);
```

The X subdirectory is for the examples and experimental stuff, but the ThreadedServer is really of production quality, I just haven't written a whole set of tests for it yet.

The server gets started like this:

```
my ($port, $pid) = Triceps::X::ThreadedServer::startServer(
    app => "chat",
    main => \&listenerT,
    port => 0,
    fork => 1,
);
```

The port option of 0 means "pick any free port", it will be returned as the result. If you know the fixed port number in advance, use it. "chat" will be the name of the App, and listenerT is the main function of the thread that will listen for the incoming connections and start the other threads. And it's also the first thread that gets started, so it's responsible for creating the core part of the App as well (though in this case there is not a whole lot of it).

The option "fork" determines how and whether the server gets started in the background. The value 1 means that a new process will be forked, and then the threads will be created there. The returned PID can be used to wait for that process to complete:

```
waitpid($pid, 0);
```

Of course, if you're starting a daemon, you'd probably write this PID to a file and then just exit the parent process.

The fork value of 0 starts the server in the current process, and the current thread becomes the App's harvester thread (the one that joins the other threads when the App shuts down).

In this case the server doesn't return until it's done, so there is not much point in the returned port value, by that time the socket will be already closed. In this case you really need to either use a fixed port or write the port number to a file from your listener thread. The PID also doesn't make sense, and it's returned as undef. Here is an example of this kind of call:

```
my ($port, $pid) = Triceps::X::ThreadedServer::startServer(
    app => "chat",
    main => \&listenerT,
    port => 12345,
    fork => 0,
);
```

Finally, the server can be started in the current process, with a new thread created as the App's harvester thread, setting the "fork" option to -1. The original thread can then continue and do other things in parallel. It's the way I use for the unit tests.

```
my ($port, $thread) = Triceps::X::ThreadedServer::startServer(
    app => "chat",
    main => \&listenerT,
    port => 0,
    fork => -1,
);
```

In this case the second value returned is not a PID but the Perl thread object for the harvester thread. You should either detach it or eventually join it:

```
$thread->join();
```

Perl propagates the errors in the threads through the join(), so if the harvester thread dies, that would show only in the join() call. And since Triceps propagates the errors too, any other App's thread dying will cause the harvester thread to die after it shuts down and joins all the App's threads. So if there are any errors in the application, it will die and you will know it right away, and not left wondering why does the application appear to run but not work right. Unless, of course, you wrap the whole thing in an eval, then you get the error message and the clean state, and can try running another App.

The listener thread is:

```perl
sub listenerT
{
  my $opts = {};
  &Triceps::Opt::parse("listenerT", $opts, {@Triceps::Triead::opts,
    socketName => [ undef, \&Triceps::Opt::ck_mandatory ],
  }, @_);
  undef @_;
  my $owner = $opts->{owner};

  my ($tsock, $sock) = $owner->trackGetFile($opts->{socketName}, "+<");

  # a chat text message
  my $rtMsg = Triceps::RowType->new(
    topic => "string",
    msg => "string",
  );

  # a control message between the reader and writer threads
  my $rtCtl = Triceps::RowType->new(
    cmd => "string", # the command to execute
    arg => "string", # the command argument
  );

  $owner->makeNexus(
    name => "chat",
    labels => [
      msg => $rtMsg,
    ],
    rowTypes => [
      ctl => $rtCtl,
    ],
    import => "none",
  );

  $owner->readyReady();

  Triceps::X::ThreadedServer::listen(
    owner => $owner,
    socket => $sock,
    prefix => "cliconn",
    handler => \&chatSockReadT,
  );
}
```

It gets the usual options of the thread start (and as usual you can pass more options to the `startServer()` and they will make their way to the listener thread. But there is also one more option added by `startServer()`: "socketName".

As described in Section 16.5: "Threads and file descriptors" (p. 301), since the the socket objects can't be passed directly between the threads, a roundabout way is taken. After `startServer()` opens a socket, it stores the dupped file descriptor in the App with a unique name and passes that name through, so that it can be used to load the socket back into the listener thread:

```perl
my ($tsock, $sock) = $owner->trackGetFile($opts->{socketName}, "+<");
```

This does multiple things:

- Loads the file descriptor from the App by name (with a `dup()`).

- Opens a Perl socket object from that file descriptor.

Dynamic threads and fragments in a socket server

- Registers that file descriptor for tracking with the TrieadOwner, so that if the thread needs to be shut down, that descriptor will be revoked and any further operations with it will fail.

- Creates a TrackedFile object that will automatically unregister the file descriptor from TrieadOwner when the Tracked-File goes out of scope. This is important to avoid that races between the revocation and the normal close of the file.

- Makes the App close and forget its file descriptor.

The `$tsock` returned is a TrackedFile object, and `$sock` is the socket filehandle.

The listener thread then creates the row types for the data messages and for the control messages between the client's reader and writer threads, and makes a nexus. The listener is not interested in the data, so it doesn't even import this nexus itself. The nexus passes the data only, so it has no label for the control messages, only the row type.

Then the mandatory `readyReady()`, and then the control goes again to the library that accepts the connections and starts the client connection threads. The handler is the main function for the thread that gets started to handle the connection. The prefix is used to build the names for the new thread, for its fragment, and for the connection socket that gets also passed through by storing it in the App. The name is the same for all three and gets created by concatenating the prefix with a number that gets increased for every connection, to keep it unique. The newly created thread will then get the option "socketName" with the name of the socket.

How does the `ThreadedServer::listen()` know when to return? It runs until the App gets shut down, and returns only when the thread is requested to die as a result of the shutdown.

As described before, each socket gets served by two threads: one runs reading from the socket and forwards the data into the model and another one runs getting data from the model and forwards it into the socket. Since the same thread can't wait for both a socket descriptor and a thread synchronization primitive, they have to be separate.

The first thread started for a connection by the listener is the socket reader. Let's go through it bit by bit.

```
sub chatSockReadT
{
  my $opts = {};
  &Triceps::Opt::parse("chatSockReadT", $opts, {@Triceps::Triead::opts,
    socketName => [ undef, \&Triceps::Opt::ck_mandatory ],
  }, @_);
  undef @_; # avoids a leak in threads module
  my $owner = $opts->{owner};
  my $app = $owner->app();
  my $unit = $owner->unit();
  my $tname = $opts->{thread};

  # only dup the socket, the writer thread will consume it
  my ($tsock, $sock) = $owner->trackDupFile($opts->{socketName}, "<");
```

The beginning is quite usual. Then it loads the socket from the App and gets it tracked with the TrieadOwner. The listener passes the name under which it stored the socket in the App as the option "socketName". `trackDupSocket()` leaves the socket instance in the App, to be found by the writer-side thread.

The socket is loaded in this thread as read-only. The writing to the socket from all the threads has to be synchronized to avoid mixing the half-messages. And the easiest way to synchronize is to always write from one thread; if the other thread wants to write something, it has to pass the data to the writer thread through the control nexus.

```
  # user messages will be sent here
  my $faChat = $owner->importNexus(
    from => "global/chat",
    import => "writer",
  );

  # control messages to the reader side will be sent here
```

```
my $faCtl = $owner->makeNexus(
  name => "ctl",
  labels => [
    ctl => $faChat->impRowType("ctl"),
  ],
  reverse => 1, # gives this nexus a high priority
  import => "writer",
);
```

Imports the chat nexus and creates the private control nexus for communication with the writer side. The name of the chat nexus is hardcoded here, since it's pretty much a solid part of the application. If this were a module, the name of the chat nexus could be passed through the options.

The control nexus is marked as reverse even though it really isn't. But the reverse option has a side effect of making this nexus high-priority. Even if the writer thread has a long queue of messages from the chat nexus, the messages from the control nexus will be read first. Which again isn't strictly necessary here, but I wanted to show how it's done.

The type of the control label is imported from the chat nexus, so it doesn't have to be defined from scratch.

```
$owner->markConstructed();

Triceps::Triead::start(
  app => $opts->{app},
  thread => "$tname.rd",
  fragment => $opts->{fragment},
  main => \&chatSockWriteT,
  socketName => $opts->{socketName},
  ctlFrom => "$tname/ctl",
);

$owner->readyReady();
```

Then the construction is done and the writer thread gets started. And then the thread becomes ready and waits for the writer thread to be ready too. The `readyReady()` works in the fragments just as it does at the start of the App. Whenever a new thread is started, the App becomes not ready, and stays this way until all the threads report that they are ready. The rest of the App keeps working like nothing happened, at least sort of. Whenever a nexus is imported, the messages from this nexus start collecting for this thread, and if there are many of them, the nexus will become backed up and the threads writing to it will block. The new threads have to call `readyReady()` as usual to synchronize between themselves, and then everything gets on its way.

Of course, if two connections are received in a quick succession, that would start two sets of threads, and `readyReady()` will continue only after all of them are ready. This is not very good but acceptable in most cases.

```
my $lbChat = $faChat->getLabel("msg");
my $lbCtl = $faCtl->getLabel("ctl");

$unit->makeHashCall($lbCtl, "OP_INSERT",
  cmd => "print", arg => "!ready," . $opts->{fragment});
$owner->flushWriters();
```

A couple of labels get remembered for the future use, and the connection ready message gets sent to the writer thread through the control nexus. By convention of this application, the messages go in the CSV format, with the control messages starting with "!". If this is the first client, this would send

`!ready,cliconn1`

to the client. It's important to call `flushWriters()` every time to get the message(s) delivered.

```
while(<$sock>) {
  s/[\r\n]+$//;
```

```
    my @data = split(/,/);
    if ($data[0] eq "exit") {
      last; # a special case, handle in this thread
    } elsif ($data[0] eq "kill") {
      eval {$app->shutdownFragment($data[1]);};
      if ($@) {
        $unit->makeHashCall($lbCtl, "OP_INSERT", cmd => "print", arg => "!error,$@");
        $owner->flushWriters();
      }
    } elsif ($data[0] eq "shutdown") {
      $unit->makeHashCall($lbChat, "OP_INSERT", topic => "*", msg => "server shutting
 down");
      $owner->flushWriters();
      Triceps::AutoDrain::makeShared($owner);
      eval {$app->shutdown();};
    } elsif ($data[0] eq "shutdown2") { # with the guarantee of the last word
      my $drain = Triceps::AutoDrain::makeExclusive($owner);
      $unit->makeHashCall($lbChat, "OP_INSERT", topic => "*", msg => "server shutting
 down");
      $owner->flushWriters();
      $drain->wait();
      eval {$app->shutdown();};
    } elsif ($data[0] eq "publish") {
      $unit->makeHashCall($lbChat, "OP_INSERT", topic => $data[1], msg => $data[2]);
      $owner->flushWriters();
    } else {
      # this is not something you want to do in a real chat application
      # but it's cute for a demonstration
      $unit->makeHashCall($lbCtl, "OP_INSERT", cmd => $data[0], arg => $data[1]);
      $owner->flushWriters();
    }
  }
```

The main loop keeps reading lines from the socket and interpreting them. The lines are in CSV format, and the first field is the command and the rest are the arguments (if any). The commands are:

publish
    Send a message with a topic to the chat nexus.

exit
    Close the connection.

kill
    Close another connection, by name.

shutdown
    Shut down the server.

subscribe
    Subscribe the client to a topic.

unsibscribe
    Unsubscribe the client from a topic.

The "exit" just exits the loop, since it works the same as if the socket just gets closed from the other side.

The "kill" shuts down by name the fragment where the threads of the other connection belong. This is a simple application, so it doesn't check any permissions, whether this fragment should allowed to be shut down. If there is no such fragment, the shutdown call will silently do nothing, so the error check and reporting is really redundant (if something goes grossly wrong in the thread interruption code, an error might still occur, but theoretically this should never happen).

The "shutdown" sends the notification to the common topic "*" (to which all the clients are subscribed by default), then drains the model and shuts it down. The drain makes sure that all the messages in the model get processed (and even written to the sockets) without allowing any new messages to be injected. "Shared" means that there is no special exceptions for some threads.

`AutoDrain::makeShared()` actually creates a drain object that keeps the drain active during its lifetime. Here this object is not assigned anywhere, so it gets immediately destroyed and lifts the drain. So potentially more messages can get squeezed in between this point and shutdown. Which doesn't matter a whole lot here.

The command "shutdown2" shows the implementation for the case when it's really important that nothing get sent after the shutdown notification.

It starts the drain in the exclusive mode, which means that this thread is excluded from the drain and allowed to send more data. When the drain is created, it waits for success, so when the new message is inserted, it will be after all the other messages. `$drain->wait()` does another wait and makes sure that this last message propagates all the way. And then the app gets shut down, while the drain is still in effect, so no more messages can be sent for sure.

The "publish" sends the data to the chat nexus (note the `flushWriters()`, as usual!).

And the rest of commands (that would be "subscribe" and "unsubscribe" but you can do any other commands like "print") get simply forwarded to the reader thread for execution. Sending through the commands like this without testing is not a good practice for a real application but it's cute for a demo.

```
  {
    # let the data drain through
    my $drain = Triceps::AutoDrain::makeExclusive($owner);

    # send the notification - can do it because the drain is excluding itself
    $unit->makeHashCall($lbCtl, "OP_INSERT", cmd => "print", arg => "!exiting");
    $owner->flushWriters();

    $drain->wait(); # wait for the notification to drain

    $app->shutdownFragment($opts->{fragment});
  }

  $tsock->close(); # not strictly necessary
}
```

The last part is when the connection get closed, either by the "exit" command or when the socket gets closed. Remember, the socket can get closed asymmetrically, in one direction, so even when the reading is closed, the writing may still work and needs to return the responses to any commands received from the socket. And of course the same is true for the "exit" command.

So here the full exclusive drain sequence is used, ending with the shutdown of this thread's own fragment, which will close the socket. Even though only one fragment needs to be shut down, the drain drains the whole model. Because of the potentially complex interdependencies, there is no way to reliably drain only a part, and all the drains are App-wide.

The last part, with `$tsock->close()`, is not technically necessary since the shutdown of the fragment will get the socket descriptor revoked anyway, and then the socket will get closed when the last reference to it disappears. But other than that, it's a good practice that unregisters the socket from the TrieadOwner and then closes it.

The socket writer thread is the last part of the puzzle:

```
sub chatSockWriteT
{
  my $opts = {};
  &Triceps::Opt::parse("chatSockWriteT", $opts, {@Triceps::Triead::opts,
    socketName => [ undef, \&Triceps::Opt::ck_mandatory ],
    ctlFrom => [ undef, \&Triceps::Opt::ck_mandatory ],
```

```
  }, @_);
  undef @_;
  my $owner = $opts->{owner};
  my $app = $owner->app();
  my $tname = $opts->{thread};

  my ($tsock, $sock) = $owner->trackGetFile($opts->{socketName}, ">");

  my $faChat = $owner->importNexus(
    from => "global/chat",
    import => "reader",
  );

  my $faCtl = $owner->importNexus(
    from => $opts->{ctlFrom},
    import => "reader",
  );
```

The usual preamble. The `trackGetFile()` consumes the socket from the App, and this time reopens it for writing. The previously created nexuses are imported.

```
  my %topics; # subscribed topics for this thread

  $faChat->getLabel("msg")->makeChained("lbMsg", undef, sub {
    my $row = $_[1]->getRow();
    my $topic = $row->get("topic");
    #printOrShut($app, $opts->{fragment}, $sock, "XXX got topic '$topic'\n");
    if ($topic eq "*" || exists $topics{$topic}) {
      printOrShut($app, $opts->{fragment}, $sock, $topic, ",", $row->get("msg"), "\n");
    }
  });
```

The logic is defined as the connected labels. The topic hash keeps the keys that this thread is subscribed to. When a message is received from the chat nexus and the topic is in the hash or is "*", the message gets sent into the socket in the CSV format:

`topic,text`

The function `printOrShut()` is imported from Triceps::X::ThreadedServer. Its first 3 arguments are fixed, and the rest are passed through to `print()`. It prints the message to the socket file handle, flushes the socket, and in case of any errors it shuts down the fragment specified in its second argument. This way if the socket gets closed from the other side, the threads handling it automatically shut down.

```
  $faCtl->getLabel("ctl")->makeChained("lbCtl", undef, sub {
    my $row = $_[1]->getRow();
    my ($cmd, $arg) = $row->toArray();
    if ($cmd eq "print") {
      printOrShut($app, $opts->{fragment}, $sock, $arg, "\n");
    } elsif ($cmd eq "subscribe") {
      $topics{$arg} = 1;
      printOrShut($app, $opts->{fragment}, $sock, "!subscribed,$arg\n");
    } elsif ($cmd eq "unsubscribe") {
      delete $topics{$arg};
      printOrShut($app, $opts->{fragment}, $sock, "!unsubscribed,$arg\n");
    } else {
      printOrShut($app, $opts->{fragment}, $sock, "!invalid command,$cmd,$arg\n");
    }
  });
```

The handling of the control commands is pretty straightforward.

```
  $owner->readyReady();
```

```
    $owner->mainLoop();

    $tsock->close(); # not strictly necessary
}
```

And the rest is taken care of by the `mainLoop()`. The thread's main loop runs until the thread gets shut down, by handling the incoming messages. So if say `printOrShut()` decides to shut down the fragment, the next iteration of the loop will detect it and exit.

And now a recorded log from a run. It has been produced with the automated testing infrastructure described in Section 16.8: "ThreadedClient, a Triceps Expect" (p. 318) .

As usual, the lines sent from the clients to the socket server are shown in bold. But since there are many clients, to tell them apart, both the sent and received lines are prefixed by the client's name and a "|". I've just picked arbitrary client names to tell them apart.

I've also marked the incoming connection as "**connect client_name**", the client's close of the socket for writing as "**close WR client_name**", and the disconnections as "**__EOF__**" after the client name.

So, here we go.

connect c1
c1|!ready,cliconn1
connect c2
c2|!ready,cliconn2

Two clients connect.

c1|publish,*,zzzzzz
c1|*,zzzzzz
c2|*,zzzzzz

A message published to the topic "*" gets forwarded to all the connected clients. In reality the messages may of course be received on separate sockets in any order, but I've ordered them here for the ease of reading.

c2|garbage,trash
c2|!invalid command,garbage,trash

An invalid command gets detected in the writer thread and responded as such.

c2|subscribe,A
c2|!subscribed,A
c1|publish,A,xxx
c2|A,xxx

A subscription request gets acknowledged, and after that all the messages sent to this topic get received by the client.

c1|subscribe,A
c1|!subscribed,A
c1|publish,A,www
c1|A,www
c2|A,www

If more than one client is subscribed to a topic, all of them get the messages.

c2|unsubscribe,A
c2|!unsubscribed,A
c1|publish,A,vvv
c1|A,vvv

The unsubscription makes the client stop receiving messages from this topic.

connect c3

```
c3|!ready,cliconn3
c3|exit
c3|!exiting
c3|__EOF__
```

The third client connects, immediately requests an exit, gets the confirmation and gets disconnected.

```
connect c4
c4|!ready,cliconn4
close WR c4
c4|!exiting
c4|__EOF__
```

The fourth client connects and then closes its write side of the socket (that is the read side for the server). It produces the same affect as the exit command.

```
c1|shutdown
c1|*,server shutting down
c1|__EOF__
c2|*,server shutting down
c2|__EOF__
```

And the shutdown command sends the notifications to all the remaining clients and closes the connections.

16.7. ThreadedServer implementation, and the details of thread harvesting

And now I want to show the internals of the ThreadedServer methods. It shows how to store the socket file handles into the App, how the threads are harvested, and how the connections get accepted. The class is defined in `lib/Triceps/X/ThreadedServer.pm`. It's of essentially production quality but misses the detailed set of tests yet, and thus for now is placed in the X subdirectory.

The most important method in it is `startServer()`:

```perl
sub startServer # ($optName => $optValue, ...)
{
  my $myname = "Triceps::X::ThreadedServer::startServer";
  my $opts = {};
  my @myOpts = (
    app => [ undef, \&Triceps::Opt::ck_mandatory ],
    thread => [ "global", undef ],
    main => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
 "CODE") } ],
    port => [ undef, \&Triceps::Opt::ck_mandatory ],
    socketName => [ undef, undef ],
    fork => [ 1, undef ],
  );
  &Triceps::Opt::parse($myname, $opts, {
    @myOpts,
    '*' => [],
  }, @_);

  if (!defined $opts->{socketName}) {
    $opts->{socketName} = $opts->{thread} . ".listen";
  }

  my $srvsock = IO::Socket::INET->new(
    Proto => "tcp",
    LocalPort => $opts->{port},
```

```
    Listen => 10,
) or confess "$myname: socket creation failed: $!";
my $port = $srvsock->sockport() or confess "$myname: sockport failed: $!";
```

So far it's pretty standard: get the options and open the socket for listening. A slightly special thing is that it saves the list of supported options in the variable @myOpts for the future use. That future use will be to pass the list of unknown options to the user thread, so the known options in @myOpts will be removed from the list before passing it. "*" is the pass-through: it tells Opt::parse to accept any options and gets all the unknown options collected in $opts->{"*"}.

```
if ($opts->{fork} > 0)  {
  my $pid = fork();
  confess "$myname: fork failed: $!" unless defined $pid;
  if ($pid) {
    # parent
    $srvsock->close();
    return ($port, $pid);
  }
  # for the child, fall through
}
```

This handles the process forking option: if forking is requested, it executes and then the parent process returns the PID while the child process continues with the rest of the logic. By the way, your success with forking a process that has multiple running threads may vary. The resulting process usually has one running thread (continuing where the thread that called fork() was) but the synchronization primitives in the new process can be inherited in any state, so the attempts to continue the threaded processing are usually not such a good idea. It's usually best to fork first before there are more threads.

```
# make the app explicitly, to put the socket into it first
my $app = Triceps::App::make($opts->{app});
$app->storeCloseFile($opts->{socketName}, $srvsock);
Triceps::Triead::start(
  app => $opts->{app},
  thread => $opts->{thread},
  main => $opts->{main},
  socketName => $opts->{socketName},
  &Triceps::Opt::drop({ @myOpts }, \@_),
);
```

Then the App gets created. Previously I've shown starting the App with startHere() that created the App, the first thread, and did a bunch of services implicitly. Here everything will be done manually.

Triceps keeps a global list of all its Apps in the process, and after an App is created, it's placed into that list and can be found by name from any thread. The App object will exist while there are references to it, including the reference from that global list. On the other hand, it's possible to remove the App from the list while it's still running but that's a bad practice because it will break any attempts from its threads to find it by name.

So the App is made, then the file handle gets stored. The listening socket has to be stored into the App before the listener thread gets started, so that the listener thread can find it. storeCloseFile() gets the file descriptor from the socket, dups it, stores into the App, and then closes the original file handle, as explained before.

Then the listener thread is started, and as shown before, it's responsible for starting all the other threads. It gets a set of fixed options plus all the unknown options that the user had specified. It will do the same trick with passing these unknown options to the thread that runs the user code and handles the accepted connections, such as readerT() in Section 16.6: "Dynamic threads and fragments in a socket server" (p. 304) . In the meantime, the startServer() continues:

```
my $tharvest;
if ($opts->{fork} < 0) {
  @_ = (); # prevent the Perl object leaks
  $tharvest = threads->create(sub {
    # In case of errors, the Perl's join() will transmit the error
    # message through.
```

```
    Triceps::App::find($_[0])->harvester();
  }, $opts->{app}); # app has to be passed by name
} else {
  $app->harvester();
}
```

The harvester logic is started. Each App must have its harvester. `startHere()` runs the harvester implicitly, unless told otherwise, but if the App is created manually, the harvester has to be run manually. It can be run either in this thread or in another thread, as determined by the ThreadedServer option "fork".

If the option "fork" is 0, the server just runs the harvester in the current process and thread. If the option "fork" is 1, the server runs in the first thread of the child process, and if the code got to this point, it would be running now in the child process. So these two cases are handled together and for them the harvester just starts in the current thread with `$app->harvester();` and returns after the App shuts down.

If the option "fork" is -1, it requires that the harvester is to be started in another newly created thread. `$tharvest` will contain that thread's identity. A special thing about starting threads with `threads->create()` is that it's sensitive to anything in @_. If @_ contains anything, it will be leaked (though the more recent versions of Perl should have it fixed). So @_ gets cleared before starting the thread.

And the harvester gets started in the new thread slightly differently. Since in the new thread all the XS objects become `undef`s, the App object has to be found by name in the global list first.

One way or the other, the harvester is started. What does it do? It joins the App's threads as they exit. After all of them exit, it removes the App from the global list of Apps, which will allow to collect the App's memory when the last reference to it is gone, and then the harvester returns.

If any of the App's threads die rather than exit nicely, they cause the App to be aborted. The aborted App shuts down immediately and remembers the identity of the failed thread and its error message (only the first message is saved because the abort is likely to cause the other threads to die too, and there is no point in seeing these derivative messages). The harvester, in turn, collects this message from the App, and after all its cleaning-up work is done, also dies, propagating this message. Then if the harvester is running not in the first thread of the program, that message will be propagated further by Perl's `join()`.

A catch is that the errors are not reported until the harvester completes. Normally all the App's threads should exit immediately on shutdown but if they don't, the program will be stuck without any indication of what happened.

It's also possible to disable this propagation of dying by using the option "die_on_abort":

```
$app->harvester(die_on_abort => 0);
```

There is a way to get the error message that caused the abort directly from the App instead.

And finally the last part of startServer():

```
  if ($opts->{fork} > 0) {
    exit 0; # the forked child process
  }

  return ($port, $tharvest);
}
```

If this was the child process forked before, it exits at this point. Otherwise the port and the harvester's thread object are returned.

The next function of the ThreadedServer is `listen()`, the function that gets called from the listener thread and takes care of accepting the connections and spawning the per-client threads.

```
sub listen # ($optName => $optValue, ...)
{
```

```
  my $myname = "Triceps::X::ThreadedServer::listen";
  my $opts = {};
  my @myOpts = (
    owner => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
"Triceps::TrieadOwner") } ],
    socket => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
"IO::Socket") } ],
    prefix => [ undef, \&Triceps::Opt::ck_mandatory ],
    handler => [ undef, sub { &Triceps::Opt::ck_mandatory(@_); &Triceps::Opt::ck_ref(@_,
"CODE") } ],
    pass => [ undef, sub { &Triceps::Opt::ck_ref(@_, "ARRAY") } ],
  );
  &Triceps::Opt::parse($myname, $opts, {
    @myOpts,
    '*' => [],
  }, @_);
  my $owner = $opts->{owner};
  my $app = $owner->app();
  my $prefix = $opts->{prefix};
  my $sock = $opts->{socket};
```

The option parsing is similar to `startServer()`: it also saves the list of supported options in the variable `@myOpts` and accepts any unknown options with "*", to pass them through to the user thread.

```
  my $clid = 0; # client id

  while(!$owner->isRqDead()) {
    my $client = $sock->accept();
    if (!defined $client) {
      my $err = "$!"; # or the text message will be reset by isRqDead()
      if ($owner->isRqDead()) {
        last;
      } elsif($!{EAGAIN} || $!{EINTR}) { # numeric codes don't get reset
        next;
      } else {
        confess "$myname: accept failed: $err";
      }
    }
```

The accept loop starts. It runs until the thread is requested to die at shutdown. The shutdown would also revoke the listening socket, and thus if it happened in the middle of `$sock->accept()`, `$sock->accept()` would return an `undef` because it will see the revocation as an error. Note that `listen()` itself doesn't request to track the socket for revocation. The caller must do that before calling `listen()`.

So, after `accept()` the code checks for errors. The first thing to check for is again the `isRdDead()`, because it's not really an error, it's an expected condition. If the shutdown is found, the loop exits. However the error text had to be saved before `isRqDead()`, because like other Triceps calls `isRdDead()` will clear the error text. Then the check for the spurious interruptions is done, and for them the loop continues. Interestingly, the `$!{}` uses the numeric part of `$!` that is independent from its text part and doesn't get cleared by the Triceps calls. And on any other errors the thread confesses. The confession will unroll the stack, eventually will get caught by the Triceps threading code, abort the App, and propagate the error message to the harvester.

```
    $clid++;
    my $cliname = "$prefix$clid";
    $app->storeCloseFile($cliname, $client);

    Triceps::Triead::start(
      app => $app->getName(),
      thread => $cliname,
      fragment => $cliname,
      main => $opts->{handler},
```

```
      socketName => $cliname,
      &Triceps::Opt::drop({ @myOpts }, \@_),
    );

    # Doesn't wait for the new thread(s) to become ready.
  }
}
```

If a proper connection has been received, the socket gets stored into the App with a unique name, for later load by the per-client thread. And then the per-client thread gets started.

`Opt::drop()` passes through all the original options except for the ones handled by this thread. In retrospect, this is not the best solution for this case. It would be better to just use `@{$opts->{"*"}}` instead. `Opt::drop()` is convenient when not all the explicitly recognized options but only some of them have to be dropped.

After starting the thread, the loop doesn't call `readyReady()` but goes for the next iteration. This is basically because it doesn't care about the started thread and doesn't ever send anything to it. And waiting for the threads to start will make the loop slower, possibly overflowing the socket's listening queue and dropping the incoming connections if they arrive very fast.

And the last part of the ThreadedServer is the function `printOrShut()`:

```
sub printOrShut # ($app, $fragment, $sock, @text)
{
  my $app = shift;
  my $fragment = shift;
  my $sock = shift;

  undef $!;
  print $sock @_;
  $sock->flush();

  if ($!) { # can't write, so shutdown
    Triceps::App::shutdownFragment($app, $fragment);
  }
}
```

Nothing too complicated. Prints the text into the socket, flushes it and checks for errors. On errors shuts down the fragment. In this case there is no need for draining. After all, the socket leading to the client is dead and there is no way to send anything more through it, so there is no point in worrying about any unsent data. Just shut down as fast as it can, before the threads have generated more data that can't be sent any more. Any data queued in the nexuses for the shut down threads will be discarded.

16.8. ThreadedClient, a Triceps Expect

In case if you're not familiar with it, `expect` is a program that allows to connect to the interactive programs and pretend being an interactive user. Obviously, the terminal programs, not the GUI ones. It has originally been done as an extension for Tcl, and later ported as a library for Perl and other languages.

The class Triceps::X::ThreadedClient implements a variety of `expect` in the Triceps framework. I'm using it for the unit tests of the Triceps servers but it can have other uses as well. Why not just use `expect`? One reason, I don't like bringing in extra dependencies, especially just for tests, second, it was an interesting exercise, and third, I didn't realize that I was writing a simplified variety of expect until I had it mostly completed. The biggest simplification compared to the real `expect` is that ThreadedClient works with the complete lines.

It gets used in the unit tests like this: first the server gets started in a background process or thread, and then the server's port number is used to create the clients. The ThreadedClient gets embedded into a Triceps App, so you can start other things in the same App. Well, the names of the ThreadedClient threads are hardcoded at the moment, so you can start only one copy of it per App, and there could be conflicts if you start your other threads and use the same names as in ThreadedClient.

But first you need to start an App. I'll show it done in yet another way this time:

```
Triceps::App::build "client", sub {
  my $appname = $Triceps::App::name;
  my $owner = $Triceps::App::global;

  # give the port in startClient
  my $client = Triceps::X::ThreadedClient->new(
    owner => $owner,
    totalTimeout => 5,
    debug => 0,
  );

  $owner->readyReady();

  $client->startClient("c1", $port);
  $client->expect("c1", '!ready');

  # this repetition tests the expecting to the first match
  $client->send("c1", "publish,*,zzzzzz\n");
  $client->send("c1", "publish,*,zzzzzz\n");
  $client->expect("c1", '\*,zzzzzz');
  $client->expect("c1", '\*,zzzzzz');

  $client->startClient("c2", $port);
  $client->expect("c2", '!ready,cliconn2');

  $client->send("c1", "kill,cliconn2\n");
  $client->expect("c2", '__EOF__');

  $client->send("c1", "shutdown\n");
  $client->expect("c1", '__EOF__');
};
```

`Triceps::App::build()` is kind of like `Triceps::Triead::startHere()` but saves the trouble of parsing the options. The app name is its first argument and the code for the main routine of the first thread is the second argument. That first Triead will run in the current Perl thread. After that the name of the app is placed into the global variable $Triceps::App::name, the App object into $Triceps::App::app, and the TrieadOwner into $Triceps::App::global. The name of the first Triead is hardcoded as "global". After the main function exits, `Triceps::App::build()` runs the harvester, very similar to `startHere()`.

`Triceps::X::ThreadedClient->new()` is used to start an instance of a client. A single client instance supports multiple connections, to the same or different servers. The option "owner" gives it the current TrieadOwner as a starting point but ThreadedClient will create its own threads starting from this point.

The port number for the connection can be specified as either the option "port" or as the second argument of `startClient()`. You can use both, and then the option will provide the default value while `startClient()` can stil override it.

The option "debug" is optional, with the default value of 0. In the non-debug mode ThreadedClient collects the trace of the run but otherwise runs silently. Setting the "debug" option to 1 makes it also print the trace as it gets collected, so if something goes not the way you expected, you can see what it is. Setting the debug to 1 also adds the printouts from the socket-facing threads, so you can also see what goes in and out the client socket.

The option "totalTimeout" can be used to set the time limit in seconds (as a floating-point number, so you can use values like 0.1 for a timeout of 0.1 second) for the whole exchange. It's very convenient for the unit tests, limiting the time for which the test could get stuck if something goes wrong. If the timeout gets exceeded, all the following calls of `expect()` will return immediately, setting the error message in $@ and appending it to the trace.

There are three ways to specify the timeout. Two of them are in the `new()`:

```
my $client = Triceps::X::ThreadedClient->new(
  owner => $owner,
  totalTimeout => $timeout,
);

my $client = Triceps::X::ThreadedClient->new(
  owner => $owner,
  timeout => $timeout,
);
```

The option "totalTimeout" gives a timeout for the whole run to complete. Once that timeout is reached, all future `expect()`s just fail immediately. The option "timeout" gives the default timeout for each `expect()`. It's possible to use both, and each call of `expect()` will be limited by the shorter time limit of the two ("totalTimeout" starts counting since the call of `new()`, **not** from `startClient()`, while "timeout" starts counting since the call of `expect()`).

The third way is to use an extra argument in `expect()`:

```
$client->expect("c1", "expected text", $timeout);
```

This timeout completely overrides whatever was set in `new()`. The value of 0 disables the timeout for this call, and even 0 still overrides the timeout from `new()`, so it can be used for the one-off calls without the timeout.

After the ThreadedClient object is created, the client connections get started with `startClient()`. Its first argument is a symbolic name of the client that will be used in the further calls and also in the trace. The second optional argument is the port number, if you want to use a different port than specified in the `new()`.

After that the data gets sent into the client socket with `send()`, and the returned lines get parsed with `expect()`. The usual procedure is that you send something, then expect some response to it.

The second argument of `expect()` is actually a regexp placed inside a string. So to expect a literal special character like "*", prefix it with a backslash and put the string into single quotes, like:

```
$client->expect("c1", '\*,zzzzzz');
```

`expect()` keeps reading lines until it finds one matching the regexp. Then all the lines including that one are added to the trace and the call returns. The sent lines are also included into the trace, with the prefix "> ".

There are a couple of special lines generated by the connection status itself rather than coming from the socket. When the socket gets connected, it generates the line "connected *name*" in the trace (but not as an expectable input). When the connection gets dropped by the server, this generates an expectable line "__EOF__". And each line in the trace gets prefixed with the client name and "|". The example of the chat server run in the previous section was created by the ThreadedClient.

The socket disconnect ("__EOF__") has another special property: if it's encountered by an `expect()` call that doesn't expect it, that call and the following ones will set the error message in $@ and append it to the trace, then immediately return. This handles the situation with the unexpectedly dropped connection quickly, without resorting to the timeout.

The recorded trace can be extracted as `$client->getTrace()`. So for unit tests you can check the trace match as

```
ok($client->getTrace(), $expected);
```

It's possible to get only the error messages with `$client->getErrorTrace()`.

One more method not shown above allows to close the client socket (in socket terms, shut it down):

```
$client->sendClose("c4", "WR");
```

The first argument is the client name. The second argument determines, which side of the socket gets closed: "WR" for the writing side, "RD" for the reading side, and "RDWR" for both. It's the same names as for the system call `shutdown()`.

Internally the ThreadedClient is very much like the chat server, except for the support of the timeouts. The way the timeouts work is described below in Section 16.9: "Thread main loop and timeouts in the guts of ThreadedClient" (p. 321.) But

there is not a whole lot of point in going over the rest of implementation in detail. Feel free to read it on your own, in `lib/Triceps/X/ThreadedClient.pm`.

16.9. Thread main loop and timeouts in the guts of ThreadedClient

First of all, let me re-iterate: if you look for repeatability of computation, you should really use an external time source synchronized with your data. But sometimes you really need to work with the real time, such as when handling the input from a socket and/or time-limiting the run of the tests. The repeatability doesn't matter much for these uses, since it's really for handling of the cases that Should Never Happen.

The ThreadedClient uses this kind of timeouts, so I'll use its implementation as an example. Let's start by stepping back and looking at the structure of ThreadedClient. It runs in multiple threads.

The collector thread is the heart of ThreadedClient. When a ThreadedClient object is created, the collector thread starts. It will be receiving data from the connections and matching it up with the expectations.

Whenever a client connection is created, two threads go with it, the reader and writer. The reader passes the data directly to the collector, which then buffers all the received data until `expect()` asks for it.

Finally, there is the thread where the user's control code runs. When a ThreadedClient gets created in the user thread, the option "owner" provides the TrieadOwner of that user thread to the ThreadedClient object.

`expect()` works by sending a request to the collector thread and waiting for the response from it. If there is a time limit and the response is not received within it, the wait gets interrupted, and a cancellation request gets sent to the collector.

`new()` creates the labels for processing the replies from the collector:

```
$self->{faReply}->getLabel("msg")->makeChained("lbReplyExpect", undef, sub {
  my ($cmd, $client, $arg) = $_[1]->getRow()->toArray();
  if ($cmd eq "expect") {
    my $ptext = $arg;
    $ptext =~ s/^/$client|/gm;
    $self->{trace} .= $ptext;
    if ($self->{debug}) {
      print $ptext;
    }
    $self->{expectDone} = 1;
  } elsif ($cmd eq "error") {
    # save the error in trace, so that it will be easily printed.
    my $ptext = $arg . "\n";
    $ptext =~ s/^/$client|/gm;
    $self->{trace} .= $ptext;
    $self->{errorTrace} .= $ptext;
    if ($self->{debug}) {
      print $ptext;
    }

    $self->{error} = $arg;
    $self->{expectDone} = 1;
  }
```

But of course these labels will not run until the thread reads the data from its input facets. Most threads read the data from the facets continuously with `TrieadOwner::mainLoop()`. Of course, the user thread that owns ThreadedClient can't do that, or it would not be able to run the user code. But anyway, let's look at the internals of the `mainLoop()` for the reference first. It's implemented in C++ literally as follows:

```
void TrieadOwner::mainLoop()
```

```
{
  while (nextXtray())
    { }
}
```

`nextXtray()` represents one step of the main loop: it reads one incoming tray from one of the input facets and processes it. "Xtray" is a special form of the trays used to pass the data across the nexus. If no tray is available, it waits. If more that one facet has trays available, the facets of the reverse nexuses get the higher priority, and then within the same priority it round-robins. It returns `true` until the thread is requested dead, and then it returns `false`. `nextXtray()` is available in the Perl API as well.

Now we're ready to look at `expect()`. The core part of `expect()`, after it computes the time limit from the three sources, is this:

```
$self->{error} = undef;
$self->{expectDone} = 0;

if ($self->{debug} > 1) {
  print "expect: $pattern\n"
}

$owner->unit()->makeHashCall($self->{faCtl}->getLabel("msg"), "OP_INSERT",
  cmd => "expect",
  client => $client,
  arg => $pattern,
);
$owner->flushWriters();

if ($limit > 0.) {
  while(!$self->{expectDone} && $owner->nextXtrayTimeLimit($limit)) { }
  # on timeout reset the expect and have that confirmed
  if (!$self->{expectDone}) {
    $owner->unit()->makeHashCall($self->{faCtl}->getLabel("msg"), "OP_INSERT",
      cmd => "cancel",
      client => $client,
    );
    $owner->flushWriters();
    # wait for confirmation
    while(!$self->{expectDone} && $owner->nextXtray()) { }
  }
} else {
  while(!$self->{expectDone} && $owner->nextXtray()) { }
}


if ($app->isAborted()) {
  confess "$myname: app is aborted";
}
$@ = $self->{error};
```

If there is no time limit, it keeps reading the data from the facets with `$owner->nextXtray()` until it gets the reply from the collector thread. If there is a time limit, it uses a special form of `nextXtray()` with the time limit. There are multiple of these special forms:

`$res = $owner->nextXtrayNoWait();`

Returns immediately if there is no data available at the moment.

`$res = $owner->nextXtrayTimeLimit($deadline);`

Returns if no data becomes available before the absolute deadline.

```
$res = $owner->nextXtrayTimeout($timeout);
```

Returns if no data becomes available before the timeout, starting at the time of the call. Just to reiterate, the difference is that the `nextXtrayTimeLimit()` receives the absolute time since the epoch while `nextXtrayTimeout()` receives the length of the timeout starting from the time of the call, both as seconds in floating-point.

All the forms that may return early on no data return `false` if they have to do so. If you need to differentiate between these functions returning `false` due to the lack of data and due to the thread being requested to shut down, you can do it with `$owner->isRqDead()`.

The absolute deadline is generally a more reliable approach than the relative timeout. It's not susceptible to drift over time caused by the slight delays between the timeout being calculated and the start of the time-limited call. To compute the deadline as an offset from the current time, you need to get the current time. The function `Triceps::now()` returns the current time in seconds since epoch as a floating-point value, including the fractions of the seconds (which is the general time format of the Triceps Perl API). Then you add the timeout to it and get the deadline value:

```
$limit = Triceps::now() + $timeout;
```

`Triceps::now()` is also convenient for finding out the performance of the Triceps code, and in general of the Perl code.

`expect()` uses the form with the absolute deadline. If the collector thread finds a match, it will send a rowop back to the expect thread, where it will get processed in `nextXtrayTimeLimit()`, calling a label that sets the flag `$self->{expectDone}`, and then `nextXtrayTimeLimit()` will return 1, and the loop will find the flag and exit.

If the collector thread doesn't find a match, `nextXtrayTimeLimit()` will return 0, and the loop will again exit. But then it will fill find the "done" flag not set, so it knows that the timeout has expired and it has to tell the collector thread that the call is being cancelled. So it sends another rowop for the cancel, and then waits for the confirmation with another `nextXtray()`, this time with no limit on it since the confirmation must arrive back quickly in any case.

It's the confirmation rowop processing that sets `$self->{error}`. But there is always a possibility that the match will arrive just after the timeout has expired but just before the cancellation. It's one of these things that you have to deal with when multiple threads exchange messages. What then? Then the normal confirmation will arrive back to the expecting thread. And when the cancel message will arrive to the collector thread, it will find that this client doesn't have an outstanding expect requests any more, and will just ignore the cancel. Thus, the second `nextXtray()` will receive either a confirmation of the cancel and set the error message, or it will receive the last-moment success message. Either way it will fall through and return (setting $@ if the cancel confirmation came back).

By the way, if the collector thread finds the socket closed, it will immediately return an error rowop, very similar to the confirmation of the cancel. Which will set both `$self->{expectDone}` and `$self->{error}` in the expect thread.

16.10. The threaded dreaded diamond and data reordering

The pipelined examples shown before had very conveniently preserved the order of the data while spreading the computational load among multiple threads. But it forced the computation to pass sequentially through every thread, increasing the latency and adding overhead. The other frequently used option is to farm out the work to a number of parallel threads and then collect the results back from them. This topology is colloquially known as "diamond" or "fork-join" (having nothing to do with the SQL joins, just that the arrows first fork from one box to multiple and then join back to one box).

The single-threaded use of this topology has already been discussed in Section 14.1: "The dreaded diamond" (p. 231) , and its diagram can be found there in Figure 14.1: "The diamond topology." (p. 231) .

There are multiple ways to decide, which thread gets which unit of data to process. One possibility that provides the natural load balancing is to keep a common queue of work items, and have the worker threads (B and C in Figure 14.1 (p. 231)) read the next work item when they become free after processing the last item. But this way has an important limitation: there is no way to tell in advance, which work item will go to which thread, so there is no way for the worker threads to

keep any state. All the state would have to be encapsulated into the work item (a work item would be a tray in the Triceps terms). And at the moment Triceps provides no way to maintain such a shared queue.

The other way is to partition the data between the worker threads based on the primary key. Usually it's done by using a hash of the key, which would distribute the data randomly and hopefully evenly. Then a worker thread can keep the state for its subset of keys, forming a partition (also known as "shard") and process the sequential updates for this partition. The way to send a rowop only to the thread B or only to the thread C would be by having a separate designated nexus for each worker thread, and the thread A making the decision based on the hash of the primary key in the data. The processed data from all the worker threads can then be sent to a single nexus feeding the thread D.

But either way the data will arrive at D in an unpredictable order. The balance of load between the worker threads is not perfect, and there is no way to predict, how long will each tray take to process. A tray following one path may overtake a tray following another path.

If the order of the data is important, D must collate the data back into the original order before processing it further. Obviously, for that it must know what the original order was, so A must tag the data with the sequential identifiers. Since the data rows themselves may get multiplied, the tagging is best done at the tray level (what happens if the trays split in the worker threads is a separate story, for now the solution is simply "keep the tray together").

When the data goes through a nexus, Triceps always keeps a tray as an indivisible bundle. It always gets read from the reading facet together, without being interspersed with the data from the other trays. As the data is sent into a writer facet, it's collected in a tray, and sent on only when the facet gets flushed, with the `TrieadOwner::flushWriters()` or `Facet::flushWriter()`.

There also are two special labels defined on every nexus, that tell the tray boundaries to the reading thread:

- "_BEGIN_" is called at the start of the tray.

- "_END_" is called at the end of the tray.

These labels can be defined explicitly, or otherwise they become defined implicitly anyway. If they get defined implicitly, they get the empty row type (one with no fields). If you define them explicitly, you can use any row type you please, and this is a good place for the tray sequence id.

And in a writer facet you can send data to these labels. When you want to put a sequence id on a tray, you define the "_BEGIN_" label with that sequence id field, and then call the "_BEGIN_" label with the appropriate id values. Even if you don't call the "_BEGIN_" and "_END_" labels, they do get called (not quite called but placed on the tray) automatically anyway, with the opcode of `OP_INSERT` and all the fields NULL. The direct calling of these labels will also cause the facet to be flushed: "_BEGIN_" flushes any data previously collected on the tray and then adds itself; "_END_" adds itself and then flushes the tray.

The exact rules of how the "_BEGIN_" and "_END_" get called are actually somewhat complicated, having to deal with the optimizations for the case when nobody is interested in them, but they do what makes sense intuitively.

The case when these labels get a call with `OP_INSERT` and no data, gets optimized out by not placing it into the actual Xtray, even if it was called explicitly. Then in the reader facet these implicit rowops are re-created but only if there is a chaining for their labels from the facet's FnReturn or if they are defined in the currently pushed FnBinding. So if you do a trace on the reading thread's unit, you will see these implicit rowops to be called only if they are actually used.

Before going to the example itself, let's talk more about the general issues of the data partitioning.

If the data processing is stateless, it's easy: just partition by the primary key, and each thread can happily work on its own. If the data depends on the previous data with the same primary key, the partitioning is still easy: each thread keeps the state for its part of the keys and updates it with the new data.

But what if you need to join two tables with independent primary keys, where the matches of the keys between them are fairly arbitrary? Then you can partition by one table but you have to give a copy of the second table to each thread.

Typically, if one table is larger than the other, you would partition by the key of the big table, and copy the small table everywhere. Since the data rows are referenced in Triceps, the actual data of the smaller table won't be copied, it would be just referenced from multiple threads, but each copy will still have the overhead of its own index.

With some luck, and having enough CPUs, you might be able to save a little overhead by doing a matrix: if you have one table partitioned into the parts A, B and C, and the other table into parts 1, 2 and 3, you can then do a matrix-combination into 9 threads processing the combinations A1, A2, A3, B1, B2, B3, C1, C2, C3. If both tables are of about the same size, this would create a total of 18 indexes, each keeping 1/3 of one original table, so the total size of indexes will be 6 times the size of one original table (or 3 times the combined sizes of both tables). On the other hand, if you were to copy the first table to each thread and split the second table into 9 parts, creating the same 9 threads, the total size of indexes will be 9 times the first table and 1 times the second table, resulting in the 10 times the size of one original table (or 5 times the combined sizes of both tables). The 3x3 matrix is more optimal, but the catch is that the results from it will really have to be restored to the correct order afterwards.

The reason is that when a row in the first table changes, it might change its join key, matching the row from a different partition of the second table. Which means that the DELETE of the old value and the INSERT of the new value will be processed in the different threads. For example, it might move the processing from the thread A1 to the thread A2. So the thread A1 would generate a DELETE for the join result, and the thread A2 would generate a following INSERT. With two separate threads, the resulting order will be unpredictable, and the INSERT coming before the DELETE would be bad. The post-reordering is required to ensure the correct order.

By contrast, if you just partition the first table and copy the second one everywhere, you get 9 threads A, B, C, D, E, F, G, H, I, and the change in a row will still keep it in the same thread, so the updates will come out of that thread strictly sequentially. If you don't care about the order of changes between different primary keys, you can get away without the post-reordering. Of course, if a key field might change and you care about it being processed in order, you'd still need the post-reordering.

The example I'm going to show is a somewhat strange mix. It's an adaptation of the Forex arbitration example from the Section 12.3: "The lookup join, done manually" (p. 180) . As you can see from the name of that section, it's doing a self-join, and doing it twice: kind of like going through the same table three times.

The partitioning in this example works as follows:

- All the data is sent to all the threads. All the threads keep a full copy of the table and update it according to the input.

- But then they compute the join only if the first currency name in the update falls into the thread's partition.

- The partitioning is done by the first letter of the symbol, with interleaving: the symbols starting with A are handled by the thread 0, with B by thread 1, and so on until the threads end, and then continuing again with the thread 0. A production-ready implementation would use a hash function instead. But the interleaving approach makes much easier to predict, which symbol goes to which thread for the demonstration.

- Naturally, all this means that the loop of 3 currencies might get created by a change in one pair and then very quickly disappear by a change to another pair.of currencies. So the post-reordering of the result is important to keep the things consistent.

I've also added a tweak allowing to artificially slow down the thread 0, making the incorrect order show up reliably, and really exercise the reordering code. For example, suppose the following input was sent quickly:

```
OP_INSERT,AAA,BBB,1.30
OP_INSERT,BBB,AAA,0.74
OP_INSERT,AAA,CCC,1.98
OP_INSERT,CCC,AAA,0.49
OP_INSERT,BBB,CCC,1.28
OP_INSERT,CCC,BBB,0.78
OP_DELETE,BBB,AAA,0.74
OP_INSERT,BBB,AAA,0.64
```

With two threads, and thread 0 working slowly, it would produce the raw result:

```
BEGIN OP_INSERT seq="2" triead="1"
BEGIN OP_INSERT seq="5" triead="1"
BEGIN OP_INSERT seq="7" triead="1"
result OP_DELETE ccy1="AAA" ccy2="CCC" ccy3="BBB" rate1="1.98"
    rate2="0.78" rate3="0.74" looprate="1.142856"
BEGIN OP_INSERT seq="8" triead="1"
BEGIN OP_INSERT seq="1" triead="0"
BEGIN OP_INSERT seq="3" triead="0"
BEGIN OP_INSERT seq="4" triead="0"
BEGIN OP_INSERT seq="6" triead="0"
result OP_INSERT ccy1="AAA" ccy2="CCC" ccy3="BBB" rate1="1.98"
    rate2="0.78" rate3="0.74" looprate="1.142856"
```

Here the BEGIN lines are generated by the code and show the sequence number of the input row and the id of the thread that did the join. They represent the "_BEGIN_" rowops of the trays that contain the transactions, and all the following result lines until the next BEGIN belong in the same tray. The result lines show the arbitration opportunities produced by the join. Obviously, not every update produces a new opportunity, most of them don't. But the INSERT and DELETE in the result come in the wrong order: the update 7 had overtaken the update 6.

The post-reordering comes to the resque and restores the order:

```
BEGIN OP_INSERT seq="1" triead="0"
BEGIN OP_INSERT seq="2" triead="1"
BEGIN OP_INSERT seq="3" triead="0"
BEGIN OP_INSERT seq="4" triead="0"
BEGIN OP_INSERT seq="5" triead="1"
BEGIN OP_INSERT seq="6" triead="0"
result OP_INSERT ccy1="AAA" ccy2="CCC" ccy3="BBB" rate1="1.98"
    rate2="0.78" rate3="0.74" looprate="1.142856"
BEGIN OP_INSERT seq="7" triead="1"
result OP_DELETE ccy1="AAA" ccy2="CCC" ccy3="BBB" rate1="1.98"
    rate2="0.78" rate3="0.74" looprate="1.142856"
BEGIN OP_INSERT seq="8" triead="1"
```

As you can see, now the sequence numbers go in the sequential order.

And finally the code of the example. I've written it with the simple approach of "read all stdin, process, print all to stdout". It's easier this way, and anyway to demonstrate the rowop reordering, all the input has to be sent quickly in one chunk. The example is found in `t/xForkJoinMt.t`. The application starts as:

```
Triceps::Triead::startHere(
  app => "ForkJoin",
  thread => "main",
  main => \&mainT,
  workers => 2,
  delay => 0.02,
);
```

The main thread is:

```
sub mainT # (@opts)
{
  my $opts = {};
  &Triceps::Opt::parse("mainT", $opts, {@Triceps::Triead::opts,
    workers => [ 1, undef ], # number of worker threads
    delay => [ 0, undef ], # artificial delay for the 0th thread
  }, @_);
  undef @_; # avoids a leak in threads module
  my $owner = $opts->{owner};
  my $app = $owner->app();
  my $unit = $owner->unit();
```

So far the pretty standard boilerplate with the argument parsing.

```
my $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
);

# the resulting trade recommendations
my $rtResult = Triceps::RowType->new(
  triead => "int32", # id of the thread that produced it
  ccy1 => "string", # currency code
  ccy2 => "string", # currency code
  ccy3 => "string", # currency code
  rate1 => "float64",
  rate2 => "float64",
  rate3 => "float64",
  looprate => "float64",
);
```

The row types originate from the self-join example code, unchanged.

```
# each tray gets sequentially numbered and framed
my $rtFrame = Triceps::RowType->new(
  seq => "int64", # sequence number
  triead => "int32", # id of the thread that produced it (optional)
);

# the plain-text output of the result
my $rtPrint = Triceps::RowType->new(
  text => "string",
);
```

Thes definitions of the service row types. The frame row type is used to send the information about the sequence number in the "_BEGIN_" label. The print row type is used to send the text for printing back to the main thread.

```
# the input data
my $faIn = $owner->makeNexus(
  name => "input",
  labels => [
    rate => $rtRate,
    _BEGIN_ => $rtFrame,
  ],
  import => "none",
);

# the raw result collected from the workers
my $faRes = $owner->makeNexus(
  name => "result",
  labels => [
    result => $rtResult,
    _BEGIN_ => $rtFrame,
  ],
  import => "none",
);

my $faPrint = $owner->makeNexus(
  name => "print",
  labels => [
    raw => $rtPrint, # in raw order as received by collator
    cooked => $rtPrint, # after collation
  ],
```

```
    import => "reader",
  );
```

The processing will go in essentially a pipeline: "read the input -> process in the worker threads -> collate -> print in the main thread". A nexus is defined for each connection between the stages of the pipeline. The worker thread stage spreads into multiple parallel threads for parititoned data, then joining the data paths back together in the collator.

```
  Triceps::Triead::start(
    app => $app->getName(),
    thread => "reader",
    main => \&readerT,
    to => $owner->getName() . "/input",
  );

  for (my $i = 0; $i < $opts->{workers}; $i++) {
    Triceps::Triead::start(
      app => $app->getName(),
      thread => "worker$i",
      main => \&workerT,
      from => $owner->getName() . "/input",
      to => $owner->getName() . "/result",
      delay => ($i == 0? $opts->{delay} : 0),
      workers => $opts->{workers},
      identity => $i,
    );
  }

  Triceps::Triead::start(
    app => $app->getName(),
    thread => "collator",
    main => \&collatorT,
    from => $owner->getName() . "/result",
    to => $owner->getName() . "/print",
  );

  my @rawp; # the print in original order
  my @cookedp; # the print in collated order

  $faPrint->getLabel("raw")->makeChained("lbRawP", undef, sub {
    push @rawp, $_[1]->getRow()->get("text");
  });
  $faPrint->getLabel("cooked")->makeChained("lbCookedP", undef, sub {
    push @cookedp, $_[1]->getRow()->get("text");
  });

  $owner->readyReady();

  $owner->mainLoop();

  print("--- raw ---\n", join("\n", @rawp), "\n");
  print("--- cooked ---\n", join("\n", @cookedp), "\n");
}
```

All the threads get started and instructed to connect to the appropriate nexuses. The collator will send the data for printing twice: first time in the order it was received ("raw"), second time in the order after collation ("cooked").

```
sub readerT # (@opts)
{
  my $opts = {};
  &Triceps::Opt::parse("readerT", $opts, {@Triceps::Triead::opts,
    to => [ undef, \&Triceps::Opt::ck_mandatory ], # dest nexus
  }, @_);
```

```
    undef @_; # avoids a leak in threads module
    my $owner = $opts->{owner};
    my $unit = $owner->unit();

    my $faIn = $owner->importNexus(
      from => $opts->{to},
      import => "writer",
    );

    my $lbRate = $faIn->getLabel("rate");
    my $lbBegin = $faIn->getLabel("_BEGIN_");
    # _END_ is always defined, even if not defined explicitly
    my $lbEnd = $faIn->getLabel("_END_");
```

This demonstrates that the labels "_BEGIN_" and "_END_" always get defined in each nexus, even if they are not defined explicitly. Here "_BEGIN_" was defined explicitly but "_END_" was not, and nevertheless it can be found and used.

```
    my $seq = 0; # the sequence

    $owner->readyReady();

    while(<STDIN>) {
      chomp;

      ++$seq; # starts with 1
      $unit->makeHashCall($lbBegin, "OP_INSERT", seq => $seq);
      my @data = split(/,/); # starts with a string opcode
      $unit->makeArrayCall($lbRate, @data);
      # calling _END_ is an equivalent of flushWriter()
      $unit->makeHashCall($lbEnd, "OP_INSERT");
    }

    {
      # drain the pipeline before shutting down
      my $ad = Triceps::AutoDrain::makeShared($owner);
      $owner->app()->shutdown();
    }
}
```

Each input row is sent through in a separate transaction, or in another word, a separate tray. The "_BEGIN_" label carries the sequence number of the tray. The trays can as well be sent on with `TrieadOwner::flushWriters()` or `Facet::flushWriter()`, but I wanted to show that you can also flush it by calling the "_END_" label.

```
sub workerT # (@opts)
{
  my $opts = {};
  &Triceps::Opt::parse("workerT", $opts, {@Triceps::Triead::opts,
    from => [ undef, \&Triceps::Opt::ck_mandatory ], # src nexus
    to => [ undef, \&Triceps::Opt::ck_mandatory ], # dest nexus
    delay => [ 0, undef ], # processing delay
    workers => [ undef, \&Triceps::Opt::ck_mandatory ], # how many workers
    identity => [ undef, \&Triceps::Opt::ck_mandatory ], # which one is us
  }, @_);
  undef @_; # avoids a leak in threads module
  my $owner = $opts->{owner};
  my $unit = $owner->unit();
  my $delay = $opts->{delay};
  my $workers = $opts->{workers};
  my $identity = $opts->{identity};

  my $faIn = $owner->importNexus(
    from => $opts->{from},
```

```
    import => "reader",
  );

  my $faRes = $owner->importNexus(
    from => $opts->{to},
    import => "writer",
  );

  my $lbInRate = $faIn->getLabel("rate");
  my $lbResult = $faRes->getLabel("result");
  my $lbResBegin = $faRes->getLabel("_BEGIN_");
  my $lbResEnd = $faRes->getLabel("_END_");
```

The worker thread starts with the pretty usual boilerplate.

```
  my $seq; # sequence from the frame labels
  my $compute; # the computation is to be done by this label
  $faIn->getLabel("_BEGIN_")->makeChained("lbInBegin", undef, sub {
    $seq = $_[1]->getRow()->get("seq");
  });
```

The processing of each transaction starts by remembering its sequence number from the "_BEGIN_" label. It doesn't send a "_BEGIN_" to the output yet because it doesn't know yet if it will be producing the output. All the threads get the same input, to be able to update their copies of the table, but then only one thread produces the output. And the thread doesn't know whether it will be the one producing the output until it knows the primary key of the data. So it can start sending the output only after it had seen the data. This whole scheme works for this example because there is exactly one data row per each transaction. A more general approach might be to have the reader thread decide up front, which worker will produce the result and put this information (as either a copy of the primary key or the computed thread id) into the "_BEGIN_" rowop.

```
  ...

  # the table gets updated for every incoming rate
  $lbInRate->makeChained("lbIn", undef, sub {
    my $ccy1 = $_[1]->getRow()->get("ccy1");
    # decide, whether this thread is to perform the join
    $compute = ((ord(substr($ccy1, 0, 1)) - ord('A')) % $workers == $identity);

    # this relies on every Xtray containing only one rowop,
    # otherwise one Xtray will be split into multiple
    if ($compute) {
      $unit->makeHashCall($lbResBegin, "OP_INSERT", seq => $seq, triead => $identity);
      select(undef, undef, undef, $delay) if ($delay);
    }

    # even with $compute is set, this might produce some output or not,
    # but the frame still goes out every time $compute is set, because
    # _BEGIN_ forces it
    $unit->call($lbRateInput->adopt($_[1]));
  });
```

This code makes the decision of whether this join is to be computed for this thread. The decisiton is remembered in the flag $compute, and used to generate the "_BEGIN_" rowop for the output. Then the table gets updated in any case ($lbRateInput is the table's input label). I've skipped over the table creation code, it's unchanged from the original self-join example.

```
  $tRate->getOutputLabel()->makeChained("lbCompute", undef, sub {
    return if (!$compute); # not this thread's problem

    ...
      if ($looprate > 1) {
        $unit->call($result);
```

```
    }
   ...
  });
  ##################################################

  $owner->readyReady();

  $owner->mainLoop();
}
```

Here again I've skipped over the way the result is computed because it's lengthy and unchanged from the original example. The important part is that if the $compute flag is not set, the whole self-joining computation is not performed.

The "_END_" label is not touched, the flushing of transactions is taken care of by the mainLoop(). Note that the "_BE-GIN_" label is always sent if the data is designated to this thread, even if no output as such is produced. This is done because the collator needs to get an uninterrupted sequence of transactions. Otherwise it would not be able to say if some transaction had been dropped or only delayed.

```
sub collatorT # (@opts)
{
  my $opts = {};
  &Triceps::Opt::parse("collatorT", $opts, {@Triceps::Triead::opts,
    from => [ undef, \&Triceps::Opt::ck_mandatory ], # src nexus
    to => [ undef, \&Triceps::Opt::ck_mandatory ], # dest nexus
  }, @_);
  undef @_; # avoids a leak in threads module
  my $owner = $opts->{owner};
  my $unit = $owner->unit();

  my $faRes = $owner->importNexus(
    from => $opts->{from},
    import => "reader",
  );

  my $faPrint = $owner->importNexus(
    from => $opts->{to},
    import => "writer",
  );

  my $lbResult = $faRes->getLabel("result");
  my $lbResBegin = $faRes->getLabel("_BEGIN_");
  my $lbResEnd = $faRes->getLabel("_END_");

  my $lbPrintRaw = $faPrint->getLabel("raw");
  my $lbPrintCooked = $faPrint->getLabel("cooked");

  my $seq = 1; # next expected sequence
  my @trays; # trays held for reordering: $trays[0] is the slot for sequence $seq
    # (only of course that slot will be always empty but the following ones may
    # contain the trays that arrived out of order)
  my $curseq; # the sequence of the current arriving tray
```

The collator thread starts very much as usual. It has its expectation of the next tray in order, which gets set correctly. The trays that arrive out of order will be buffered in the array @trays. Well, more exactly, for simplicity, all the trays get buffered there and then sent on if their turn has come. But it's possible to make an optimized version that would let the data flow through immediately if it's arriving in order.

```
  # The processing of data after it has been "cooked", i.e. reordered.
  my $bindRes = Triceps::FnBinding->new(
    name => "bindRes",
    on => $faRes->getFnReturn(),
```

```
    unit => $unit,
    withTray => 1,
    labels => [
      "_BEGIN_" => sub {
        $unit->makeHashCall($lbPrintCooked, "OP_INSERT", text => $_[1]->printP("BEGIN"));
      },
      "result" => sub {
        $unit->makeHashCall($lbPrintCooked, "OP_INSERT", text => $_[1]->printP("result"));
      }
    ],
  );
  $faRes->getFnReturn()->push($bindRes); # will stay permanently
```

The data gets collected into trays through a binding that gets permanently pushed onto the facet's FnReturn. Then when the tray's turn comes, it will be simply called and will produce the print calls for the cooked data order.

```
  # manipulation of the reordering,
  # and along the way reporting of the raw sequence
  $lbResBegin->makeChained("lbBegin", undef, sub {
    $unit->makeHashCall($lbPrintRaw, "OP_INSERT", text => $_[1]->printP("BEGIN"));
    $curseq = $_[1]->getRow()->get("seq");
  });
  $lbResult->makeChained("lbResult", undef, sub {
    $unit->makeHashCall($lbPrintRaw, "OP_INSERT", text => $_[1]->printP("result"));
  });
  $lbResEnd->makeChained("lbEnd", undef, sub {
    my $tray = $bindRes->swapTray();
    if ($curseq == $seq) {
      $unit->call($tray);
      shift @trays;
      $seq++;
      while ($#trays >= 0 && defined($trays[0])) {
        # flush the trays that arrived misordered
        $unit->call(shift @trays);
        $seq++;
      }
    } elsif ($curseq > $seq) {
      $trays[$curseq-$seq] = $tray; # remember for the future
    } else {
      # should never happen but just in case
      $unit->call($tray);
    }
  });

  $owner->readyReady();

  $owner->mainLoop();
};
```

The input rowops are not only collected in the binding's tray but also chained directly to the labels that print the raw order of arrival. The handling of "_BEGIN_" also remembers its sequence number.

The handler of "_END_" (the "_END_" rowops get produced implicitly at the end of transaction) then does the heavy lifting. It looks at the sequence number remembered from "_BEGIN_" and makes the decision. If the received sequence is the next expected one, the data collected in the tray gets sent on immediately, and then the contents of the @trays array continues being sent on until it hits a blank spot of missing data. Or if the received sequence leaves a gap, the tray is placed into an appropriate spot in @trays for later processing.

This whole logic can be encapsulated in a class but I haven't decided yet on the best way to do it. Maybe some time in the future.

Chapter 17. TQL, Triceps Trivial Query Language

17.1. Introduction to TQL

Triceps by itself is a library that can be embedded into any program to add the CEP functionality. But sometimes having a ready server process that handles the communications and queries and wraps the CEP logic within itself is a great convenience. TQL, the Triceps Trivial Query Language, is used in this server.

The server is useful as both a tool to play with Triceps programs and as an example of implementation. It all started with the example of simple queries in Section 7.9: "Main loop with a socket" (p. 54) , then I've wanted to use the queries to demonstrate a feature of the streaming functions, then I've wanted to use it for a threaded logic demonstration, so it has been growing over time.

This server and the TQL are so far of only an example quality, but TQL is extensible and it already can do some interesting things. The "example quality" means that they work but the set of commands is limited and they don't have an extensive set of tests for every possibility, so it could happen that some corner cases don't work so well.

Why not SQL, after all, there are multiple parser building tools available in Perl? Partially, because I wanted to keep it trivial and to avoid introducing extra dependencies, especially just for the examples. Partially, because I don't like SQL. I think that the queries can be expressed much more naturally in the form of shell-like pipelines. At one of my past jobs I wrote a simple toolkit for querying and comparison of the CSV files (yeah, I didn't find the DBD::CSV module), I've used a pipeline semantics and it worked pretty well. It also did things that are quite difficult with SQL, like mass renaming and reordering of fields, and diffing. Although TQL is not a descendant of the language I've used in that query tool, it is a further development of the pipeline idea. As I've found later, there are other products that also use the pipeline approach for the queries, such as the PowerShell.

17.2. TQL syntax

Syntactically, TQL is very simple: its query is represented as a nested list, similar to Tcl (or if you like Lisp better, you can think that it's similar to Lisp but with different parentheses). A list is surrounded by curly braces "{}". The elements of a list are either other lists or words consisting of non-space characters.

```
{word1 {word21 word22} word3}
```

Unlike Tcl, there are no quotes in the TQL syntax, the quote characters are just the normal word characters. If you want to include spaces into a word, you use the curly braces instead of the quotes.

```
{ this is a {brace-enquoted} string with spaces and nested braces }
```

Note that the spaces inside a list are used as delimiters and thrown away but within a brace-quoted word-string they are significant. How do you know, which way they will be treated in a particular case? It all depends on what is expected in this case. If the command expects a string as an argument, it will treat it as a string. If the command expects a list as an argument, it will treat it as a list.

What if you need to include an unbalanced brace character inside a string? Escape it with a backslash, "\{". The other usual Perl backslash sequences work too (though in the future TQL may get separated from Perl and then only the C sequences will work, that is to be seen). Any non-alphanumeric characters (including spaces) can be prepended with a backslash too. An important point is that when you build the lists, unlike shell, and like Tcl, you do the backslash escaping only once, when accepting a raw string. After that you can include the string with escapes into the lists of any depth without any extra escapes (and you must not add any extra escapes in the lists).

Unlike shell, you can't combine a single string out of the quoted and unquoted parts. Instead the quoting braces work as implicit separators. For example, if you specify a list as {a{b}c d}, you don't get two strings "abc" and "d", you get four strings "a", "b", "c", "d".

The parsing of the lists is done with the package Braced, with some more examples shown in Section 19.14: "Braced reference" (p. 386) .

A TQL query is a list that represents a pipeline. Each element of the list is a command. The first command reads the data from a table, and the following commands perform transformations on that data. For example:

```
{read table tWindow} {project fields {symbol price}} {print tokenized
    0}
```

If the print command is missing at the end of the pipeline, it will be added implicitly, with the default arguments: {print}.

The arguments of each TQL command are always in the option name-value format, very much like the Perl constructors of many Triceps objects. There aren't any arguments in TQL that go by themselves without an option name.

So for example the command read above has an option "table" with value "tWindow". The command project has an option "fields" with a list value of two elements. In this case the elements are simple words and don't need the further bracing. But the braces around it won't hurt. Say, if you wanted to rename the field "price" to "trade_price", you use the Triceps::Fields::filter() syntax for it, and even though the format doesn't contain any spaces and can be still used just as a word, it looks nicer with the braces around it:

```
{project fields {symbol {price/trade_price} }}
```

17.3. TQL commands

I'm sure that the list of commands and their options will expand and change over time. So far the supported commands are:

read
>   Defines a table to read from and starts the command pipeline.
>
>   Options:
>
>   table - name of the table to read from. When a Triceps model gets wrapped in the server, it defines, what tables it has available.

project
>   Projects (and possibly renames) a subset of fields in the current pipeline. In other words, all the files besides the specified ones get thrown away.
>
>   Options:
>
>   fields - an array of field definitions in the syntax of Triceps::Fields::filter() (same as in the joins), as described in Section 19.10: "Fields reference" (p. 379) .

print
>   The last command of the pipeline, which prints the results. If not used explicitly, the query adds this command implicitly at the end of the pipeline, with the default options.
>
>   Options:
>
>   tokenized (optional) - Flag: print in the name-value format, as in Row::printP(). Otherwise prints only the values in the CSV format. Default: 1.

join
> Joins the current pipeline with another table. This is the functionally similar to LookupJoin, although the options are closer to JoinTwo.
>
> Options:
>
> `table` - name of the table to join with. The current pipeline is considered the "left side", the table the "right side". The duplicate key fields on the right side are always excluded from the result, as by the LookupJoin option `fields-DropRightKey => 1`.
>
> `rightIdxPath` (optional) - path name of the table's index on which to join. As usual, the path is an array of nested index type names. If this option is not specified, the index path will be found automatically by the join fields.
>
> `by` (semi-optional) - the join equality condition specified as pairs of fields. Similarly to JoinTwo, it's a single-level array with the fields logically paired:{leftFld1 rightFld1 leftFld2 rightFld2 ... }. Options "by" and "byLeft" are mutually exclusive, and one of them must be present.
>
> `byLeft` (semi-optional) - the join equality condition specified as a transformation on the left-side field set in the syntax of `Triceps::Fields::filter()`, with an implicit element `{!.*}` added at the end. Options "by" and "byLeft" are mutually exclusive, and one of them must be present.
>
> `leftFields` (optional) - the list of patterns for the left-side fields to pass through and possibly rename, in the syntax of `Triceps::Fields::filter()`. Default: pass all, with the same name.
>
> `rightFields` (optional) - the list of patterns for the right-side fields to pass through and possibly rename, in the syntax of `Triceps::Fields::filter()`. The key fields get implicitly removed before. Default: pass all, with the same name.
>
> `type` (optional) - type of the join, "inner" or "left". Default: "inner".

where
> Filters/selects the rows.
>
> Options:
>
> `istrue` - a Perl expression, the condition for the rows to pass through. The particularly dangerous constructions are not allowed in the expression, including the loops and the general function calls. The fields of the row are referred to as `$%field`, these references get translated before the expression is compiled.

Here are some examples of the Tql queries, with results produced from the output of the code examples that will be shown below.

```
query,{read table tSymbol}
query OP_INSERT symbol="AAA" name="Absolute Auto Analytics Inc"
    eps="0.5"
+EOD,OP_NOP,query
```

Reads the stock symbol information table and prints it in the default tokenized format. The input line is CSV, containing as usual in the examples from Section 7.9: "Main loop with a socket" (p. 54) , the command and the data for it. Here the command "query" has been defined to handle the TQL queries. It's possible to define multiple TQL-handling commands in a server, in case if you want them to query different units, or different subsets of the tables. The parsing of the data part is smart enough not to break up the text of the query on the commas in it.

The tokenized result format is a bit messy for now, a mix of tokenized data lines and a CSV end-of-data line.

In the simpler examples in Section 7.9: "Main loop with a socket" (p. 54) the end-of-data has been marked by either a row with opcode OP_NOP or not marked at all. For the TQL queries I've decided to try out a different approach: send a CSV

row on the pseudo-label "+EOD" with the value equal to the name of the command that has been completed. The labels with names starting with "+" are special in this convention, they represent some kind of metadata.

```
query,{read table tWindow} {project fields {symbol price}}
query OP_INSERT symbol="AAA" price="20"
query OP_INSERT symbol="AAA" price="30"
+EOD,OP_NOP,query
```

Reads the trade window rows and projects the fields "symbol" and "price" from them.

```
query,{read table tWindow} {project fields {symbol price}} {print
    tokenized 0}
query,OP_INSERT,AAA,20
query,OP_INSERT,AAA,30
+EOD,OP_NOP,query
```

The same, only explicitly prints the data in the CSV format.

```
query,{read table tWindow} {where istrue {$%price == 20}}
query OP_INSERT id="3" symbol="AAA" price="20" size="20"
+EOD,OP_NOP,query
```

Selects the trade window row with price equal to 20.

```
query,{read table tWindow} {join table tSymbol byLeft {symbol}}
query OP_INSERT id="3" symbol="AAA" price="20" size="20"
    name="Absolute Auto Analytics Inc" eps="0.5"
query OP_INSERT id="5" symbol="AAA" price="30" size="30"
    name="Absolute Auto Analytics Inc" eps="0.5"
+EOD,OP_NOP,query
```

Reads the trade window and enriches it by joining with the symbol information.

A nice feature of TQL is that it allows to combine the operations in the pipeline in any order, repeated any number of times. For example, you can read a table, filter it, join with another table, filter again, join with the third table, filter again and so on. SQL in the same situation has to resort to specially named clauses, for example *WHERE* filters before grouping and *HAVING* filters after grouping.

Of course, a typical smart SQL compiler would determine the earliest application point for each *WHERE* sub-expression and build a similar pipeline. But TQL allows to keep the compiler trivial, following the explicit pipelining in the query. And nothing really prevents a smart TQL compiler either, it could as well analyze, split and reorder the pipeline stages.

As mentioned above, the TQL queries are compiled before the execution into the normal Triceps code. A query is built in a separate unit. After the query is built, the data is fed into it to produce the result, and then the unit gets destroyed. Potentially, TQL could be extended for writing the general Triceps programs as well.

17.4. TQL in a single-threaded server

The TQL support may be instantiated in both the single-threaded and multi-threaded applications. The single-threaded support is simpler, so we'll look at it first. The TQL itself stays the same in both cases, and even the way to construct the TQL server is similar but then the way for the TQL queries to extract the data from the application is different.

The code that produced the query output examples from the previous section looks like this:

```
# The basic table type to be used for querying.
# Represents the trades reports.
our $rtTrade = Triceps::RowType->new(
  id => "int32", # trade unique id
  symbol => "string", # symbol traded
```

```
    price => "float64",
    size => "float64", # number of shares traded
);

our $ttWindow = Triceps::TableType->new($rtTrade)
  ->addSubIndex("bySymbol",
    Triceps::SimpleOrderedIndex->new(symbol => "ASC")
      ->addSubIndex("last2",
        Triceps::IndexType->newFifo(limit => 2)
      )
  )
;
$ttWindow->initialize();

# Represents the static information about a company.
our $rtSymbol = Triceps::RowType->new(
  symbol => "string", # symbol name
  name => "string", # the official company name
  eps => "float64", # last quarter earnings per share
);

our $ttSymbol = Triceps::TableType->new($rtSymbol)
  ->addSubIndex("bySymbol",
    Triceps::IndexType->newHashed(key => [ "symbol" ])
  )
;
$ttSymbol->initialize();

my $uTrades = Triceps::Unit->new("uTrades");
my $tWindow = $uTrades->makeTable($ttWindow, "tWindow");
my $tSymbol = $uTrades->makeTable($ttSymbol, "tSymbol");

# The information about tables, for querying.
my $tql = Triceps::X::Tql->new(
  name => "tql",
  tables => [
    $tWindow,
    $tSymbol,
  ],
);

my %dispatch;
$dispatch{$tWindow->getName()} = $tWindow->getInputLabel();
$dispatch{$tSymbol->getName()} = $tSymbol->getInputLabel();
$dispatch{"query"} = sub { $tql->query(@_); };
$dispatch{"exit"} = \&Triceps::X::SimpleServer::exitFunc;

# calls Triceps::X::SimpleServer::startServer(0, \%dispatch);
Triceps::X::DumbClient::run(\%dispatch);
```

It's very much like the example shown before in Section 7.9: "Main loop with a socket" (p. 54) , with one of the dispatch entries being a TQL query method. The list of tables is given to the Tql object which figures out all by itself how to run the queries on them.

Just as before, DumbClient is a class for both starting the server and running the unit tests on it, so in reality the dispatch table is handled in `SimpleServer::startServer()`.

The dispatched labels have the CSV line received from the socket broken up into the fields and formed into the rowops but the dispatched functions receive the whole argument line as the client had sent it. The functions can then do the text parsing in their own way, which comes real handy for TQL. The `Tql::query()` method splits off the name of the label to be used as the name of the query, and parses the rest as the query body.

There are multiple ways to create a Tql object. By default the option "tables" lists all the queryable tables, and their "natural" names will be used in the queries, as was shown above. It's possible to specify the names explicitly as well:

```
my $tql = Triceps::X::Tql->new(
  name => "tql",
  tables => [
    $tWindow,
    $tSymbol,
    $tWindow,
    $tSymbol,
  ],
  tableNames => [
    "window",
    "symbol",
    $tWindow->getName(),
    $tSymbol->getName(),
  ],
);
```

This version defines each table under two synonymous names. The tables and their names go in the parallel arrays in the same order.

It's also possible to create a Tql object without tables, and add tables to it later as they are created:

```
my $tql = Triceps::X::Tql->new(name => "tql");
$tql->addNamedTable(
  window => $tWindow,
  symbol => $tSymbol,
);
# add 2nd time, with different names
$tql->addTable(
  $tWindow,
  $tSymbol,
);
$tql->initialize();
```

Multiple tables can be added in one method call, as shown here. The tables can be added with explicit names or with "natural" names. After all the tables are added, the Tql object has to be initialized.

The two ways of creation are mutually exclusive: if the option "tables" is used, the object will be initialized right away in the constructor. If this option is absent, the explicit initialization has to be done later. The methods `addTable()` and `addNamedTable()` cannot be used on an initialized table, and `query()` cannot be used on an uninitialized table.

17.5. TQL in a multi-threaded server

As the single-threaded version of TQL works symbiotically with the SimpleServer, the multithreaded version works with the ThreadedServer. The multithreaded version of TQL does all that the single-threaded one does, and more: it allows to define the dynamic queries. Some day TQL might become the language to define the whole Triceps models.

One thread created by the programmer contains the "core logic" of the model. It doesn't technically have to be all in a single thread: the data can be forwarded to the other threads and then the results forwarded back from them. But a single core logic thread is a convenient simplification. This thread has some input labels, to receive data from the outside, and some tables with the computed results that can be read by TQL. Of course, it's entirely realistic to have also just the output labels without tables, sending a stream of computed rowops, but again for simplicity let's leave this out for now.

This core logic thread creates a TQL instance, which listens on a socket, accepts the connections, forwards the input data to the core logic, performs queries on the tables from the core logic and sends the results back to the client. To this end, the TQL instance creates a few nexuses in the core logic thread and uses them to communicate between all the fragments.

The input labels and tables in the core thread also get properly connected to these nexuses. Figure 17.1 shows the thread structure, I'll use it for the reference throughout the discussion.
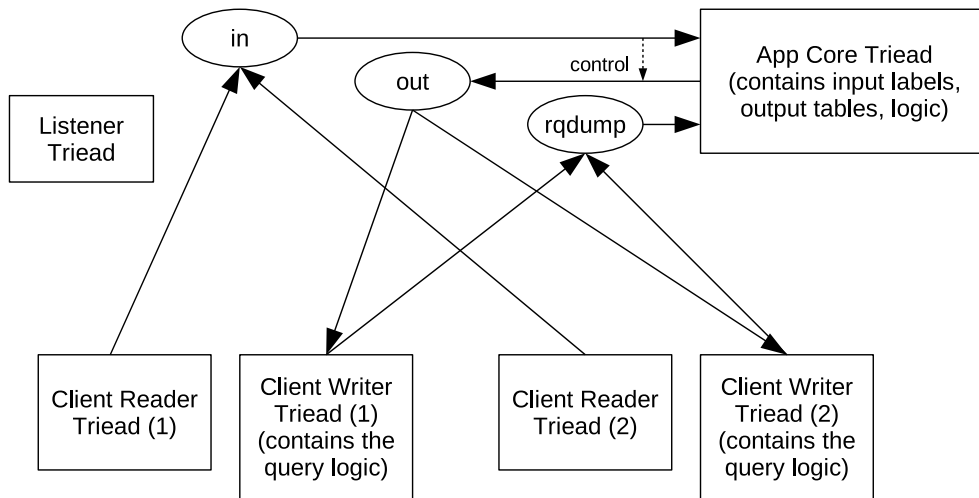
Figure 17.1. Multithreaded TQL application structure.

The core logic thread then goes into its main loop and performs as its name says, the core logic computations.

Here is a very simple example of a TQL application:

```
sub appCoreT # (@opts)
{
  my $opts = {};
  &Triceps::Opt::parse("appCoreT", $opts, {@Triceps::Triead::opts,
    socketName => [ undef, \&Triceps::Opt::ck_mandatory ],
  }, @_);
  undef @_; # avoids a leak in threads module
  my $owner = $opts->{owner};
  my $app = $owner->app();
  my $unit = $owner->unit();

  # build the core logic

  my $rtTrade = Triceps::RowType->new(
    id => "int32", # trade unique id
    symbol => "string", # symbol traded
    price => "float64",
    size => "float64", # number of shares traded
  );

  my $ttWindow = Triceps::TableType->new($rtTrade)
    ->addSubIndex("byId",
      Triceps::SimpleOrderedIndex->new(id => "ASC")
    )
  ;
  $ttWindow->initialize();
```

```
  # Represents the static information about a company.
  my $rtSymbol = Triceps::RowType->new(
    symbol => "string", # symbol name
    name => "string", # the official company name
    eps => "float64", # last quarter earnings per share
  );

  my $ttSymbol = Triceps::TableType->new($rtSymbol)
    ->addSubIndex("bySymbol",
      Triceps::SimpleOrderedIndex->new(symbol => "ASC")
    )
  ;
  $ttSymbol->initialize();

  my $tWindow = $unit->makeTable($ttWindow, "tWindow");
  my $tSymbol = $unit->makeTable($ttSymbol, "tSymbol");

  # export the endpoints for TQL (it starts the listener)
  my $tql = Triceps::X::Tql->new(
    name => "tql",
    trieadOwner => $owner,
    socketName => $opts->{socketName},
    tables => [
      $tWindow,
      $tSymbol,
    ],
    tableNames => [
      "window",
      "symbol",
    ],
    inputs => [
      $tWindow->getInputLabel(),
      $tSymbol->getInputLabel(),
    ],
    inputNames => [
      "window",
      "symbol",
    ],
  );

  $owner->readyReady();

  $owner->mainLoop();
}

{
  my ($port, $thread) = Triceps::X::ThreadedServer::startServer(
      app => "appTql",
      main => \&appCoreT,
      port => 0,
      fork => -1, # create a thread, not a process
  );
}
```

This core logic is the same as in the single-threaded example: all it does is create two tables and then send the input data into them. The server gets started in a background thread (`fork => -1`) because this code is taken from a test that then goes and runs the `expect` with the ThreadedClient.

The specification of tables for TQL is the same as for the single-threaded version. The new options available only in the multi-threaded mode are the "threadOwner", "inputs" and "inputNames". The "threadOwner" is how TQL knows that it must run in the multithreaded mode, and it's used to create the nexuses for communication between the core logic and

the rest of TQL. The "inputs" are needed because the multithreaded TQL parses and forwards the input data, unlike the single-threaded version that relies on the SimpleServer to do that according to the user-defined dispatch table.

The names options don't have to be used: if you name your labels and tables nicely and suitable for the external vieweing, the renaming-for-export can be skipped.

Similarly to the single-threaded version, if any of the options "tables" or "inputs" is used, the TQL object gets initialized automatically, otherwise the tables and inputs can be added piecemeal with `addTable()`, `addNamedTable()`, `addInput()`, `addNamedInput()`, and then the whole thing initialized manually.

Then the clients can establish the connections with the TQL server, send in the data and the queries. To jump in, here is a trace of a simple session that sends some data, then does some table dumps and subscribes, not touching the queries yet. I'll go through it fragment by fragment and explain the meaning. The dumps and subscribes were the warm-up exercises before writing the full queries, but they're useful in their own right, and here they serve as the warm-up exercises for the making of the queries!

The trace is marked with the client name "c1", just as it is in Section 16.6: "Dynamic threads and fragments in a socket server" (p. 304) , since it's also a trace from SimpleClient.

```
connect c1
c1|ready
```

The "connect" is not an actual command send but just the indication in the trace that the connection was set up by the client "c1". The "ready" response is sent when the connection is opened.

```
c1|subscribe,s1,symbol
c1|subscribe,s1,symbol
```

This is a subscription request. It means "I'm not interested in the current state of a table but send me all the updates". The response is the mirror of the request, so that the client knows that the request has been processed. The format of the requests is different in the multi-threaded mode than in the single-threaded, it has the extra elements. The first element is always the command. "s1" is the unique identifier of the request, so that the client can match together the responses it received to the requests it sent. Keeping the uniqueness is up to the client, the server may refuse the requests with duplicate identifiers. And "symbol" is the name of the table. Once a subscription is in place, there is no way to unsubscribe other than by disconnecting the client (it's doable but adds complications, and I wanted to skip over the nonessential parts). Subscribing multiple times to the same table will send a confirmation every time but the repeated confirmations will have no effect: only one copy of the data will be sent anyway.

Side-tracking a bit, if a second subscription attempt is to be done with the same id, the error would look like:

```
error,s1,Duplicate id 's1': query ids must be unique,bad_id,s1
```

The response type field contains "error", followed by the id of the request, the error message, the error name, and the data that caused the error.

Let's send the data into the model:

```
c1|d,symbol,OP_INSERT,ABC,ABC Corp,1.0
c1|d,symbol,OP_INSERT,ABC,ABC Corp,1
```

And since it propagates through the subscription, the data gets sent back too. The "symbol" here means two different things: on the input side it's the name of the label where the data is sent, on the output side it's the name of the table that has been subscribed to.

The data lines start with the command "d" (since the data is sent much more frequently than the commands, I've picked a short one-letter command name for it), then the label/table name, opcode and the row fields in CSV format.

```
c1|confirm,cf1
c1|confirm,cf1,,,
```

The "confirm" command provides a way for the client to check that the data it send had propagated through the model. And it doesn't have to subscribe back to the data and read them. Send some data lines, then send the "confirm" command and wait for it to come back (again, the unique id allows to keep multiple confirmations in flight if you please). This command doesn't guarantee that all the clients have seen the results from that data. It only guarantees that the core logic had seen the data, and more weakly guarantees that the data had been processed by the core logic, and this particular client had already seen all the results from it.

Why weakly? It has to do with the way it works inside, and it depends on the core logic. If the core logic consists of one thread, the guarantee is quite strong. But if the core logic farms out the work from the main thread to the other threads and then collects the results back, the guarantee breaks.

You can see in Figure 17.1 that unlike the chat server shown Section 16.6: "Dynamic threads and fragments in a socket server" (p. 304), TQL doesn't have any private nexuses for communication between the reader and writer threads of a client. Instead it relies on the same input and output nexuses, adding a control label to them, to forward the commands from the reader to the writer. The TQL object in the core logic thread creates a short-circuit connection between the control labels in the input and output nexuses, forwarding the commands. And if the core logic all runs in one thread, this creates a natural pipeline: the data comes in, gets processed, comes out, the "confirm" command comes in, comes out after the data. But if the core logic farms out the work to more threads, the confirmation can "jump the line" because its path is a direct short circuit.

```
c1|drain,dr1
c1|drain,dr1,,,
```

The "drain" is an analog of "confirm" but more reliable and slower: the reader thread drains the whole model before sending the command on. This guarantees that all the processing is done, and all the output from it has been sent to all the clients.

```
c1|dump,d2,symbol
c1|startdump,d2,symbol
c1|d,symbol,OP_INSERT,ABC,ABC Corp,1
c1|dump,d2,symbol
```

The "dump" command dumps the current contents of a table. Its result starts with "startdump", and the same id and table name as in the request, then goes the data (all with OP_INSERT), finishing with the completion confirmation echoing the original command. The dump is atomic, the contents of the table doesn't change in the middle of the dump. However if a subscription on this table is active, the data rows from that subscription may come before and after the dump.

```
c1|dumpsub,ds3,symbol
c1|startdump,ds3,symbol
c1|d,symbol,OP_INSERT,ABC,ABC Corp,1
c1|dumpsub,ds3,symbol
```

The "dumpsub" command is a combination of a dump and subscribe: get the initial state and then get all the updates. The confirmation of "dumpsub" marks the boundary between the original dump and the following updates.

```
c1|d,symbol,OP_INSERT,DEF,Defense Corp,2.0
c1|d,symbol,OP_INSERT,DEF,Defense Corp,2
```

Send some more data, and it comes back only once, even though the subscription was done twice: once in "subscribe" and once in "dumpsub". The repeated subscription requests simply get consumed into one subscription.

```
c1|d,window,OP_INSERT,1,ABC,101,10
```

This sends a row to the other table but nothing comes back because there is no subscription to that table.

```
c1|dumpsub,ds4,window
c1|startdump,ds4,window
c1|d,window,OP_INSERT,1,ABC,101,10
c1|dumpsub,ds4,window
c1|d,window,OP_INSERT,2,ABC,102,12
c1|d,window,OP_INSERT,2,ABC,102,12
```

This demonstrates the pure dump-and-subscribe without any interventions.

```
c1|shutdown
c1|shutdown,,,,
c1|__EOF__
```

And the shutdown command works the same as in the chat server, draning and then shutting down the whole server.

Now on to the queries.

```
connect c1
c1|ready
c1|d,symbol,OP_INSERT,ABC,ABC Corp,1.0
```

Starts a client connection and sends some data.

```
c1|querysub,q1,query1,{read table symbol}{print tokenized 0}
c1|d,query1,OP_INSERT,ABC,ABC Corp,1
c1|querysub,q1,query1
```

The "querysub" command does the "query-and-subscribe": reads the initial state of the table, processed through the query, and then subscribes to any future updates. The single-threaded variety of TQL doesn't do this, it does just the one-time queries. The multithreaded TQL could potentially do the one-time queries, and also just the subscribes without the initial state, but so far I've been cutting corners and the only thing that's actually available is the combination of two, the "query-sub".

Similarly to the other commands, "q1" is the command identifier. The next field "query1" is the name for the query, it's the name that will be shown for the data lines coming out of the query. And then goes the query in the brace-quoted format, same as in the single-threaded TQL (and there is no further splitting by commas, so the commas can be used freely in the query).

The identifier and the name of the query sound kind of redundant. But the client may generate them in different ways and need both. The name has the more symbolic character. The identifier can be generated as a sequence of numbers, so that the client can keep track of its progress more easily. And the error reports include the identifier but not the query name in them.

For the query, there is no special line coming out before the initial dump. Supposedly, there would not be more than one query in flight with the same name, so they could be easily told apart based on the name in the data lines. There is also an underlying consideration that when the query involves a join, in the future the initial dump might be happening in multiple chunks, requiring to either surround every chunk with the start-end lines or just let them go without the extra notifications, as they are now.

And the initial dump ends as usual with getting the echo of the command (without the query part) back.

This particular query is very simple and equivalent to a "dumpsub".

```
c1|d,symbol,OP_INSERT,DEF,Defense Corp,2.0
c1|d,query1,OP_INSERT,DEF,Defense Corp,2
```

Send more data and it will come out of the query.

```
c1|querysub,q2,query2,{read table symbol}{where istrue {$%symbol =~
    /^A/}}{project fields {symbol eps}}
c1|t,query2,query2 OP_INSERT symbol="ABC" eps="1"
c1|querysub,q2,query2
```

This query is more complicated, doing a selection (the "where" query command) and projection. It also prints the results in the tokenized format (the "print" command gets added automatically if it wasn't used explicitly, and the default options for it enable the tokenized format).

The tokenized lines come out with the command "t", query name and then the contents of the row. The query name happens to be sent twice, and I'm not sure yet if it's a feature or a bug.

```
c1|d,symbol,OP_INSERT,AAA,Absolute Auto Analytics Inc,3.0
c1|d,query1,OP_INSERT,AAA,Absolute Auto Analytics Inc,3
c1|t,query2,query2 OP_INSERT symbol="AAA" eps="3"
c1|d,symbol,OP_DELETE,DEF,Defense Corp,2.0
c1|d,query1,OP_DELETE,DEF,Defense Corp,2
```

More examples of the data sent, getting processed by both queries. In the second case the "where" filters out the row from query2, so only query1 produces the result.

```
c1|shutdown
c1|shutdown,,,,
c1|__EOF__
```

And the shutdown as usual.

Now the *piece de resistance*: queries with joins.

```
connect c1
c1|ready
c1|d,symbol,OP_INSERT,ABC,ABC Corp,2.0
c1|d,symbol,OP_INSERT,DEF,Defense Corp,2.0
c1|d,symbol,OP_INSERT,AAA,Absolute Auto Analytics Inc,3.0
c1|d,window,OP_INSERT,1,AAA,12,100
```

Connect and send some starting data.

```
c1|querysub,q1,query1,{read table window}{join table symbol byLeft
    {symbol} type left}
c1|t,query1,query1 OP_INSERT id="1" symbol="AAA" price="12" size="100"
    name="Absolute Auto Analytics Inc" eps="3"
c1|querysub,q1,query1
```

A left join of the tables "window" and "symbol", by the field "symbol" as join condition.

The TQL joins, even in the multithreaded mode, are still implemented internally as LookupJoin, driven only by the main flow of the query. So the changes to the joined dimension tables will not update the query results, and will be visible only when a change on the main flow picks them up, potentially creating inconsistencies in the output. This is wrong, but fixing it presents complexities that I've left alone until some later time.

```
c1|d,window,OP_INSERT,2,ABC,13,100
c1|t,query1,query1 OP_INSERT id="2" symbol="ABC" price="13" size="100"
    name="ABC Corp" eps="2"
c1|d,window,OP_INSERT,3,AAA,11,200
c1|t,query1,query1 OP_INSERT id="3" symbol="AAA" price="11" size="200"
    name="Absolute Auto Analytics Inc" eps="3"
```

Sending the data updates the results of the query.

```
c1|d,symbol,OP_DELETE,AAA,Absolute Auto Analytics Inc,3.0
c1|d,symbol,OP_INSERT,AAA,Alcoholic Abstract Aliens,3.0
```

As described above, the modifications of the dimension table are not visible in the query directly.

```
c1|d,window,OP_DELETE,1
c1|t,query1,query1 OP_DELETE id="1" symbol="AAA" price="12" size="100"
    name="Alcoholic Abstract Aliens" eps="3"
```

But an update on the main flow brings them up (an in this case inconsistently, the row getting deleted is not exactly the same as the row inserted before).

```
c1|querysub,q2,query2,{read table window}{join table symbol byLeft
    {symbol} type left}{join table symbol byLeft {eps} type left
```

```
     rightFields {symbol/symbol2}}
c1|t,query2,query2 OP_INSERT id="2" symbol="ABC" price="13" size="100"
    name="ABC Corp" eps="2" symbol2="ABC"
c1|t,query2,query2 OP_INSERT id="2" symbol="ABC" price="13" size="100"
    name="ABC Corp" eps="2" symbol2="DEF"
c1|t,query2,query2 OP_INSERT id="3" symbol="AAA" price="11" size="200"
    name="Alcoholic Abstract Aliens" eps="3" symbol2="AAA"
c1|querysub,q2,query2
```

This is a more complicated query, involving two joins, with the same dimension table "symbol". The second join by "eps" makes no real-world sense whatsoever but it's interesting from the technical perspective: if you check the table type of this table at the start of the section, you'll find that it has no index on the field "eps". The join adds this index on demand!

The way it works, all the dimension tables are copied into the client's writer thread, created from the table types exported by the core logic throuhg the output nexus. (And if a table is used in the same query twice, it's currently also copied twice). This provides a nice opportunity to amend the table type by adding any necessary secondary index before creating the table, and TQL makes a good use of it.

17.6. Internals of a TQL join

The TQL implementation provides interesting examples of the sophisticated Triceps usage. Remember, it isn't somehow special or monolothic with the rest of Triceps. It uses the normal triceps API and anyone can write a similar language layer from scratch. The code is located in `perl/Triceps/lib/Triceps/X/DumbClient.pm`.

The most complex operation implemented in TQL is the join in the multi-threaded server. It requires copying of multiple tables from the core logic thread to the client thread that executes the join, and rebuilding them in the new thread, possibly with the automatically added indexes.

It all starts in the Tql initialization method. In the multithreaded mode it builds the nexuses for communication. I'll skip the input nexus (it's straightforward) and show the building of only the output and request-dump nexuses:

```
  # row type for dump requests and responses
  my $rtRequest = Triceps::RowType->new(
    client => "string", #requesting client
    id => "string", # request id
    name => "string", # the table name, for convenience of requestor
    cmd => "string", # for convenience of requestor, the command that it is executing
  );
```

The request row type is used by the client writer thread to request the table dumps from the core logic, and to get back the notifications about the dumps.

```
  # build the output side
  for (my $i = 0; $i <= $#{$self->{tables}}; $i++) {
    my $name = $self->{tableNames}[$i];
    my $table = $self->{tables}[$i];

    push @tabtypes, $name, $table->getType()->copyFundamental();
    push @labels, "t.out." . $name, $table->getOutputLabel();
    push @labels, "t.dump." . $name, $table->getDumpLabel();
  }
  push @labels, "control", $rtControl; # pass-through from in to out
  push @labels, "beginDump", $rtRequest; # framing for the table dumps
  push @labels, "endDump", $rtRequest;

  $self->{faOut} = $owner->makeNexus(
    name => $self->{nxprefix} . "out",
    labels => [ @labels ],
    tableTypes => [ @tabtypes ],
```

```
      import => "writer",
  );
  $self->{beginDump} = $self->{faOut}->getLabel("beginDump");
  $self->{endDump} = $self->{faOut}->getLabel("endDump");
```

On the output side each table is represented by 3 elements:

- its fundamental table type (stripped down to the primary index);

- its output label for normal updates;

- its dump label for the responses to the dump requests.

There also are the "beginDump" and "endDump" labels that frame each response to a dump request.

The row type `$rtControl` and label "control" are used to pass the commands from the client reader to client writer, but its exact contents is not important here. All the magic happens in the client writer thread. The reader thread just passes the commands to the writer thread, and then the writer thread controls the parsing and processing of the commands.

The dump request nexus is built in a similar way:

```
  # build the dump requests, will be coming from below
  undef @labels;
  for (my $i = 0; $i <= $#{$self->{tables}}; $i++) {
    my $name = $self->{tableNames}[$i];
    my $table = $self->{tables}[$i];

    push @labels, "t.rqdump." . $name, $rtRequest;
  }
  $self->{faRqDump} = $owner->makeNexus(
    name => $self->{nxprefix} . "rqdump",
    labels => [ @labels ],
    reverse => 1, # avoids making a loop, and gives priority
    import => "reader",
  );
  # tie together the labels
  for (my $i = 0; $i <= $#{$self->{tables}}; $i++) {
    my $name = $self->{tableNames}[$i];
    my $table = $self->{tables}[$i];

    $self->{faRqDump}->getLabel("t.rqdump." . $name)->makeChained(
      $self->{nxprefix} . "rqdump." . $name, undef,
      \&_dumpTable, $self, $table
    );
  }
```

Each table has a label created for requesting its contents. The dumps are executed in the function _dumpTable:

```
sub _dumpTable # ($label, $rowop, $self, $table)
{
  my ($label, $rop, $self, $table) = @_;
  my $unit = $label->getUnit();
  # pass through the client id to the dump
  $unit->call($self->{beginDump}->adopt($rop));
  $table->dumpAll();
  $unit->call($self->{endDump}->adopt($rop));
  $self->{faOut}->flushWriter();
}
```

The data gets framed around by the copies of the original request sent to the labels "beginDump" and "endDump". This helps the client writer thread keep track of its current spot. The flushing of the writer is not strictly needed. Just in case if

multiple dump requests are received in a single tray, the flush breaks up the responses into a separate tray for each dump, keeping the size of the trays lower. Not that this situation could actually happen yet.

This part taken care of, let's jump around and see how the client writer thread processes a "querysub" command:

```
  } elsif ($cmd eq "querysub") {
    if ($id eq "" || exists $queries{$id}) {
      printOrShut($app, $fragment, $sock,
        "error,$id,Duplicate id '$id': query ids must be unique,bad_id,$id\n");
      next;
    }
    my $ctx = compileQuery(
      qid => $id,
      qname => $args[0],
      text => $args[1],
      subError => sub {
        chomp $_[2];
        $_[2] =~ s/\n/\\n/g; # no real newlines in the output
        $_[2] =~ s/,/;/g; # no confusing commas in the output
        printOrShut($app, $fragment, $sock, "error,", join(',', @_), "\n");
      },
      faOut => $faOut,
      faRqDump => $faRqDump,
      subPrint => sub {
        printOrShut($app, $fragment, $sock, @_);
      },
    );
    if ($ctx) { # otherwise the error is already reported
      $queries{$id} = $ctx;
      &$runNextRequest($ctx);
    }
  }
}
```

The query id is used to keep track of the outstanding queries, so the code makes sure that it's unique. This is the origin of the duplicate id error that was shown before.

The bulk of the work is done in the method `compileQuery()`. Its arguments supply the details of the query and also provide the closures for the functionality that differs between the single-threaded and multi-threaded versions. The option "subError" is used to send the errors to the client, and "subPrint" is used to send the output to the client, it gets used for building the labels in the "print" command of the query.

`compileQuery()` returns the query context, which contains a compiled sub-model that executes the query and a set of requests that tell the writer how to connect the query to the incoming data. Or on error it reports the error using the closure from the option "subError" and returns an `undef`. If the compilation succeeded, the writer thread remembers the query and starts the asynchronous execution of the requests. The requests are the data dump requests to be sent back to the application core thread. More about the requests later, now let's look at the query compilation and context.

The context is created in `compileQuery()` thusly:

```
  my $ctx = {};
  $ctx->{qid} = $opts->{qid};
  $ctx->{qname} = $opts->{qname};

  # .. skipped the parts related to single-threadde TQL

  $ctx->{faOut} = $opts->{faOut};
  $ctx->{faRqDump} = $opts->{faRqDump};
  $ctx->{subPrint} = $opts->{subPrint};
  $ctx->{requests} = []; # dump and subscribe requests that will run the pipeline
  $ctx->{copyTables} = []; # the tables created in this query
    # (have to keep references to the tables or they will disappear)
```

```
# The query will be built in a separate unit
$ctx->{u} = Triceps::Unit->new($opts->{nxprefix} . "${q}.unit");
$ctx->{prev} = undef; # will contain the output of the previous command in the pipeline
$ctx->{id} = 0; # a unique id for auto-generated objects
# deletion of the context will cause the unit in it to clean
$ctx->{cleaner} = $ctx->{u}->makeClearingTrigger();
```

It has some parts common and some parts differing for the single- and multi-threaded varieties, here I've skipped over the single-threaded parts.

One element that is left undefined here is `$ctx->{prev}`. It's the label created as the output of the previous stage of the query pipeline. As each command in the pipeline builds its piece of processing, it chains its logic from `$ctx->{prev}` and leaves its result label in `$ctx->{next}`. Then `compileQuery()` moves `next` to `prev` and calls the compilation of the next command in the pipeline. The only command that accepts an undefined `prev` (and it must be undefined for it) is "read" which reads the table at the start of the pipeline.

`$ctx->{copyTables}` also has an important point behind it. When you create a label, it's OK to discard the original reference after you chain the label into the logic, that chaining will keep a reference, and the label will stay alive. Not so with a table: if you create a table, then chain its input label and drop the reference to a table, the table will be discarded. When the input label would try to send any data to the table, it will die. So it's important to keep the direct table references, and that's what this array is for.

`$ctx->{id}` is used to generate the unique names for the objects built in this context.

Each query is built in its own unit. This is convenient, after the query is done or the compilation encounters an error, the unit with its whole contents can be easily discarded. The clearing trigger placed in the context makes sure that the unit gets properly cleared and discarded.

Next goes the compilation of the join query command, I'll go through it in chunks.

```
sub _tqlJoin # ($ctx, @args)
{
  my $ctx = shift;
  die "The join command may not be used at the start of a pipeline.\n"
    unless (defined($ctx->{prev}));
  my $opts = {};
  &Triceps::Opt::parse("join", $opts, {
    table => [ undef, \&Triceps::Opt::ck_mandatory ],
    rightIdxPath => [ undef, undef ],
    by => [ undef, undef ],
    byLeft => [ undef, undef ],
    leftFields => [ undef, undef ],
    rightFields => [ undef, undef ],
    type => [ "inner", undef ],
  }, @_);

  my $tabname = bunescape($opts->{table});
  my $unit = $ctx->{u};
  my $table;

  &Triceps::Opt::checkMutuallyExclusive("join", 1, "by", $opts->{by}, "byLeft", $opts-
>{byLeft});
  my $by = split_braced_final($opts->{by});
  my $byLeft = split_braced_final($opts->{byLeft});

  my $rightIdxPath;
  if (defined $opts->{rightIdxPath}) { # propagate the undef
    $rightIdxPath = split_braced_final($opts->{rightIdxPath});
  }
```

It starts by parsing the options and converting them to the internal representation, removing the braced quotes.

```
  # If we were to use a JoinTwo (which is more correct), the data
  # incoming through the query would have to be put into a table too.
  # And that requires finding the primary key for the data.
  # I suppose, after the sequence ids for the rows would get worked
  # out, that would provide the easy default primary key.
  if ($ctx->{faOut}) {
    # Potentially, the tables might be reused between multiple joins
    # in the query if the required keys match. But for now keep things
    # simpler by creating a new table from scratch each time.

    my $tt = eval {
      # copy to avoid adding an index to the original type
      $ctx->{faOut}->impTableType($tabname)->copy();
    };
    die ("Join found no such table '$tabname'\n") unless ($tt);

    if (!defined $rightIdxPath) {
      # determine or add the index automatically
      my @workby;
      if (defined $byLeft) { # need to translate
        my @leftfld = $ctx->{prev}->getRowType()->getFieldNames();
        @workby = &Triceps::Fields::filterToPairs("Join option 'byLeft'",
          \@leftfld, [ @$byLeft, "!.*" ]);
      } else {
        @workby = @$by;
      }

      my @idxkeys; # extract the keys for the right side table
      for (my $i = 1; $i <= $#workby; $i+= 2) {
        push @idxkeys, $workby[$i];
      }
      $rightIdxPath = [ $tt->findOrAddIndex(@idxkeys) ];
    }

    # build the table from the type
    $tt->initialize();
    $table = $ctx->{u}->makeTable($tt, "tab" . $ctx->{id} . $tabname);
    push @{$ctx->{copyTables}}, $table;

    # build the request that fills the table with data and then
    # keeps it up to date;
    # the table has to be filled before the query's main flow starts,
    # so put the request at the front
    &_makeQdumpsub($ctx, $tabname, 1, $table->getInputLabel());
  } else {
    die ("Join found no such table '$tabname'\n")
      unless (exists $ctx->{tables}{$tabname});
    $table = $ctx->{tables}{$tabname};
  }
```

The presence of $ctx->{faOut} means that the query is compiled in the multithreaded context.

The command handlers, including the join command handler, may freely die, and the error messages will be caught by compileQuery() and nicely (at least, sort-of) reported back to the user.

If an explicit "rightIdxPath" option was not requested, it gets found or added automatically. On the way there the index fields need to be determined. Which can be specified either as the explicit field pairs in the option "by" or as the name translation syntax in the option "byLeft". If we've got a "byLeft", first it gets translated to the same format as "by", and then the right-side fields are extracted from the format of "by". After that $tt->findOrAddIndex() takes care of all

the heavy lifting. It either finds a matching index type in the table type or creates a new one from the specified fields, and either way returns the index path (an invalid field will make it confess).

You might wonder, how come the explicit "rightIdxPath" option is not checked in any way? It will be checked later by LookupJoin(), so not much point in doing the check twice.

After that the table is created in a straightforward way, and rememebered in $ctx->{copyTables}. And the requests list gets prepended with a request to dump and subscribe to this table in _makeQdumpsub(). I'll get back to that, for now let's finish up with _tqlJoin().

```perl
my $isLeft = 0; # default for inner join
my $type = $opts->{type};
if ($type eq "inner") {
  # already default
} elsif ($type eq "left") {
  $isLeft = 1;
} else {
  die "Unsupported value '$type' of option 'type'.\n"
}

my $leftFields = split_braced_final($opts->{leftFields});
my $rightFields = split_braced_final($opts->{rightFields});

my $join = Triceps::LookupJoin->new(
  name => "join" . $ctx->{id},
  unit => $unit,
  leftFromLabel => $ctx->{prev},
  rightTable => $table,
  rightIdxPath => $rightIdxPath,
  leftFields => $leftFields,
  rightFields => $rightFields,
  by => $by,
  byLeft => $byLeft,
  isLeft => $isLeft,
  fieldsDropRightKey => 1,
);

$ctx->{next} = $join->getOutputLabel();
}
```

The rest of the options get parsed, and then all the collected data gets forwarded to the LookupJoin constructor. Finally the next label is assigned from the join's result.

Now jumping to the _makeQdumpsub(). It's used by both the "read" and "join" query commands to initiate the joins and subscriptions.

```perl
sub _makeQdumpsub # ($ctx, $tabname, [$front, $lbNext])
{
  my $ctx = shift;
  my $tabname = shift;
  my $front = shift;
  my $lbNext = shift;

  my $unit = $ctx->{u};

  my $lbrq = eval {
    $ctx->{faRqDump}->getLabel("t.rqdump.$tabname");
  };
  my $lbsrc = eval {
    $ctx->{faOut}->getLabel("t.out.$tabname");
  };
```

```
    die ("Found no such table '$tabname'\n") unless ($lbrq && $lbsrc);

    # compute the binding for the data dumps, that would be a cross-unit
    # binding to the original faOut but it's OK
    my $fretOut = $ctx->{faOut}->getFnReturn();
    my $dumpname = "t.dump.$tabname";
    # the dump and following subscription data will merge on this label
    if (!defined $lbNext) {
      $lbNext = $unit->makeDummyLabel(
        $lbsrc->getRowType(), "lb" . $ctx->{id} . "out_$tabname");
    }

    my $bindDump = Triceps::FnBinding->new(
      on => $fretOut,
      name => "bind" . $ctx->{id} . "dump",
      labels => [ $dumpname => $lbNext ],
    );
```

First it finds all the proper labels. The label `$lbNext` will accept the merged dump contents and the following subscription, and it might be either auto-generated or received as an argument. A join will pass it as an argument, set to `$table->getInputLabel()`, so all the data goes to the copied table.

The binding is used to receive the dump. It's a bit of an optimization. Remember, the dump responses are sent to all the clients. Whenever any client requests a dump, all the clients will get the response. A client finds that the incoming dump is destined for it by processing the "beginDump" label. If it contains this client's name, the dump is destined here, and the client reacts by pushing the appropriate binding onto the facet's FnReturn, and the data flows. The matching "endDump" label then pops the binding and the data stops flowing. The binding allows to avoid checking every rowop for whether it's supposed to be accepted and if yes then where exactly (rememeber, the same table may be dumped independently multiple times by multiple queries). Just check once at the start of the bundle and then let the data flow in bulk.

```
    # qdumpsub:
    #   * label where to send the dump request to
    #   * source output label, from which a subscription will be set up
    #     at the end of the dump
    #   * target label in the query that will be tied to the source label
    #   * binding to be used during the dump, which also directs the data
    #     to the same target label
    my $request = [ "qdumpsub", $lbrq, $lbsrc, $lbNext, $bindDump ];
    if ($front) {
      unshift @{$ctx->{requests}}, $request;
    } else {
      push @{$ctx->{requests}}, $request;
    }
    return $lbNext;
}
```

Finally, the created bits and pieces get packaged into a request and added to the list of requests in the query context. The last tricky part is that the request can be added at the back or the front of the list. The "normal" way is to add at the back, however the dimension tables for the joins have to be populated before the main data flow of the query starts. So for them the argument `$front` is set to 1, and they get added at the front.

Now jumping back to the writer thread logic, after it called `compileQuery()`, it starts the query execution by calling `&$runNextRequest()`. Which is a closure function defined inside the client writer thread function, and knows how to send the dump requests we've just seen created to the core logic.

```
# The requests from a query context get sent one by one, and
# after one is done, the next is sent until they all are done.
my $runNextRequest = sub { # ($ctx)
  my $ctx = shift;
  my $requests = $ctx->{requests};
```

```
  undef $ctx->{curRequest}; # clear the info of the previous request
  my $r = shift @$requests;
  if (!defined $r) {
    # all done, now just need to pump the data through
    printOrShut($app, $fragment, $sock,
      "querysub,$ctx->{qid},$ctx->{qname}\n");
    return;
  }
```

First it clears the information about the previous request, if any. This function will be called after each request, to send the next one, so on all its calls except the first one of a query it will have something to clear.

Then it checks if all the requests are already done. If so, it sends the query confirmation to the client socket and returns. The subscription part of the query will continue running on its own.

```
  $ctx->{curRequest} = $r; # remember until completed
  my $cmd = $$r[0];
  if ($cmd eq "qdumpsub") {
    # qdumpsub:
    #   * label where to send the dump request to
    #   * source output label, from which a subscription will be set up
    #       at the end of the dump
    #   * target label in the query that will be tied to the source label
    #   * binding to be used during the dump, which also directs the data
    #       to the same target label
    my $lbrq = $$r[1];
    # this code very specifically ignores %dump, doing its requests
    # independently for each query, and showing off another way to
    # do things
    # print "DBG next request {" . $ctx->{qname} . "} qdumpsub " . $lbrq->getName() .
"\n";
    $unit->makeHashCall($lbrq, "OP_INSERT",
      client => $fragment, id => $ctx->{qid}, name => $ctx->{qname}, cmd => $cmd);
  } else {
    printOrShut($app, $fragment, $sock,
      "error,", $ctx->{qid}, ",Internal error: unknown request '$cmd',internal,", $cmd,
"\n");
    $ctx->{requests} = [];
    undef $ctx->{curRequest};
    # and this will leave the query partially initialized,
    # but it should never happen
    return;
  }
};
```

The request data might vary by the command. If the command is "qdumpsub", the request gets translated to a rowop sent through the nexus to the dump request label in the core logic.

And a catch-all just in case if the query compiler ever decides to produce an invalid request.

Next goes the handling of the dump labels (again, this gets set up during the build of the client reader threads, and then the nature is left to run its course, reacting to the rowops as they come in). It gets set up in the writer thread function:

```
  $faOut->getLabel("beginDump")->makeChained("lbBeginDump", undef, sub {
    my $row = $_[1]->getRow();
    my ($client, $id, $name, $cmd) = $row->toArray();
    return unless ($client eq $fragment);
    if ($cmd eq "qdumpsub") {
      return unless(exists $queries{$id});
      my $ctx = $queries{$id};
      $fretOut->push($ctx->{curRequest}[4]); # the binding for the dump
```

```
    } else {
      return unless (exists $dumps{$name});
      printOrShut($app, $fragment, $sock,
        "startdump,$id,$name\n");
      $fretOut->push($bindDump);
    }
  });
```

As described before, it checks if this is the destination client, and if there is an active request with this id, then it pushes the appropriate binding.

The handling of the end of transaction also gets set up in the writer thread function:

```
  $faOut->getLabel("endDump")->makeChained("lbEndDump", undef, sub {
    my $row = $_[1]->getRow();
    my ($client, $id, $name, $cmd) = $row->toArray();
    return unless ($client eq $fragment);

    if ($cmd eq "qdumpsub") {
      return unless(exists $queries{$id});
      my $ctx = $queries{$id};
      $fretOut->pop($ctx->{curRequest}[4]); # the binding for the dump
      # and chain together all the following updates
      $ctx->{curRequest}[2]->makeChained(
        "qsub$id." . $ctx->{curRequest}[3]->getName(), undef,
        sub {
          # a cross-unit call
          $_[2]->call($_[3]->adopt($_[1]));
        },
        $ctx->{u}, $ctx->{curRequest}[3]
      );

      &$runNextRequest($ctx);
    } else {
      # .. skipped the handling of dump/dumpsub
    }
  });
```

Same as the "beginDump", it checks if this is the right client, and if it has an outstanding dump request, then pops the binding. After the dump is completed, the subscription has to be set up, so it sets up a label that forwards the normal output of this table to the label specified in the request. Since each query is defined in its own unit, this forwarding is done as a cross-unit call.

And then the next request of this query can be started until they all are done. And that's it, the end of the example.

Chapter 18. Performance

Obviously, the performance depends on the machine on which Triceps runs. One of the Perl unit tests, `t/Perf.t` allows you to measure the Triceps performace on your own machine. By default it runs only one thousand iterations, to be fast and not delay the run of the full tests suite. But the number can be increased by setting an environment variable, such as:

```
$ TRICEPS_PERF_COUNT=100000 perl t/Perf.t
Performance test, 100000 iterations, real time.
Empty Perl loop 0.006801 s, 14702927.05 per second.
Empty Perl function of 5 args 0.031918 s, 3133046.99 per second.
Empty Perl function of 10 args 0.027418 s, 3647284.30 per second.
Row creation from array and destruction 0.277232 s, 360708.19 per second.
Row creation from hash and destruction 0.498189 s, 200727.04 per second.
Rowop creation and destruction 0.155996 s, 641043.69 per second.
Calling a dummy label 0.098480 s, 1015437.21 per second.
Calling a chained dummy label 0.110546 s, 904601.83 per second.
  Pure chained call 0.012066 s, 8287664.25 per second.
Calling a Perl label 0.512559 s, 195099.61 per second.
Row handle creation and destruction 0.195778 s, 510781.66 per second.
Repeated table insert (single hashed idx, direct) 0.772934 s, 129377.12 per second.
Repeated table insert (single hashed idx, direct & Perl construct) 1.109781 s, 90107.89
 per second.
  RowHandle creation overhead in Perl 0.336847 s, 296871.05 per second.
Repeated table insert (single sorted idx, direct) 2.122350 s, 47117.59 per second.
Repeated table insert (single hashed idx, call) 0.867846 s, 115227.88 per second.
Table insert makeRowArray (single hashed idx, direct) 1.224588 s, 81660.14 per second.
  Excluding makeRowArray 0.947355 s, 105557.02 per second.
Table insert makeRowArray (double hashed idx, direct) 1.443053 s, 69297.51 per second.
  Excluding makeRowArray 1.165821 s, 85776.47 per second.
  Overhead of second index 0.218466 s, 457738.04 per second.
Table insert makeRowArray (single sorted idx, direct) 29.880962 s, 3346.61 per second.
  Excluding makeRowArray 29.603730 s, 3377.95 per second.
Table lookup (single hashed idx) 0.287407 s, 347938.44 per second.
Table lookup (single sorted idx) 9.160540 s, 10916.39 per second.
Lookup join (single hashed idx) 3.940388 s, 25378.21 per second.
Nexus pass (1 row/flush) 0.578993 s, 172713.78 per second.
Nexus pass (10 rows/flush) 2.266709 s, 441168.31 per row per second.
  Overhead of each row 0.187524 s, 533265.07 per second.
  Overhead of flush 0.391469 s, 255448.33 per second.
```

An important caveat, the test is of the Perl interface, so it includes all the overhead of constructing the Perl objects. I've tried to structure it so that some of the underlying performance can be deduced, but it's still approximate. I haven't done the performance testing of just the underlying C++ implementation yet, it will be better.

The computations are done with the real elapsed time, so if the machine is not idle, the time of the other processes will get still counted against the tests, and the results will show slower than they really are.

The numbers above are from my old-ish laptop (dual-CPU Intel Core2 T9900 3GHz). The time in seconds printed for each value is for the whole test loop of 100K iterations. The "per second" number shows the opposite, how many loop iterations were done per second.

I've had an opportunity to run the performance tests on a few more laptops. Al of the same Core2 generation, but with the different CPU frequencies. The 2GHz version showed expectedly an about 30% lower performance proportionally. A 2.2GHz CPU also showed an about proportional change, except for the inter-thread communication through the nexus, it went down more than proportional. I'm not sure, what was up with the 2.2GHz CPU, maybe the timing worked out just wrong to add more overhead.

Here is the row-by-row description of the measurements:

```
Performance test, 100000 iterations, real time.
```

The first thing it prints is the iteration count, to set the expectations for the run length and precision. The shorter runs tend to include more randomness.

```
Empty Perl loop 0.006801 s, 14702927.05 per second.
Empty Perl function of 5 args 0.031918 s, 3133046.99 per second.
Empty Perl function of 10 args 0.027418 s, 3647284.30 per second.
```

A calibration to see, how much overhead is added by the execution of the loop itself. As it turns out, not much. The Perl function calls add more but still don't dominate the numbers.

```
Row creation from array and destruction 0.277232 s, 360708.19 per second.
```

The `makeRowArray()` for a row of 5 fields. Each created row gets destroyed before the next one gets created.

```
Row creation from hash and destruction 0.498189 s, 200727.04 per second.
```

The `makeRowHash()` for a row of 5 fields.

```
Rowop creation and destruction 0.155996 s, 641043.69 per second.
```

The `makeRowop()` from an existing row. Same thing, each rowop gets destroyed before constructing the next one.

```
Calling a dummy label 0.098480 s, 1015437.21 per second.
```

Repeated calls of a dummy label with the same rowop object. This can be thought of as the basic overhead of a C++ label call, with a limited amount (25-30% or so) of the Perl overhead mixed in.

```
Calling a chained dummy label 0.110546 s, 904601.83 per second.
  Pure chained call 0.012066 s, 8287664.25 per second.
```

Repeated calls of a dummy label that has another dummy label chained to it. The "pure" part is the difference from the previous case that gets added by appending another chained dummy label.

```
Calling a Perl label 0.512559 s, 195099.61 per second.
```

Repeated calls of a Perl label with the same rowop object. The Perl label has an empty `sub` but that empty `sub` still gets executed, along with all the support functionality.

```
Row handle creation and destruction 0.195778 s, 510781.66 per second.
```

The creation of a table's row handle from a single row, including the creation of the Perl wrapper for the row handle object.

```
Repeated table insert (single hashed idx, direct) 0.772934 s, 129377.12 per second.
```

Insert of the same row into a table. Since the row is the same, it keeps replacing the previous one, and the table size stays at 1 row. The code is $tSingleHashed->insert($row1). Even though the row is the same, a new row handle gets constructed for it every time by the table. "Single hashed idx" means that the table has a single Hashed index, on an `int32` field. "Direct" means the direct `insert()` call, as opposed to using the table's input label.

```
Repeated table insert (single hashed idx, direct & Perl construct) 1.109781 s, 90107.89
 per second.
  RowHandle creation overhead in Perl 0.336847 s, 296871.05 per second.
```

The same, only the row handles are constructed in Perl before inserting them: `$tSingle-Hashed->insert($tSingleHashed->makeRowHandle($row1))`. And the second line shows that the overhead of wrapping the row handles for Perl is pretty noticeable (it's the difference from the previous test case).

```
Repeated table insert (single sorted idx, direct) 2.122350 s, 47117.59 per second.
```

The same thing, only for a table that uses a Sorted index that executes a Perl comparison on the same `int32` field. As you can see, it gets almost 3 times slower.

```
Repeated table insert (single hashed idx, call) 0.867846 s, 115227.88 per second.
```

The same thing, again the table with a single Hashed index, but this time by sending the rowops to its input label.

```
Table insert makeRowArray (single hashed idx, direct) 1.224588 s, 81660.14 per second.
  Excluding makeRowArray 0.947355 s, 105557.02 per second.
```

Now the different rows get inserted into the table, each row having a different key. At the end of this test the table contains 100K rows (or however many were requested by the environment variable). Naturally, this is slower than the repeated insertions of the same row, since the tree of the table's index becomes deeper and requires more comparisons and rebalancing. This performance will be lower in the tests with more rows, since the index will become deeper and will create more overhead. Since the rows are all different, they are created on the fly, so this row creation overhead needs to be excluded to get the actual Table's performance.

```
Table insert makeRowArray (double hashed idx, direct) 1.443053 s, 69297.51 per second.
  Excluding makeRowArray 1.165821 s, 85776.47 per second.
  Overhead of second index 0.218466 s, 457738.04 per second.
```

Similar to previous one but on a table that has two Hashed indexes (both on the same $int32$ field). The details here also compute the overhead contributed by the second index.

```
Table insert makeRowArray (single sorted idx, direct) 29.880962 s, 3346.61 per second.
  Excluding makeRowArray 29.603730 s, 3377.95 per second.
```

Similar but for a table with a Sorted index with a Perl expression. As you can see, with the makeRowArray overhead excluded it's about 30 times slower (and it gets even worse for the larger row sets).

```
Table lookup (single hashed idx) 0.287407 s, 347938.44 per second.
```

Finding a row in a table of 100K (or however many iterations requested) rows by a hashed index.

```
Table lookup (single sorted idx) 9.160540 s, 10916.39 per second.
```

Finding a row in a table of 100K (or however many iterations requested) rows by a Perl sorted index. Just like row insertion, it's 30-odd times slower than a hashed index.

```
Lookup join (single hashed idx) 3.940388 s, 25378.21 per second.
```

Joins in the LookupJoin template (and the performance of the JoinTwo template will be the same, since it consists of two LookupJoins). It performs essentially the same table lookup, only with the added overhead of constructing the new combined rows, and the general overhead of the extra Perl label calls.

```
Nexus pass (1 row/flush) 0.578993 s, 172713.78 per second.
Nexus pass (10 rows/flush) 2.266709 s, 441168.31 per row per second.
  Overhead of each row 0.187524 s, 533265.07 per second.
  Overhead of flush 0.391469 s, 255448.33 per second.
```

The passing of the rows between threads through a Nexus. The time also includes the draining and stopping of the app, so it's a little pessimistic, and gets more pessimistic for the short runs. Both cases pass 100K of transactions, so the second one passes 10 times more rows. The difference in performance of different transaction sizes allows to compute the overhead of the transaction passing (i.e. flushes) versus the overhead of adding extra rows to transactions.

Chapter 19. 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. 27) .

RowType in Section 5.2: "Row types" (p. 27) .

Row in Section 5.4: "Rows" (p. 30) .

Label in Chapter 6: "*Labels and Row Operations*" (p. 33) .

Rowop in Section 6.4: "Row operations" (p. 36) .

The method `Triceps::now()` stands kind of alone. It's similar to Perl `time()` but returns the current time as a floating point number, with the fractional seconds:

```
$time = &Triceps:now();
```

This time format is used by a few Triceps methods, mostly in the multithreading support.

19.1. Unit and FrameMark reference

The Unit class represents an execution unit and keeps the state of the Triceps execution for one thread. Each thread running Triceps must have its own execution unit.

It's perfectly possible to have multiple execution units in the same thread. This is typically done when there is some permanent model plus some small intermittent sub-models created on demand to handle the user requests. These small sub-models would be created in the separate units, to be destroyed when their work is done.

A unit is created as:

```
$myUnit = Triceps::Unit->new($name);
```

The `$name` argument will be used in the error messages, making easier to find, which exact part of the model is having troubles. By convention the name should be the same as the name of the unit variable ("myUnit" in this case). When a Unit is created as a part of Triead (a Triceps thread object), it will be given the same name as the Triead.

The name can be read back:

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

Also, as usual, the variable `$myUnit` 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 (discussed in Section 8.2: "Clearing of the labels" (p. 78) and Section 6.2: "Label construction" (p. 34)), 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 that can be used to create the objects that don't process any data (for which just the fact of a rowop passing through is important). The creation of an empty row type in each unit instead of having a single global one is purely for the conveniece of memory management in the threads, they are all equivalent. You could also create your own empty row type.

The rowops are enqueued with the calls:

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

"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".

Call() executes the rowop immediately, fork() puts it onto the current stack frame to be executed after the current label returns, and schedule() puts the rowop onto the outermost frame, to be executed when the model becomes idle.

Calling these functions with multiple arguments produces the same result as doing multiple calls with one argument at a time. The rowop and tray arguments may be mixed arbitrarily.

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

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

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"

- &Triceps::EM_IGNORE or "EM_IGNORE"

The EM_IGNORE is a "no-op" among the enqueueing methods. It means that the rowop will be simply ignored and not executed at all. It is very rarely used in practice.

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

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

And as usual, if the value can not be translated, the functions with Safe return undef, the functions without it confess.

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

```
$mark = Triceps::FrameMark->new($name);
```

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_or_trays);
```

`setMark` must be called in the first label of the loop, normally before doing anything else. `loopAt()` is called in any label inside the loop that wants to send a rowop back to the start of the loop. The rowop or tray arguments of the `loopAt()` are the same as for the other enqueueing functions.

The examples of the `loopAt()` operation are presented in Section 7.7: "Topological loops" (p. 46) , and the details of its internal works are in Section 7.12: "The gritty details of Triceps loop scheduling" (p. 72) .

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.

The Unit also serves as a factory for creation of the objects associated with it:

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

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

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

$table = $unit->makeTable($tableType, "name");

$tray = $unit->makeTray(@rowops);
```

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. The references are at the C++ level, not Perl level.

With the tables, the references 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 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. 77) .

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

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

($label, $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, debug, and read later than a topological one.

`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()`, but as-is it's slightly more efficient.

They both return the same results, a pair:

* The label that will be the real start of the loop, where you send a rowop to both initiate the loop and to do the next iteration of the loop with `loopAt`.

* The frame mark that you use in `loopAt()`. You don't need to set the frame mark, it will be set for you in the wrapper logic.

The name argument will become the name of the created label. The FrameMark object also has a name, useful for diagnostics, that gets created by adding a suffix to the argument: "*name*.mark".

`makeLoopAround()` takes the row type for its wrpper label from `$labelFirst`.

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

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

Check whether all the frames are empty.

```
$res = $unit->isFrameEmpty();
```

Check whether the current inner frame is empty. This method is useful if you run multiple units in the same thread, with some potentially complicated cross-unit scheduling. It's what `TrieadOwner::nextXtray()` does with a multi-unit Triead, repeatedly calling `drainFrame()` for all the units that are found not empty. In this situation the simple `empty()` cannot be used because the current inner frame might not be the outer frame, and draining the inner frame can be repeated forever while the outer frame will still contain rowops. The more precise check of `isFrameEmpty()` prevents the possibility of such endless loops.

```
$res = $unit->isInOuterFrame();
```

Check whether the unit's current inner frame is the same as its outer frame, which means that the unit is not in the middle of a call.

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 inner 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.

The recursive calls may be enabled per-unit with the methods:

```
$unit->setMaxStackDepth($n);
$unit->setMaxRecursionDepth($n);
```

`setMaxStackDepth()` sets the limit on the total depth of the unit's call stack. That's the maximal length of the call chain, whether it goes straight or in loops.

`setMaxRecursionDepth()` sets the limit on the number of times each particular label may appear on the call stack. So if you have a recursive code fragment (a simple-minded loop or a recursive streaming function), this is the limit on its recursive reentrances.

You can change them at any time, even when the unit is running (but they will be enforced only on the next attempt to execute a rowop). Both these limits accept the 0 and negative values to mean "unlimited". The default is: unlimited stack depth, recursion depth of 1.

You can read the current limits with:

```
$n = $unit->maxStackDepth();
$n = $unit->maxRecursionDepth();
```

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.

```
$unit->clearLabels();
```

Clear all the labels in the unit, then drop the references from unit to them. Normally should be called only when the unit is about to be destroyed, and is called automatically when the unit gets destroyed. However in case of cyclic references to the unit, it may need to be called manually or through a UnitClearingTrigger object to break these cycles. For more information, see Section 8.1: "Reference cycles" (p. 77) .

In case if any errors are found during the clearing of the labels (such as a bug in the user code in the label clearing routine), they get printed to stderr. The reason for that is that the clearing is normally called from the object destructors, and there just isn't any proper way to report an error from a destructor. The only thing that can be done is printing them.

```
$trigger = $unit->makeClearingTrigger();
```

Create a unit clearing trigger object. When this object gets destroyed (at exiting the block, or when its encompassing object gets destroyed) it will cause the clearing of the unit.

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

Set and read back a tracer object that will be called before and after each rowop executes in the unit. An `undef` means "no tracer". See more information in Section 7.10: "Tracing the execution" (p. 63) .

```
$unit->callBound($rowop_or_tray, $fnreturn => $fnbinding, ...);
$unit->callBound([@rowops], $fnreturn => $fnbinding, ...);
```

An encapsulation of a streaming function call. The first argument is a rowop or a tray or a reference to an array of rowops (but the trays are not allowed in the array). The rest are the pairs of FnReturns and FnBindings. The bindings are pushed onto the FnReturns, then the rowops are called, then the bindings are popped. It replaces a whole block that would contain an AutoFnBind (described in Section 19.17: "AutoFnBind reference" (p. 394)) and the call:

```
{
  my $ab = Triceps::AutoFnBind->new(
    $fnreturn => $fnbinding, ...
  );
  $unit->call($rowop_or_tray);
}
```

Only `callBound()` does its work in C++, so it's more efficient than a Perl block, and it's shorter to write too.

19.2. 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);
```

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);
```

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)
;
```

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();
```

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");
$indexType = $tt->findSubIndexSafe("indexName");
```

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. The `Safe` version returns `undef` if the index is not found, instead of confessing.

```
$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.

```
@idxPath = $tt->findIndexPathForKeys(@keyFields);
```

Finds an index path that matches the set of key fields. It returns the array that represents the path. And then having the path you can find the index type as such. The index type and all the types in the path have to be of the Hashed variety (returning their set of keys with the method `getKey()`). If the correct index cannot be found, an empty array is returned. If you specify the fields that aren't present in the row type in the first place, this is simply treated the same as being unable to find an index for these fields. If more that one index would match, the first one found in the direct order of the index tree walk is returned.

```
$indexType = $tt-findSubIndexById($indexTypeId);
```

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.

```
$newtt = $tt->copy();
```

Copy the table type, along with copying all the index types in it, since each table type must have its own instances of the index types. The copied table type is always uninitialized and thus can be further extended by defining more indexes and aggregators.

In case if the table type collected errors, the errors aren't copied, and in general you should not copy such a table type. The errors will be detected again when you try to initialize the copy.

```
$newtt = $tt->copyFundamental();
$newtt = $tt->copyFundamental(@idx_type_paths);
```

```
$newtt = $tt->copyFundamental("NO_FIRST_LEAF", @idx_type_paths);
```

Copy only the fundamental data organization of the table type: the row type and the primary index (the whole path to the first leaf index type), and leave alone the rest. All the aggregators on all the indexes, even on the primary one, are not included in the copy. This is a convenient way to create another table, usually in a different thread, that would add its own secondary indexes and aggregators. See Section 16.4: "Object passing between threads" (p. 297) for more detail.

The optional index type paths can be used to include the secondary indexes into the copy. Each path is a reference to an array of index names, such as ["byDate", "byAddress", "fifo"]. The path must lead all the way to a leaf index. A special path element "+" can be used last with the meaning "the first leaf index from this point". Multiple paths may overlap, their overlapping parts will be copied only once. A special string "NO_FIRST_LEAF" used as the first argument excludes the first leaf index from the copy, then only the explicitly specified index paths will be copied.

19.3. 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, ...);
```

Creates a Hashed index type. The only available and mandatory option is:

```
key => [ @fields ]
```
    The argument is a reference to an array of strings that specify the names of the key fields (key => ["f1", "f2"]).

```
$it = Triceps::IndexType->newFifo($optionName => $optionValue, ...);
```

Creates a FIFO index type. The options are:

```
limit => $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. Don't try to set it to negative values, they will be treated as unsigned, and thus become some very large positive ones. Optional. Default: 0.

`jumping => 0/1`
> Determines the variation of the replacement policy in effect. If set to 0, 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. Optional. Default: 0.

`reverse => 0/1`
> 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! Optional. Default: 0.

```
$it = Triceps::IndexType->newPerlSorted($sortName, $initFunc,
  $compareFunc, @args...);
```

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). A source code string may be used instead of the function reference, see Section 4.4: "Code references and snippets" (p. 21) .

`$compareFunc`
> a function reference to the fixed comparison function, if preferred (or use `undef` if it will be generated dynamically by the init function). A source code string may be used instead of the function reference, see Section 4.4: "Code references and snippets" (p. 21) .

`@args`
> optional extra arguments for the initialization and/or comparison function.

See the details in Section 9.8: "Sorted index" (p. 95) .

The handling of the fatal errors (as in `die()`) in the initialization and especially comparison functions is an interesting subject. The errors propagate properly through the table, and the table operations confess with the Perl handler's error message. But since an error in the comparison function means that things are going very, very wrong, after that the table becomes inoperative and will die on all the subsequent operations as well. You need to be very careful in writing these functions.

```
$it = Triceps::SimpleOrderedIndex->new($fieldName => $order, ...);
```

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);
```

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");
$itSub = $it->findSubIndexSafe("indexName");
$itSub = $it->findSubIndexById($indexTypeId);

@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);
```

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 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.

```
$itCopy = $it->flatCopy();
```

Creates a copy of the index type object itself, without any connected object hierarchies such as the nested indexes or aggregators.

```
$tabType = $it->getTabtype();
$tabType = $it->getTabtypeSafe();
```

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, getTabtype() would confess while getTabtypeSafe() would return an undef. Which method to use, depends on the circumstances: if this situation is valid and you're ready to check for it and handle it, use getTabtypeSafe(), otherwise use getTabtype().

```
$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...);
```

A special method that works only on the PerlSorted index types. Sets an auto-generated 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 confesses. A source code string may be used instead of the function reference, see Section 4.4: "Code references and snippets" (p. 21) .

19.4. 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);
```

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.

`$initFunc` and `$handlerFunc` provide either references or source code strings for the init and handler functions (as usual, if the format is the source code, the `sub { ... }` wrapper will be added implicitly, see Section 4.4: "Code references and snippets" (p. 21)). 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. 155) , Section 11.6: "Additive aggregation" (p. 157) and Section 11.7: "Computation function arguments" (p. 161) . The optional `@args` are passed to both the init and handler functions.

If any fatal errors (as in `die()`) occur in the aggregator functions, they propagate properly through the table, and the table operations confess with the Perl handler's error message. But since an error in the aggregator function means that things are going very, very wrong, after that the table becomes inoperative and will die on all the subsequent operations as well. You need to be very careful in writing these functions.

```
$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.

```
$rt = $at->getRowType();
```

Get back the result row type.

19.5. SimpleAggregator reference

SimpleAggregator provides an easier way to describe aggregations with the SQL-like aggregation functions. It also supports the user-defined aggregation functions.

The table types with SimpleAggregator cannot be currently exported between threads through a nexus, see Section 16.4: "Object passing between threads" (p. 297) for more detail.

```
$tabType = Triceps::SimpleAggregator::make($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 => $name`
> The aggregator type name.

`tabType => $tabType`
> Table type to put the aggregator on. It must be un-initialized yet.

`idxPath => [ @path ]`
> 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 => [ [fieldName => $type, $funcName, $argFunc], ... ]`
> A reference to an array defining the result of the aggregation. It consists of the repeating groups of four elements:
>
> `fieldName => $type, $funcName, $argFunc,`
>
> Here `$type` is the field type name, `$funcName` is the name of the aggregtion function (case-sensitive, see the list in Section 11.9: "SimpleAggregator" (p. 167)), and `$argFunc` is the code reference that computes the argument of the aggregation function. It's 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 => \$rt`
> 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 constructed, its result row type may also be found with `$table->getAggregatorLabel("aggName")->getRowType()`.

`saveInitTo => \$code`
> Optional. A reference to a scalar where to save the auto-generated source code of the initialization function for diagnostics.

`saveComputeTo => \$code`
> Optional. A reference to a scalar where to save the auto-generated source code of the handler function for diagnostics.

`functions => { %definitions }`
> Optional. The additional user-defined aggregation functions. See the description of their structure in Section 11.9: "SimpleAggregator" (p. 167) .

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).

19.6. Table reference

The Table provides the structured data storage in Triceps.

```
$t = $unit->makeTable($tabType, "tableName");
```

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 table name is used for the error messages and as a base for the names of the table labels.

A word needs to be said about the handling of the fatal errors (as in `die()`) in the custom Perl functions provided in the user elements of the table, such as the PerlSortedIndex and aggregators. The errors propagate properly through the table, and the table operations confess with the Perl handler's error message. But since an error in these functions means that things are going very, very wrong, after that the table becomes inoperative and will die on all the subsequent operations as well. This is known as a "sticky error". You need to be very careful in writing these functions.

```
$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->getDumpLabel();
$lb = $t->getAggregatorLabel("aggName");
```

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);
$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);
```

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. Returns 1 on success, 0 if the insert can not be done (the row handle is already in the table or NULL).

```
$result = $t->remove($rh);
```

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 result is 1 on success (even if the row handle was not in the table).

```
$result= $t->deleteRow($row);
```

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. 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 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).

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

```
$table->clear($limit);
```

Deletes the rows from the table. If `$limit` is absent or 0, the whole table gets cleared. If it's greater than 0, no more than this number of rows will be deleted. A negative limit is an error. The deletion happens in the usual order of the first leaf index, and the rowops are sent to the table's output label as usual. It's really the same thing as running a loop over all the row handles and removing them, only with a C++ imnplementation it's more efficient than in Perl. There is no return value, the errors cause a confession.

```
$fret = $table->fnReturn();
```

Returns an FnReturn object connected to the output of this table. The FnReturn contains the labels "pre", "out", "dump" (used with `dumpAll` methods) and the named labels for all aggregators. The FnReturn object is created on the first call of this method and is kept in the table. All the following calls return the same object. This has some interesting consequences for the "pre" label: the rowop for the "pre" label doesn't get created at all if there is nothing chained from that label. But when the FnReturn gets created, one of its labels gets chained from the "pre" label. Which means that once you call `$table->fnReturn()` for the first time, you will see that table's "pre" label called in all the traces. It's not a huge extra overhead, but still something to keep in mind and not be surprised when calling `fnReturn()` changes all your traces. See more detail in Section 15.7: "Streaming functions and tables" (p. 264) .

```
$table->dumpAll();
$table->dumpAll($opcode);
```

Sends all the table's rows in their default order to the label "dump". The opcode for the created rowops is OP_INSERT by default, or can be specified explicitly as an argument. The typical usage is to push a binding on the table's FnReturn to process these dumped rowops.

```
$table->dumpAllIdx($indexType);
$table->dumpAllIdx($indexType, $opcode);
```

Similar to `dumpAll()`, only allows to send the rows in an order specified by another index type. As usual, the index type must belong to the exact type of this table.

19.7. 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);
```

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. If the row handle is NULL, it will quietly return 0.

```
$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();
$row = $rh->getRowSafe();
```

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, in this situation getRow() will confess and getRowSafe() will return undef.

```
$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 are essentially syntactic sugar over the table methods.

19.8. 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->last();
```

Returns the last row handle of the group. In case of an empty group it would return 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 19.6: "Table reference" (p. 370), 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()`.

19.9. 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.

A special case is the passing through of the options: you can accept the arbitrary options, typically if your function is a wrapper to another function, and you just want to process a few options and let the others through. The `Triead::start()` is a good example, passing the options through to the main function of the thread.

The acceptance of the arbitrary options is specified by using an option named "*" in the `Opt::parse()` arguments. For example:

```
  &Triceps::Opt::parse($myname, $opts, {
    app => [ undef, \&Triceps::Opt::ck_mandatory ],
    thread => [ undef, \&Triceps::Opt::ck_mandatory ],
    fragment => [ "", undef ],
    main => [ undef, sub { &Triceps::Opt::ck_ref(@_, "CODE") } ],
    '*' => [],
  }, @_);
```

The specification array for "*" is empty. The unknown options will be collected in the array referred to from `$opts->{'*'}`, that is `@{$opts->{'*'}}`.

From there on the wrapper has the choice of either passing through all the options to the wrapped function, using `@_`, or explicitly specifying a few options and passing through the rest from `@{$opts->{'*'}}`.

There is also the third possibility: filter out only some of the incoming options. This can be done with `Opt::drop()`. It lets through only the options that are not present in the description:

```
@filteredOpts = &Triceps::Opt::drop(\%optdescr, @opts);
@filteredOpts = &Triceps::Opt::dropExcept(\%optdescr, @opts);
```

The `Opt::drop()` takes the specification of the options to drop as a hash reference, the same as `Opt::parse()`. The values in the hash are not important in this case, only the keys are used. But it's simpler to store the same specification of the options and reuse it for both `parse()` and `drop()` than to write it twice.

There is also an opposite function, `Opt::dropExcept()`. It passes through only the listed options and drops the rest. It can come handy if your wrapper wants to pass different subsets of its incoming options to multiple functions.

The functions `drop()` and `dropExcept()` can really be used on any name-value arrays, not just the options as such. And the same goes for the `Fields::filter()` and friends. So you can use them interchangeably: you can use `Opt::drop()` on the row type specifications and `Fields::filter()` on the options if you feel that it makes your code simpler.

For an example of `drop()`, `Triead::startHere()` works like this:

```
# The options for start(). Keeping them in a variable allows the individual
# thread main functions to copy and reuse their definition instead of
# reinventing it.
our @startOpts = (
  app => [ undef, \&Triceps::Opt::ck_mandatory ],
  thread => [ undef, \&Triceps::Opt::ck_mandatory ],
  fragment => [ "", undef ],
  main => [ undef, sub { &Triceps::Opt::ck_ref(@_, "CODE") } ],
);

sub startHere # (@opts)
{
  my $myname = "Triceps::Triead::start";
  my $opts = {};
  my @myOpts = ( # options that don't propagate through
    harvest => [ 1, undef ],
    makeApp => [ 1, undef ],
  );

  &Triceps::Opt::parse($myname, $opts, {
    @startOpts,
    @myOpts,
    '*' => [],
  }, @_);

  my @args = &Triceps::Opt::drop({
    @myOpts
  }, \@_);
  @_ = (); # workaround for threads leaking objects

  # no need to declare the Triead, since all the code executes synchronously anyway
  my $app;
  if ($opts->{makeApp}) {
    $app = &Triceps::App::make($opts->{app});
  } else {
    $app = &Triceps::App::resolve($opts->{app});
  }
  my $owner = Triceps::TrieadOwner->new(undef, undef, $app, $opts->{thread}, $opts-
>{fragment});
  push(@args, "owner", $owner);
  eval { &{$opts->{main}}(@args) };
  $owner->abort($@) if ($@);
  # In case if the thread just wrote some rows outside of nextXtray()
  # and exited, flush to get the rows through. Otherwise things might
```

```
    # get stuck in a somewhat surprising way.
    eval { $owner->flushWriters(); };
    $owner->markDead();
    if ($opts->{harvest}) {
      $app->harvester();
    }
}
```

The `@startOpts` are both used by the `startHere()` and passed through. The `@myOpts` are only used in `startHere()` and do not pass through. And the rest of the options pass through without being used in `startHere()`. So the options from `@myOpts` get dropped from `@_`, and the result goes to the main thread.

The Opts package also provides the helper methods for processing the sets of options.

```
&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).

19.10. 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. 136) .

The option-processing functions `Opt::drop()` and `Opt::dropExcept()` can also be used to filter the field lists.

```
@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 => [ @rts ]`
    Reference to an array of row types for the original rows.

`filterPairs => [ @pairs ]`
    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 => \$code`
    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.

19.11. 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 => $unit`
    Scheduling unit object where this template belongs. May be skipped if "leftFromLabel" is used.

`name => $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 => $rt`
    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 => $label`
    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`
    Table object where to do the look-ups.

`rightIdxPath => [ @path ]`
> 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: automatically found by the set of key fields. Whether explicitly specified or automatically found, the index must be available in the table.

`leftFields => [ @patterns ]`
> 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 => [ @patterns ]`
> 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 => 0/1`
> 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 => 0/1`
> 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.

`fieldsDropRightKey => 0/1`
> Flag: remove the key fields on the right side from the result. Convenient to avoid their duplication, especially if the key fields are named the same on both sides. Optional. Default: 0.

`by => [ @fields ]`
> 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 => [ @patterns ]`
> 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 => 0/1`
> Flag: 1 for left outer join, 0 for inner join. Optional. Default: 1.

`limitOne => 0/1`
> 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 => 0/1`
> 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 => 0/1`
> 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 => $func
```
Reference to a function that would compute the group size for this side's table. Optional, used only when "opposite-Outer" 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 => \$code
```
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.

This method has become largely obsolete since addition of fnReturn() as described in Section 15.8: "Streaming functions and template results" (p. 266) but is still present in case if it comes useful.

```
$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.

```
$fret = $joiner->fnReturn();
```

Returns an FnReturn object connected to the output of this joiner. The FnReturn contains one label "out". The FnReturn is created on the first call of this method and is kept in the joiner object. All the following calls return the same object. See more detail in Section 15.8: "Streaming functions and template results" (p. 266) .

```
$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.

19.12. 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
```
    Name of this object. Will be used to create the names of internal objects.

```
leftTable => $table
```
    Table object to join, for the left side. Both tables must be of the same unit.

```
rightTable => $table
```
    Table object to join, for the right side. Both tables must be of the same unit.

```
leftFromLabel => $label
```
    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 => $label
```
    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 => [ @path ]
```
    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 => [ @path ]
```
    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 => [ @patterns ]`
　　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 => [ @patterns ]`
　　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 => 0/1`
　　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 => $enum`
　　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 => [ @fields ]`
　　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 => [ @patterns ]`
　　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 => $enum
```
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 => \$code
```
Reference to a scalar where to save a copy of the joiner function source code for the left side. Optional.

```
rightSaveJoinerTo => \$code
```
Reference to a scalar where to save a copy of the joiner function source code for the right side. Optional.

```
overrideSimpleMinded => 0/1
```
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 => 0/1
```
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").

```
$fret = $joiner->fnReturn();
```

Returns an FnReturn object connected to the output of this joiner. The FnReturn contains one label "out". The FnReturn is created on the first call of this method and is kept in the joiner object. All the following calls return the same object. See more detail in Section 15.8: "Streaming functions and template results" (p. 266) .

```
$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.

19.13. 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. 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).

```
$fret = $collapse->fnReturn();
```

Returns an FnReturn object connected to the output of this collapser. The FnReturn contains a label for each data set, named accordingly. The order of the labels matches the specified order of data sets. The FnReturn is created on the first call of this method and is kept in the collapser object. All the following calls return the same object. See more detail in Section 15.8: "Streaming functions and template results" (p. 266) .

19.14. Braced reference

The package Braced is designed to parse the Tcl-like nested lists where the elements are separated by whitespace, and braces are used to enquote the elements with spaces in them. These lists are used to write the pipelines that form the Tql queries. For example:

```
{read table tWindow} {project fields {symbol price}} {print tokenized 0}
```

These lists can then be parsed into elements, and the elements might be also lists that could be parsed into elements and so on. The spaces between the braces are optional, braces also serve as separators. For example, the following lines are equivalent:

```
a b c
{a} {b} {c}
{a}{b}{c}
{a}b{c}
```

In case if a brace character needs to be included into one of the strings, they can be escaped by backslashes, for example:

```
{a\{} b\}c
```

Any other Perl backslash escapes, such as "\n" or "\x20", work too. The quote characters have no special meaning, they don't need to be escaped and they don't group the words. For example, the following two are equivalent:

```
"a b c"
{"a} {b} {c"}
```

Escaping the spaces ("\ ") provides another way to combine the words into one element. The following two are equivalent:

```
{a b c}
a\ b\ c
```

There is no need for the nested escaping. The characters need to be escaped only once, and then the resulting strings can be wrapped into any number of brace levels.

All the methods in this module are static, there are no objects.

```
$string = $data;
@elements = Triceps::Braced::raw_split_braced($string)
confess "Unbalanced braces around '$string'" if $string;
```

Split the string into the braced elements. If any of the elements were enclosed into their own braces, these braces are left in place, the element string will still contain them. For example, "a {b} {c d}" will be split into "a", "{b}", "{c d}". No unescaping is done, the escaped characters are passed through as-is. This method of splitting is rarely used, it's present as a baseline.

The original string argument will be fully consumed. If anything is left unconsumed, this is an indication of a syntax error, with unbalanced braces. The argument may not be a constant because it gets modified.

```
$string = $data;
@elements = Triceps::Braced::split_braced($string)
confess "Unbalanced braces around '$string'" if $string;
```

Split the string into the braced elements. If any of the elements were enclosed into their own braces, these braces will be removed from the results. For example, "a {b} {c d}" will be split into "a", "b", "c d". No unescaping is done, the escaped characters are passed through as-is. This is the normal method of splitting, it allows the elements to be split further recursively.

The original string argument will be fully consumed. If anything is left unconsumed, this is an indication of a syntax error, with unbalanced braces. The argument may not be a constant because it gets modified.

```
$result = Triceps::Braced::bunescape($string);
```

Un-escape a string by processing all the escape characters in it. This step is normally done last, after all the splitting is done. The result will become unsuitable for the future splitting because the escaped characters will lose their special meaning. If any literal braces are present in the argument, they will pass through to the result as literals. For example, "{a \{b }" will become "{a {b }".

```
@results = Triceps::Braced::bunescape_all(@strings);
```

Perform the un-escaping on a whole array of strings. The result array will contain the same number of elements as the argument.

```
$ref_results = Triceps::Braced::split_braced_final($string);
confess "Unbalanced braces around '$string'" if $string;
```

The combined functionality of splitting a string and un-escaping the result elements. That's why it's final: no further splits must be done after un-escaping. **The return value is different from the other split methods.** It is a reference to the array of result strings. The difference has been introduced to propagate the undef from the argument to the result: if the argument string is undef, the result will be also undef, **not** a reference to an empty array. The string gets consumed in the same way as for the other split methods, and anything left in it indicates an unbalanced brace.

19.15. FnReturn reference

The FnReturn represents the return value of a streaming function. The return value consists of a stream of rowops, and gets processed by sending them to the labels through a binding.

```
$fret = Triceps::FnReturn->new($optName => $optValue, ...);
```

Construct an FnReturn object. The options are:

```
name => $name
```
    Name of this object. Will be used to create the names of the labels in it.

```
unit => $unit
```
    The unit where this object belongs.

```
labels => [ @definitions ]
```
    Definition of the labels in the FnReturn, where the results of the streaming function will be sent. The full names of these labels will be "return_name.label_name". The label names within a return must be unique. The value for this option is an array reference, with the labels defined as name-value pairs in the array, in one of two forms:

```
labels => [
  $name1 => $rowType1,
  $name2 => $fromLabel2,
  ...
]
```

    If the second element in the pair is a row type, a label of that row type will be created in the FnReturn.

    If the second element in the pair is a label, its row type will be used to create a label in FnReturn and that new label will also be automatically chained off the specified one. This is convenient if you already have the logic of the function defined and just want to forward the result data from an existing label into the FnReturn. The chaining is normally done with chainFront(), unless the option chainFront => 0 tells otherwise. The front chaining is convenient if you want to pass both the original request and the result into the return. Usually you would define the result computation and then define the return. With the chaining at the back, this would lead to the computation chained off the input label first and the return going after it. This would lead to the result coming out before the argument, and special contortions would be needed to avoid it. With chaining at the front, the return will go in the chain before the computation, even if the return was defined last.

```
chainFront => 0/1
```
    Flag: Determines whether the FnReturn labels built by chaining off the other labels will be chained at the back (if 0) or at the front (if 1). Optional. Default: 1.

```
onPush => $code
```
    The code to execute whenever an FnBinding is pushed onto this FnReturn. This is useful to maintain the extended call contexts for the streaming function. Its argument can be specified in one of two forms: either just a code reference, or a reference to an array containing the code reference and the extra arguments for it. I.e. either onPush => $code

or `onPush => [ $code, @args ]`. The first argument of the function will always be the FnReturn object itself, with extra arguments going after it: `&$code($thisFnReturn, @optional_args)`. As usual, a source code string may be used instead of the function reference. Optional.

`onPop => $code`

The code to execute whenever an FnBinding is popped from this FnReturn. This is useful to maintain the extended call contexts for the streaming function. Its argument can be specified in one of two forms: either just a code reference, or a reference to an array containing the code reference and the extra arguments for it. I.e. either `onPop => $code` or `onPop => [ $code, @args ]`. The first argument of the function will always be the FnReturn object itself, with extra arguments going after it: `&$code($thisFnReturn, @optional_args)`. As usual, a source code string may be used instead of the function reference. Optional.

The FnReturn has a concept of clearing: and once any of the labels owned by the FnReturn gets cleared, the FnReturn is also cleared. The clearing drops the onPush and onPop handlers, thus breaking any reference cycles they might be engaged in.

Also, when an FnReturn gets destroyed, it clears and disconnects from the Unit all the labels defined in that FnReturn. The reason for that is that the labels in FnReturn cannot work without the FnReturn object anyway, and cannot hold a reference to it either (to avoid the cyclical references).

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

Get back the object's name.

```
$res = $fret->size();
```

Get the number of labels in the return.

```
@names = $fret->getLabelNames();
```

Get an array of label names, in the same order as they were defined (the order of the label definitions is important).

```
@labels = $fret->getLabels();
```

Get an array of references to the FnReturn's internal labels, in the same order as they were defined.

```
%labels = $fret->getLabelHash();
```

Get the interspersed list of label names and references, suitable to initialize a hash.

```
%namemap = $fret->getLabelMapping();
```

Get the interspersed list of label names and their indexes in order starting from 0, suitable to initialize a hash.

```
$label = $fret->getLabel($name);
```

Get a label by name. Will confess if this name was not defined.

```
$label = $fret->getLabelAt($idx);
```

Get a label by index, starting from 0. Will confess if the index is out of range.

```
$idx = $fret->findLabel($name);
```

Translate a label name to index. Will confess if this name was not defined.

```
%rts = $fret->getRowTypeHash();
```

Get the interspersed list of label names and references to their row types, suitable to initialize a hash. In Perl, this is the closest thing to the C++ API's RowSetType.

```
$res = $fret->equals($fret2);
```

```
$res = $fret->equals($fbind2);
$res = $fret->match($fret2);
$res = $fret->match($fbind2);
```

Compare the equality or match of types with an FnReturn or FnBinding. Since their type objects are not directly visible in the Perl API, the comparison has to be done on the FnReturns and FnBindings themselves. The types are considered equal if they contain the equal row types with equal names going in the same order. They are considered matching if they contain matching row types going in the same order, with any names. If the match condition seems surprising to you, think of it as "nothing will break if one type is substituted for another at execution time".

```
$res = $fret->same($fret2);
```

The usual check for two references referring to the same FnReturn object.

```
$fret->push($fbind);
```

Push a binding on the return stack. The binding must be of a matching type. The reference to the binding will be kept in the FnReturn until it's popped.

```
$fret->pop($fbind);
$fret->pop();
```

Pop a binding from the return stack. The binding argument specifies, which binding is expected to be popped. Without argument, pops any binding. The call with argument is recommended since it allows to catch any mess-ups with the return stack early. If the stack is empty or the top binding is not the same as the argument, will confess.

```
$res = $fret->bindingStackSize();
```

Get the current size of the return stack (AKA the stack of bindings). Useful for debugging.

```
@names = $fret->bindingStackNames();
```

Get the names of all the bindings on the return stack. Useful for debugging. The top of stack is on the right.

```
$res = fret->isFaceted();
```

Returns 1 if this FnReturn object is a part of a Facet.

19.16. FnBinding reference

An FnBinding represents a single call of a streaming function. It forwards the stream of rowops coming out of an FnReturn to the logic in the caller.

```
$fbind = Triceps::FnBinding->new($optName => $optValue, ...);
```

Construct an FnBinding object. The object may be reused for multiple calls. It's very much like a function call in a procedural language: present once in the program text but can be executed many times. The options are:

```
name => $name
```
    Name of this object. Will be used to create the names of the labels in it.

```
unit => $unit
```
    The unit where the labels specified as Perl inline code will be created. If no labels are specified in this format, this option is not required.

```
on => $fret
```
    The FnReturn on which this binding is defined. It determines what labels are available for binding. The created Fn-Binding may be used not only with that particular FnReturn but also with any FnReturn of a matching type. If that

other FnReturn has not an equal but only a matching type, the labels will be connected by the order in which they were defined in their respective FnReturns. It's a bit tricky: when an FnBinding is created, the labels in it get connected to the FnReturn on which it's defined by name. But at this point each of them gets assigned a number, in order the labels went in that original FnReturn. After that only this number matters: if this FnBinding gets connected to another matching FnReturn, it will get the data from the FnReturn's label with the same number, not the same name.

`labels => [ @definitions ]`

Definition of the labels in the FnBinding, that will be connected to the labels in FnReturn when the FnBinding is pushed on the FnReturn. The label names within an FnBinding must match the names from FnReturn. They may go in any order but internally will be reordered to match the label order from FnReturn. The value for this option is an array reference, with the labels defined as name-value pairs in the array, in one of two forms:

```
labels => [
  $name1 => $label1,
  $name2 => sub { ... },
  ...
]
```

If the second element in the pair is a function reference or a source code snippet in a string, a label will be automatically created using the unit from the option "unit", the row type from the matching label in FnReturn, and the provided function as the label's code. It's a convenient way to make your code shorter. The full names of these created labels will be "binding_name.label_name".

The labels in an FnBinding may belong to a different unit than the labels of the FnReturn. The FnReturn-FnBinding pair is an official way to connect two units.

It's also possible to include labels from multiple units into a single FnBinding. It's not particularly recommended but it happens to work. It creates complications when dealing with trays.

The details of the label execution in a binding are described in Section 15.13: "The gritty details of streaming functions scheduling" (p. 286) .

`withTray => 0/1`

Flag: Determines, whether the rowops coming into the FnBinding will be executed immediately (`withTray => 0`) or collected in a tray (`withTray => 1`). Optional, 0 by default.

`clearLabels => 0/1`

Flag: Determines whether the labels in FnBinding will be automatically cleared and forgotten in their unit when the FnBinding gets destroyed. This allows to create the temporary FnBindings dynamically and destroy them after the use, without leaking the labels. This option applies only to the labels specified as the label objects. The labels created by the FnBinding from the code references are always cleared and forgotten.

Unlike the C++ API, there is no way to set this option separately for each label, it's all or none. But normally all the labels need the same treatment anyway, so it just makes things easier.

Optional, 0 by default. Even for the dynamically created and destroyed FnBindings, this option is usually best left at 0, since usually the Label objects given to it will not be dynamically created, they would stay fixed between their uses. The only exception is if you create the labels along with the FnBinding and destroy them together as well. If you're doing that, probably you'd be better off having FnBinding create the labels for you from the code snippets, and those are always cleared and forgotten after the FnBinding destruction, no matter what this option is set to.

`Triceps::FnBinding::call($optName => $optValue, ...);`

A convenience function to create an FnBinding dynamically, do a single streaming function call with it, and destroy the FnBinding. This is not the most efficient way, since the FnBinding objects and its labels are constructed and destoryed on every call, but easy to use.

The options largely match the FnBinding constructor, with the addition of the options that do the streaming function call.

`name => $name`
> Name of the FnBinding object that will be constructed and destoryed for this call. Will be used to create the names of the labels in it.

`unit => $unit`
> The unit where the labels specified as Perl inline code will be created. If no labels are specified in this format, this option is not required.

`on => $fret`
> The FnReturn on which this binding is defined (see mode details on that in the description of FnBinding constructor) and also the FnReturn on which the call will be performed.

`labels => [ @definitions ]`
> Definition of the labels in the FnBinding, that will be connected to the labels in FnReturn when the FnBinding is pushed on the FnReturn. The format and meaning is the same as in the FnBinding constructor:

```
labels => [
  $name1 => $label1,
  $name2 => sub { ... },
  ...
]
```

`delayed => 0/1`
> Flag: An analog of option "withTray" of the FnBinding constructor. Determines, whether the rowops coming into the FnBinding will be executed immediately (`delayed => 0`) or collected in a tray and the tray executed after the FnBinding is popped from FnReturn (but before FnBinding is destroyed). Since the FnBinding is created and destroyed inside the call, there is no way to get the tray out of it, instead the tray gets automatically executed. Optional, 0 by default.

`clearLabels => 0/1`
> Flag: Determines whether the labels in FnBinding will be automatically cleared and forgotten in their unit when the FnBinding gets destroyed. The details are described with the constructor. Optional, 0 by default, and highly unlikely to be used.

`rowop => $rowop`
> A reference to rowop to execute as the argument to the streaming function. Exactly one of options "rowop", "rowops", "tray", "code" must be used.

`rowops => [ @rowops ]`
> A reference to an array of rowops (`rowops => [ @rowops ]`) to execute as the argument to the streaming function. This is just a convenient way to inline multiple rowops without constructing a Tray. Exactly one of options "rowop", "rowops", "tray", "code" must be used.

`tray => $tray`
> A reference to tray to execute as the argument to the streaming function. Exactly one of options "rowop", "rowops", "tray", "code" must be used.

`code => $code`

`code => [ $code, @args ]`
> A code reference or snippet to execute that sends the arguments to the streaming function. The source code snippet in a string also works. This option may also specify a function reference (or, again, its source code) together with its arguments. In this case the value would be an array reference: `code => [ $code, @args ]`. Exactly one of options "rowop", "rowops", "tray", "code" must be used.

`$res = $fbind->same($fbind2);`

The usual check for two references referring to the same FnBinding object.

```
$res = $fbind->equals($fret2);
$res = $fbind->equals($fbind2);
$res = $fbind->match($fret2);
$res = $fbind->match($fbind2);
```

Compare the equality or match of types with an FnReturn or FnBinding. Since their type objects are not directly visible in the Perl API, the comparison has to be done on the FnReturns and FnBindings themselves. The types are considered equal if they contain the equal row types with equal names going in the same order. They are considered matching if they contain matching row types going in the same order, with any names. If the match condition seems surprising to you, think of it as "nothing will break if one type is substituted for another at execution time".

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

Get back the name of the binding.

```
$res = $fbind->size();
```

Get the number of labels in the underlying FnReturn. That is, not just of the labels actually used in the binding but of all the labels that could be potentially used in it.

```
@names = $fbind->getLabelNames();
```

Get an array of label names, in the same order as they were defined in the underlying FnReturn. This will include all the possible labels from the FnReturn, even those that were not used in FnBinding.

```
@names = $fbind->getDefinedLabelNames();
```

Get an array of label names that are actually used in the FnBinding. The order is still the same as they were defined in FnReturn, only the ones not mentioned in FnBinding are skipped.

```
@labels = $fret->getLabels();
```

Get an array of references to the FnBinding's labels, in the order they were defined in the FnReturn. If some label name was defined in FnReturn but not used in FnBinding, that position will contain an `undef`.

```
%labels = $fret->getLabelHash();
```

Get the interspersed list of label names and references, suitable to initialize a hash. If some label name was defined in FnReturn but not used in FnBinding, the reference for it will be `undef`.

```
%namemap = $fret->getLabelMapping();
```

Get the interspersed list of label names and their indexes in order starting from 0, suitable to initialize a hash. This includes all the labels from FnReturn.

```
$label = $fret->getLabel($name);
```

Get a label by name. Will confess if this name was not defined in FnReturn. If a label was defined in FnReturn but not used in FnBinding, returns an `undef`.

```
$label = $fret->getLabelAt($idx);
```

Get a label by index, starting from 0. The index range is as defined in the underlying FnReturn. Will confess if the index is out of range. If a label was defined in FnReturn but not used in FnBinding, returns an `undef`.

```
$idx = $fret->findLabel($name);
```

Translate a label name to index. Will confess if this name was not defined.

```
%rts = $fret->getRowTypeHash();
```

Get the interspersed list of label names and references to their row types, suitable to initialize a hash. This includes all the label names from FnReturn, even if they weren't used in the FnBinding. In Perl, this is the closest thing to the C++ API's RowSetType.

```
$res = $fbind->withTray();
$res = $fbind->withTray($on);
```

Get back or change the value of "withTray" option. If the argument `$on` is used, it sets the new value. The change can be done at any time. Either form returns the previous value of the option. Disabling the tray mode discards the current tray.

```
$res = $fbind->traySize();
```

Get the size of the currently collected tray. If the tray is disabled, returns 0.

```
$res = $fbind->trayEmpty();
```

Check whether the tray is currently empty. Also returns 1 is the tray is disabled.

```
$tray = $fbind->swapTray();
```

Get the current tray out of the FnBinding, and replace it with an empty tray. If the tray contains a mix of rowops for different units (which might happen if the FnBinding refers to labels from different units), it will be detected at this point and the metod will confess. The Perl API doesn't support dealing with such mixed trays. It's still possible to deal with this situation in Perl by using `callTray()`. This method will also detect and confess if any of the rowops in the tray are destined for the cleared labels. Normally such rowops are simply ignored but this is a special case. It reduces the surprise if you accidentally clear the labels and then wonder why is the binding not working right.

```
$res = $fbind->traySize();
```

Get the size of the currently collected tray. If the tray is disabled, returns 0.

```
$fbind->callTray();
```

Logically, a combination of `swapTray()` and the call of that returned tray. However this method also does the correct handling of the trays that contain a mix of rowops for different units. Each rowop will be called in its proper unit.

19.17. AutoFnBind reference

AutoFnBind provides a scoped way of pushing and popping of the FnBindings on the FnReturns. The pushing happens when an AutoFnBind object is constructed, the popping when it is destroyed (or on an explicit clearing).

```
$ab = AutoFnBind->new($ret1 => $binding1, ...);
```

Create an AutoFnBind object. The arguments consist of pairs of FnReturns and FnBindings to push on them. The pushing will happen in the constructor in the order specified, the popping in the destructor or `clear()` method in the opposite order.

If any corruption of the call stack is detected at the popping time (i.e. the binding on the top of the stack is not the one that was pushed on it), this error will be detected. However since there is no way to report an error from a Perl destructor, this error message will be printed on stderr and the whole program will exit with the code 1. If you care about the ability to catch this error, use the method `clear()` before destroying the AutoFnBind object.

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

Pop the bindings, just as during the destruction, and discard the references to the FnReturn and FnBinding objects. After the clearing is done once, the destructor will not pop anything any more, and the repeated clearings will have no effect.

Any detected stack corruptions make this method confess. The popping will always go through the whole list of bindings, collecting the error reports along the way. At the end of the run it will make sure that it doesn't have any references to anything any more, and only then confess if any errors were found. This cleans up the things as much as possible and as

much as can be handled, but the end result will still not be particularly clean: the returns that got their stacks corrupted will still have their stacks corrupted, and some very serious application-level cleaning will be needed to continue. Probably a better choice would be to destroy everything and restart from scratch. But at least it allows to get safely to this point of restarting from scratch.

```
$res = $ab->same($ab2);
```

The usual check for two references referring to the same object.

19.18. App reference

An App represents a complete multithreaded Triceps application. There may be multiple applications running in the same process.

19.18.1. App instance management

The first part of the App API keeps track of all the App instances in the program, and allows to list and find them. It's global by necessity.

```
@apps = Triceps::App::listApps();
```

Returns the array of name-value pairs, values containing the App references, listing all the Apps in the program (more exactly, in this Triceps library, if you compile multiple Triceps libraries together by renaming them, each of them will have its own Apps list). The returned array can be placed into a hash.

```
$app = Triceps::App::find($name);
```

Find an App by name. If an App with such a name does not exist, it will confess.

```
$app = Triceps::App:make($name);
```

Create a new App, give it the specified name, and put it on the list. The names must not be duplicate with the other existing Apps, or the method will confess.

```
Triceps::App::drop($app);
Triceps::App::drop($appName);
```

Drop the app, by reference or by name, from the list. The App is as usual reference-counted and will exist while there are references to it. The global list provides one of these references, so an App is guaranteed to exist while it's still on the list. When dropped, it will still be accessible through the existing references, but obviously will not be listed any more and could not be found by name.

Moreover, a new App with the same name can be added to the list. Because of this, dropping an App by name requires some care in case if there could be a new App created again with the same name: it creates a potential for a race, and you might end up dropping the new App instead of the old one. Of course, if it's the same thread that drops the old one and creates the new one, then there is no race. Dropping an application by name that doesn't exist at the moment is an error and will confess.

Dropping the App by reference theoretically allows to avoid the race: a specific object gets dropped, and if it already has been dropped, the call has no effect. Theoretically. However in practice Perl has a limitation on passing the object values between threads, and thus whenever each thread starts, first thing it does is finding its App by name. It's a very similar kind of race and is absolutely unavoidable except by making sure that all the App's threads have exited and joined (i.e. harvesting them). So make sure to complete the App's thread harvesting before dropping the App in the harvester thread, and by then it doesn't matter a whole lot if the App is dropped by name or by reference.

19.18.2. App resolution

Many (but not all) of the App methods allow to specify the App either by reference or by name, and they automatically sort it out, doing the internal look-up by name if necessary. So the same method could be used as either of:

```
$app->method(...);
Triceps::App::method($app, ...);
Triceps::App::method($appName, ...);
```

Obviously, you cannot use the "->" syntax with a name, and obviously if the name is not in the app list, the method will confess. Below I'll show the calls that allow the dual formats as `Triceps::App::method($appOrName, ...)` but keep in mind that you can use the "->" form of them too with a reference.

```
$app = Triceps::App::resolve($appOrName);
```

Do just the automatically-sorting-out part: gets a reference or name and returns a reference either way. A newly created reference is returned in either case (not the argument reference). You can use this resolver before the methods that accept only a reference.

19.18.3. App introspection

```
$result = $app->same($app2);
```

Check if two references are for the same App object. Here they both must be references and not names.

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

Get the name of the App, from a reference.

```
Triceps::App::declareTriead($appOrName, $trieadName);
```

Declare a Triead (Triceps thread) in advance. Any attempts to look up the nexuses in that thread will then wait for the thread to become ready. (Attempts to look up in an undeclared and undefined thread are errors). This is necessary to prevent a race at the thread creation time. For the most part, the method `Triead::start()` just does the right thing by calling this method automatically and you don't need the use it manually, except in some very special circumstances. Declaring a thread with the same name more than once or declaring an already defined thread is perfectly all right.

```
%trieads = Triceps::App::getTrieads($appOrName);
```

Get the list of currently defined Trieads in the App, as name-value pairs. Keep in mind that the other threads may be modifying the list of Trieads, so if you do this call multiple times, you may get different results. However the Trieads are returned as references, so they are guaranteed to stay alive and readable even if they get removed from the App, or even if the App gets dropped.

19.18.4. App harvester control

The harvester logic is a necessary part of an App. It manages the collection of the Trieads that died and allows the rest of the program find out when the App terminates. It runs in a designated thread (an OS/Perl thread, not a Triead, though in some special cases it may use a thread left over from a Triead).

Normally the harvester thread runs in parallel with the rest of the App logic. To wait for the App's exit, wait for the harvester's thread to exit. In case of the small programs (such as the unit tests), the App's "main Triead" may share the thread with the harvester: first the main Triead method would run, and after it returns, the harvester. This obviously means that the threads that have exited will be sitting in the zombie state during the life of the App, until the harvester has a chance to run.

The harvesting is an absolutely necessary part of the App life cycle, however in most of the usage patterns (such as with `Triead::startHere()` or `App::build()`) the harvester is called implicitly from the wrapping library functions, so you don't need to care about it.

```
$app->harvester(@options);
```

Run the harvester in the current thread. The harvester gets notifications from the threads when they are about to exit, and joins them. After all the threads have been joined, it automatically drops the App, and returns.

Note that if you're running the harvester manually, you must call it only after the first thread has been defined or at least declared. Otherwise it will find no threads in the App, consider it dead and immediately drop it.

If the App was aborted, the harvester will normally confess after if had joined all the threads and disposed of the App, unless the option "die_on_abort" (see below) has been set to 0. This propagates the error to the caller. However there is a catch: if some of the threads don't react properly by exiting on an abort indication, the program will be stuck and you will see no error message until these threads get unstuck, possibly forever.

Options:

```
die_on_abort => 0/1
```
Flag: If the App was aborted, the harvester will normally confess after if had joined all the threads and disposed of the App. Setting this option to 0 will make the aborts silently ignored. This option does not affect the errors in joining the threads: if any of those are detected, harvester will still confess after it had disposed of the app. Optional. Default: 1.

```
$dead = $app->harvestOnce();
```

Do one run of the harvesting. Joins all the threads that have exited since its last call. If no threads have exited since then, returns immediately. Returns 1 if all the threads have exited (and thus the App is dead), 0 otherwise. If a thread join fails, immediately confesses (if multiple threads were ready for joining, the ones queued after the failed one won't be joined, call `harvestOnce()` once more to join them).

```
$app->waitNeedHarvest();
```

Wait for at least one thread to become ready for harvesting. If the App is already dead (i.e. all its threads have exited), returns immediately.

These two methods allow to write the custom harvesters if you're not happy with the default one. The basic harvester logic can be written as:

```
do {
  $app->waitNeedHarvest()
} while(!$app->harvestOnce());
$app->drop();
```

However the real harvester also does some smarter things around the error handling. You can look it up in the source code in `cpp/app/App.cpp`.

19.18.5. App state management

```
$res = Triceps::App::isDead($appOrName);
```

Returns 1 if the App is dead (i.e. it has no alive Trieads, all the defined and declared threads have exited). Right after the App is created, before the first Triead is created, the App is also considered dead, and becomes alive when the first Triead is declared or defined. If an App becomes dead later, when all the Trieads exit, it can still be brought back alive by creating more Trieads. But this is considered bad practice, and will cause a race with the harvester (if you want to do this, you have to make your own custom harvester).

Calling this method with a name for the argument is probably a bad idea, since the App may be dropped quickly quickly after it becomes dead, and you may end up with this method confessing when it could not find the dropped App.

```
$res = Triceps::App::isShutdown($appOrName);
```

Returns 1 if the App has been requested to shut down, either normally or by being aborted. The Trieads might still run for some time, until they properly detect and process the shutdown, and exit. So this condition is not equivalent to Dead, althouh they are connected. If any new Trieads get started, they will be shut down right away and won't run.

To reiterate: if all the Trieads just exit by themselves, the App becomes dead but not shut down. You could still start more Trieads and bring the App back alive. If the App had been shut down, it won't become immediately dead, but it will send

the shutdown indication to all the Trieads, and after all of them eventually exit, the App will become dead too. And after shutdown there is no way to bring the App back alive, since any new Trieads will be shut down right away. There might be a short period until they detect the shutdown, so the App could spike as alive for a short time, but then will become dead again.

```
$res = Triceps::App::isAborted($appOrName);
```

Returns 1 if the App has been aborted. The App may be aborted explicitly by calling the method `abortBy()`, or the thread wrapper logic automatically converts any unexpected deaths in the App's threads to the aborts. If any thread dies, this aborts the App, which in turn requests the other threads to die on their next thread-related call. Eventually the harvester collects them all and confesses, normally making the whole program die with an error.

```
($tname, $message) = Triceps::App::getAborted($appOrName);
```

Get the App abort information: name of the thread that caused the abort, and its error message.

```
Triceps::App::abortBy($appOrName, $tname, $msg);
```

Abort the application. The thread name and message will be remembered, and returned later by `getAborted()` or in the harvester. If `abortBy()` is called multiple times, only the first pair of thread name and message gets remembered. The reason is that on abort all the threads get interrupted in a fairly rough manner (all their ongoing and following calls to the threading API die), which typically causes them to call `abortBy()` as well, and there is no point in collecting these spurious messages.

The thread name here doesn't have to be the name of the actual thread that reports the issue. For example, if the thread creation as such fails (maybe because of the OS limit on the thread count) that gets detected by the parent thread but reported in the name of the thread whose creation has failed. And in general you can pass just any string as the thread name, App itself doesn't care, just make it something that makes sense to you.

```
Triceps::App::waitDead($appOrName);
```

Will wait for the App to become dead and return after that. Make sure to not call `waitDead()` from any of App's Trieads: that would cause a deadlock.

```
Triceps::App::shutdown($appOrName);
```

Shut down the App. The shutdown state is sticky, so any repeated calls will have no effect. The call returns immediately and doesn't wait for the App to die. If you want to wait, call `waitDead()` afterwards. Make sure to not call `waitDead()` from a Triead: that would cause a deadlock.

```
Triceps::App::shutdownFragment($appOrName, $fragName);
```

Shut down a named fragment. This does not shut down the whole App, it just selectively shuts down the Trieads belonging to this fragment. See the explanation of the fragments in Section 16.6: "Dynamic threads and fragments in a socket server" (p. 304) . The fragment shutdown is not sticky: after a fragment has been shut down, it's possible to create another fragment with the same name. To avoid races, a fragment may be shut down only after all its Trieads are ready. So the caller Triead must call `readyReady()` before it calls `shutdownFragment()`. If any of the fragment's Trieads are not ready, the call will confess.

19.18.6. App drain control

The next few methods have to do with the drains. Generally, using the AutoDrain class to automatically limit the scope of the drains is a better idea. But if the automatic scoping is not desired, the App methods can be used directly.

```
Triceps::App::requestDrain($appOrName);
```

Request a shared drain. Does not wait for the drain to complete. This method may be called repeatedly, potentially from multiple threads, which will keep the drain active and increase the recursion count. The drain will be released when `undrain()` is called the matching number of times.

If an exclusive drain by another thread is active when `requestDrain()` is called, the call will be stuck until the exclusive drain becomes released (and potentially more exclusive drains might be queued up before the shared drain, using the POSIX read-write-lock implementation). Otherwise the call will set the appropriate state and return immediately.

If this thread has requested an exclusive drain previously (and didn't release it yet), an attempt to get a shared drain in the same thread will likely deadlock. The same applies in the opposite order as well.

Once the shared drain is requested, the input-only threads will be blocked from sending more data (any their attempts to flush their write facets will get stuck until release), and the rest of the threads will continue churning through the data buffered in the nexuses until all the nexuses are empty and there is no more data to process (yes, these threads will continue writing to their write facets).

It is important to not request a shared drain from an input-only thread and then try to write more data into an output facet, that would deadlock. The whole concept is doable, but an exclusive drain must be used instead ("exclusive" means that a designated input-only thread is excluded from blocking).

If the new Trieads are created while a drain is active, these Trieads will be notified of the drain. This means that the input-only Trieads won't be able to send any data until the drain is released. However the Trieads in the middle of the model will follow the normal protocol for such threads: the drain will become incomplete after the Triead is marked as ready and until it blocks on the following `TrieadOwner::nextXtray()`. Normally this should be a very short amount of time. However such Trieads should take care to check `TrieadOwner::isRdDrain()` and never send any rowops on their own before reading from `nextXtray()` if they find that `isRdDrain()` returns true. Otherwise they may introduce the data into the model at a very inconvenient moment, when some logic expects that no data is changing, and cause a corruption. This is the same caveat as for using `nextXtray()` varieties with the timeouts: if you want to send data on a timeout, always check `isRqDrain()`, and never send any data on timeouts if `isRqDrain()` returns true.

```
Triceps::App::waitDrain($appOrName);
```

Wait for the requested shared drain to complete. This means that all the queues in all the nexuses have become empty.

The effect of `waitDrain()` without a preceding `requestDrain()` is undefined. If you definitely know that some other thread has requested a drain and didn't release yet, it will work as normal, so you can have any number of threads wait for drain in parallel. (And the semantics, shared or exclusive, will match that currently active request). If no thread requested a drain, it might either return immediately irrespective of the state of the nexuses or might wait for some other thread to request a drain and succeed.

This call may also be used with an exclusive (or shared) drain requested through a TrieadOwner or any drain requested through an AutoDrain (though a better style is to use the same object as used for requesting the drain).

```
Triceps::App::drain($appOrName);
```

A combination of `requestDrain()` and `waitDrain()` in one call.

```
Triceps::App::undrain($appOrName);
```

Release the drain and let the normal processing continue. Make sure to call it exactly the same number of times as the `requestDrain()`.

This call may also be used with a drain requested through a TrieadOwner (though a better style is to use the same object as used for requesting the drain).

```
$result = Triceps::App::isDrained($appOrName);
```

Check whether the App is currently drained (i.e. all the nexuses are empty), without waiting for it. Returns 1 if drained, 0 if not. If no drain is active during this call, the result is undefined, not even in the "best effort" sense: it may return 1 even if there are millions of records queued up. The only reliable way is to request a drain first.

This call may also be used with a drain requested through a TrieadOwner or through an AutoDrain.

App drain control

19.18.7. App start timeout

Even though Triceps is quite eager in watching for deadlocks in the threads topology, it's still possible to get some threads, and with them the whole program, stuck during initialization. For example, if you declare a thread and then never define it. These situations are very unpleasant because you start the program and expect it to work but it doesn't, without any indication to why. So Triceps imposes an initialization timeout. The App (and thus all its threads) must become ready within the timeout after the definition or declaration of the last Triead. If not, the App will be aborted (and the error message will tell, which thread did not initialize in time). The same applies to the creation of Trieads (usually in the fragments) after the App is already running: all the threads must become ready within the timeout since the last thread has been defined or declared.

```
Triceps::App::setTimeout($appOrName, $main_to_sec);
Triceps::App::setTimeout($appOrName, $main_to_sec, $frag_to_sec);
```

Set the readiness timeout.

The default timeout is 30 seconds, or symbolically

```
&Triceps::App::DEFAULT_TIMEOUT
```

(subject to possible changes of the exact value in the future).

But the timeout can be changed. Technically, there are two timeouts:

- one starts when the App is created

- one restarts when any Triead is defined or declared

For the App to be aborted, both timeouts must expire. By default they are set to the same length, so only the second timeout really matters, since it will always be the last one to start and last one to end. But if you set the first timeout to be longer, you can allow for a longer initialization period at the App start ("main timeout") than later when more threads are added to a running App ("fragment timeout", since the threads added later are typically the threads in fragments).

The one-argument form of `setTimeout()` sets both timeouts to the same value, the two-argument form sets these timeouts separately.

The timeouts may be set only before the first thread has been created. This is largely due to the historical reasons of implementation, with the current implementation it should actually be safe to allow changing the timeouts at a later point as well, and this limitation may be removed in the future.

The timeout values are represented as integer whole seconds.

Note that it's still possible to get the initialization stuck without any indication if it gets stuck in the other libraries. The reason is that the harvester waits for all the threads to be joined before it propagates the error, so if a thread doesn't get aborted properly and doesn't get joined, the harvester will be stuck. For example, if you open a file on NFS as a part of Triead initialization, and the NFS server doesn't respond, this thread will be stuck for a long time if not forever. The App will detect a timeout and try to interrupt the threads but the NFS operations are often not interruptable, so the harvester will wait for this thread to complete this operation and exit before it propagates the error, and thus the whole program will be silently stuck forever (and to avoid this, the NFS mounts should be done in the "soft" mode but it's a separate story). This will likely be improved in the future but it needs more thinking about how to do it right.

```
Triceps::App::setDeadline($appOrName, $deadline_sec);
```

Set the "main" timeout in the form of an absolute deadline. This is actually closer to the way it works internally: the limit is expressed as a deadline, and `setTimeout()` just adds the timeout value to the current time to compute the deadline, while `setDeadline()` sets it straight. Like `setTimeout()`, this may be called only before any Trieads were created.

The time is represented as floating-point seconds since epoch. `Triceps::now()` can be used to get the current time with the fractional seconds (or if you don't care about the fractional seconds, you can always use the stock `time()`).

```
Triceps::App::refreshDeadline($appOrName);
```

Restart the "fragment" timeout. Same as when a Triead is defined or declared, only without involving a Triead. This method can be called at any time.

19.18.8. File descriptor transfer through an App

Perl doesn't allow the sharing or transfers of the file handles between the threads. So Triceps works around this limitation by passing the underlying file descriptors through the App. After that the receiving Triead may construct a new file handle from the descriptor.

The concepts are described in detail in Section 16.5: "Threads and file descriptors" (p. 301) , this is just a short reference. This API works under the hood of `TrieadOwner::trackGetFile()` and friends but can also be used directly.

```
Triceps::App::storeFd($appOrName, $name, $fd, $fileClassName);
```

Store a file descriptor in the App object, allowing to load it into other threads, and thus pass it around between the threads. $name is the name for this descriptor that will later be used to get the file descriptor back (generally, you want to generate a unique name for each file descriptor stored to avoid confusion, and then pass this name to the target thread). `$fd` is the file descriptor, an integer number, normally received from `fileno()`. The file descriptor is dupped before it gets stored, so the original will continue to exist, and if you have no other use for it, you should close it.

The `$fileClassName` allows to pass through the information about the class of the Perl file handle (such as IO::Handle or IO::Socket::INET) that owned this descriptor originally, and re-create the correct object later when loading the descriptor back. It's a plain string, and an empty string can be used for the plain files.

If a file descriptor with this name already exists in the App, this call will confess.

```
($fd, $fileClassName) = Triceps::App::loadFd($appOrName, $name);
```

Load back the file descriptor that was previously stored. The file handle class name is loaded back along with it. If no descriptor with such a name is stored in the App, it will confess. The descriptor will keep existing in the App, so to keep things consistent, there are two options:

One is to let your code take over the ownership of the file descriptor, and tell the App to forget about it with `forgetFd()`.

The other one is to never close the received descriptor in your code (a good example would be to dup it right away for the future use and then leave the original alone), and let App keep its ownership.

```
($fd, $fileClassName) = Triceps::App::loadDupFd($appOrName, $name);
```

Very much the same as `loadFd()`, only does the dupping for you and returns the dupped descriptor. In this case your code is responsible for closing that descriptor. Which method is more suitable, `loadFd()` or `loadDupFd()`, depends on the nature of the code that will use the file descriptor and whether you want to leave the descriptor in the App for more threads to load.

```
Triceps::App::forgetFd($appOrName, $name);
```

Forget the file descriptor in the App. It doesn't get closed, so closing it becomes your responsibility, or it will leak. If no descriptor with this name exists, the call will confess.

```
Triceps::App::closeFd($appOrName, $name);
```

Close and forget the file descriptor in the App. If no descriptor with this name exists, the call will confess.

```
Triceps::App::storeFile($appOrName, $name, FILE);
```

A convenience wrapper for `closeFd()`, calls `fileno()` on the file and stores the resulting file descriptor. The name of the class of the file handle is stored along with the descriptor.

```
Triceps::App::storeCloseFile($appOrName, $name, FILE);
```

Stores the file descriptor extracted from the file, and closes the original file (since the file descriptor gets dupped on store, that copy continues to exist in the App).

```
$file = Triceps::App::loadDupFile($appOrName, $name, $mode);
```

Load a file descriptor and build a file handle object from it, using the stored file handle class name. The mode string may be specified in either the `open()` format (`</>/>>/+</+>/+>>`) or the C stdio format (`r/w/a/r+/w+/a+`). Note that the mode must match or be a subset of the mode used to originally open the file descriptor. If you open a file read-only, store its descriptor, and load back as a write-only file, you will have a bad time.

There is no corresponding `loadFile()`, since `loadFd()` is a more dangerous method that is useful for the low-level operations but doesn't make much sense for the higher level.

```
$file = Triceps::App::loadDupFileClass($appOrName, $name, $mode, $fileClassName);
```

Load a file descriptor and build an arbitrary handle object from it, overriding the stored class name. The class (specified by its name) should normally be a subclass of IO::Handle, or at the very least must implement the method `new_from_fd()` similar to IO::Handle.

19.18.9. App build

There are multiple ways to build an App:

- Create the App manually with `App::make()`, then manually create its Trieads and start the harvester. To wait for the app completion, wait for the harvester thread to exit.

- Create the App with `App::build()`, as described below. The Trieads get constructed from the function that is passed as an argument to `build()`. After running the thread construction, `build()` runs the harvester automatically, and returns after the harvester exits.

- Create the App with `Triead::startHere()`. It's very much like `App::build()`, except that the code started from it is an actual Triead, running in the context of the current thread. This is convenient for running the short-lived Apps that get started, run for a short time and then need to return the results back to the calling thread. The harvester also runs automatically by `Triead::startHere()` but only after the the "main" Triead exits.

```
Triceps::App::build($name, $builder);
```

Build an App instance. It creates the App instance, then the builder function is called to create the App's nexuses and threads, then the harvester is executed, that eventually destroys the App object after collecting all its threads. For a very basic example:

```
Triceps::App::build "a1", sub {
  Triceps::App::globalNexus(
    name => "types",
    rowTypes => [
      rt1 => $rt1,
    ],
  );
  Triceps::Triead::start(
    app => $Triceps::App::name,
    thread => "t1",
    main => sub {
      my $opts = {};
      &Triceps::Opt::parse("t1 main", $opts, {@Triceps::Triead::opts}, @_);
      my $to = $opts->{owner};
      $to->importNexus(
        from => "global/types",
        import => "writer", # if importing just for the types, use "writer"!
```

```
        );
        $to->readyReady();
    },
  );
};
```

The builder function runs in the current Perl thread, however from the logical standpoint it runs in the App's first Triead named "global" (this Triead name is hardcoded in `build()`). This frees you from the worry about the App being technically dead until the first Triead is created: by the time the builder function is called, the first Triead is already created. However it also means that you can't change the readiness timeouts. After the builder function returns, the global Triead exits, and the harvester starts in the same current Perl thread.

The builder has access to a few global variables:

• `$Triceps::App::name` is the App name, from the `build()` first argument.

• `$Triceps::App::app` is the reference to the App object.

• `$Triceps::App::global` is the reference to the TrieadOwner object of the global Triead.

After the builder function exits, these variables become undefined.

```
Triceps::App::globalNexus(@nexusOptions);
```

Creates a nexus with the import mode of "none", on the global Triead. The arguments are the same options as for the normal nexus creation. This is a simple convenience wrapper for the nexus creation. Since the global thread is supposed to exit immediately, there is no point in importing this nexus into it.

The global thread is found by this function from `$Triceps::App::global`, so this method can only be used from inside the builder function of `build()`.

19.19. Triead reference

The Triead class is the public interface of a Triceps thread, i.e. what of it is visible to the other threads. It's intended pretty much for introspection only, and all its method are only for reading the state. They are all synchronized, but of course the thread may change its state at any moment. The Triead objects can be obtained by calling `App::getTrieads()`.

The class also contains some static methods that are used to construct the Trieads.

```
$result = $t->same($t2);
```

Check that two Triead references point to the same Triead object.

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

Get the Triead's name.

```
$fragment = $t->fragment();
```

Get the name of the Triead's fragment. If the Triead doesn't belong to a fragment, returns an empty string "".

```
$result = $t->isConstructed();
```

Check whether the Triead has been constructed. For the explanation of the Triead lifecycle states, see Section 16.2: "The Triead lifecycle" (p. 288) .

```
$result = $t->isReady();
```

Check whether the Triead is ready.

```
$result = $t->isDead();
```

Check whether the Triead is dead.

```
$result = $t->isInputOnly();
```

Check whether the Triead is input-only, that is, it has no reader nexuses imported into it. When the Triead is created, this flag starts its life as `false` (0 for Perl), and then its correct value is computed when the Triead becomes ready. So, to check this flag correctly, you must first check that the Triead is ready.

```
@nexuses = $t->exports();
```

Get the list of nexuses exported from this Triead, as name-value pairs, suitable to be assigned into a hash. The values are the references to nexus objects.

```
@nexuses = $t->imports();
@nexuses = $t->readerImports();
@nexuses = $t->writerImports();
```

Get the list of nexuses imported into this Triead, as name-value pairs. The `imports()` returns the full list, without the ability to tell, which nexuses are imported for reading and which for writing, while `readImports()` and `writeImports()` return these subsets separately.

The names here are the "as-names" used for the import (the full names of the Nexuses can be obtained from the Nexus objects). The values are the references to nexus objects.

The next part of the API is the static construction methods. They really are wrappers of the TrieadOwner methods but Triead is a shorter name, and thus more convenient.

```
Triceps::Triead::start(@options);
```

Start a new Triead in a new Perl thread. The options are:

`app => $appname`
    Name of the App that owns the new Triead. The App object will be looked up by name for the actual construction.

`thread => $threadname`
    Name of the new Triead.

`fragment => $fragname`
    Name of the new Triead's fragment. Optional. Default: "", which means no fragment.

`immed => 0/1`
    Flag: when the new thread imports its nexuses, it should import them in the immediate mode. This flag is purely advisory, and the thread's main function is free to use or ignore it depending on its logic. It's provided as a convenience, since it's a typical concern for the helper threads. Optional. Default: 0.

`main => $code`
    The main function of the thread that will be called with all the options of `start()` plus some more:

    ```
    &$func(@opts, owner => $ownerObj)
    ```

    The extra option of the main is:

    `owner => $ownerObj`
        The TrieadOwner object constructed for this thread, through which its state can be controlled.

    Also, any other options may be added, and they will be forwarded to the main function without parsing. The main function is then free to parse them by itself, and if it finds any unknown options, it will fail.

    For the convenience of writing the main functions, the set of "standard" options is provided in the global variable

    ```
    @Triceps::Triead::opts
    ```

The main function then uses this variable as a preamble for any of its own options, for example:

```
sub listenerT
{
    my $opts = {};
    &Triceps::Opt::parse("listenerT", $opts, {@Triceps::Triead::opts,
        socketName => [ undef, \&Triceps::Opt::ck_mandatory ],
    }, @_);
    ...
}
```

Another convenience method is:

```
Triceps::Triead::startHere(@options);
```

It's very much the same as `start()` but starts the Triead in the current Perl thread. It's intended for the short-lived Apps that perform some computation and then have to return the result into the original thread. The unit tests are a good example of such apps.

And because of the typical usage, this method has some extra functionality compared to the plain `start()`: unless told otherwise, it will first create the App (with the name specified by the option "app"), then construct and run it, and after the main function of this thread exits, run the harvester and drop the App. So it just does the whole package, similar to `App::build()`, only typically the Triead started here runs as a normal real Triead, collects the results of computations and places them into some variables (global or referenced by the user-specific options of the main function).

This extra functionality can be disabled by the additional options (disable by setting them to 0):

```
harvest => 0/1
```
 After the main function exits, automatically run the harvesrer. If you set it to 0, don't forget to call the harvester after this function returns. Optional. Default: 1.

```
makeApp => 0/1
```
 Before doing anything, create the App. Obviously, this App must not exist yet. Optional. Default: 1.

These options are not passed through to the main function, unlike all the others.

19.20. TrieadOwner reference

TrieadOwner is the thread's private interface used to control its state and interact with the App (the App uses the thread's identity to detect the deadlocks in these interactions). Whenever a Triead is constructed, its OS/Perl thread receives the TrieadOwner object for it.

19.20.1. TrieadOwner construction

Normally the TrieadOwner object is constructed inside `Triead::start()` or `Triead::startHere()` and passed to the thread's main function. The following constructor is used inside `start()`, and it's pretty much a private method. The only reason to use it would be if you want to do something very unusual, and even then you probably should write a wrapper method for your unusual thing and then call that wrapper method. The constructor constructs both Triead and TrieadOwner as a two sides of the same item, and registers the thread with the App.

```
$to = Triceps::TrieadOwner::new($tid, $handle, $appOrName, $tname, $fragname);
```

Here `$tid` is the Perl thread id where this TrieadOwner belongs (it can be obtained with `$thr->tid()`). `$handle` is the Perl thread's low-level handle (as in `$thr->handle_()`), it's the underlying POSIX thread handle, used to interrupt the thread on shutdown (the long story is that in the Perl threads the `kill()` call doesn't actually send a signal to another thread but just sets a flag; to interrupt a sleeping system call a real signal has to be delivered through the POSIX API). `$handle` is a dangerous argument, and passing a wrong value there may cause a crash.

Both `$tid` and `$handle` may be undef. If `$tid` is undef, the thread won't be joined by the harvester and you can either detach it or join it yourself. If either `$tid` or `$handle` is undef, the thread won't be interrupted on shutdown.

The signal used for interruption is `SIGUSR2`. Triceps sets its default handler that does nothing on this signal. This is done in the method `Triceps::sigusr2_setup()`, and it gets called during the Triceps module loading. Internally it translates to the C++ method `Sigusr2::setup()` that sets the dummy handler on the first call. The reason why it has to be done in C++ is that Perl 5.19 crashes on SIGUSR2 if the handler is set in Perl. If you're interested in the details, see https://rt.perl.org/rt3//Public/Bug/Display.html?id=118929.

`$appOrName` is the App object or its name that would be automatically looked up (or will confess if not found). `$tname` is the name for the thread, that must be unique within the App (though it might be declared before). `$fragname` is the name of the fragment where the thread belongs, use "" for no fragment.

19.20.2. TrieadOwner general methods

`$app = $to->app();`

Get the App where this Triead belongs.

`$unit = $to->unit();`

Whenever a Triead is constructed, a Unit is automatically created to execute its logic. This call returns that unit. When the Triead is destroyed, the unit will be cleaned and unreferenced.

The unit is named the same as the thread.

`$to->addUnit($moreUnit);`

It's possible to split the Triead's logic into multiple units, all running in the same Perl thread. This call puts an extra unit under Triead's control, and has two effects: First, the unit will be referenced for the life of the Triead, and cleaned and unreferenced when the Triead is destroyed. Second, when the Triead's main loop runs, after each incoming rowop it will check all the controlled units for any rowops scheduled in them, and will run them until all such rowops are processed.

The names of the units are not checked in any way, it's your responsibility to name them sensibly and probably differently from each other.

The repeated calls with the same unit will have no effect.

`$to->forgetUnit($moreUnit);`

Pull a unit out of Triead's control. After that the cleaning of the unit becomes your responsibility. The thread's main unit cannot be forgotten, the attempts to forget it will be simply ignored. The same goes for the units that aren't under the Triead's control in the first place, these calls are ignored.

`@units = $to->listUnits();`

Get the list of units under Triead's control. The main unit (the same as returned with `$to->unit()`) will always be the first in the list. The list contains only the unit references, **not** the name-value pairs (and you can always get the names from the unit objects themselves).

`$triead = $to->get();`

Get the public API of this Triead.

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

Get this Triead's name.

`$frag = $to->fragment();`

Get the name of this Triead's fragment ("" if not in a fragment).

```
$to->markConstructed();
```

Advance the Triead to the Constructed state. After that point no more nexuses may be exported in the Triead. Any look-ups by other Trieads for the Nexuses of this Triead will proceed at this point, either succeeding or failing (if the requested nexus is not exported).

In all the `mark*()` methods, the state advance is cumulative: it brings the thread through all the intermediate states.

If the Triead is already in the Constructed or later state, this call has no effect.

```
$to->markReady();
```

Advance the Triead to the Ready (fully initialized) state. After that point no more nexuses may be imported into this Triead.

In all the `mark*()` methods, the state advance is cumulative: it brings the thread through all the intermediate states.

If the App has been already shut down, this Triead will be immediately requested to die.

If this is the last Triead in the App to become ready, this method will invoke the check for the topological correctness of the App. If the check finds an error (a loop of nexuses of the same direction), it will abort the App and confess with a message describing the nature of the error.

If the Triead is already in the Ready or later state, this call has no effect.

```
$to->readyReady();
```

Mark this Triead as Ready and wait for all the App's Trieads to become Ready. There is no method that just waits for readiness because that would be likely causing a deadlock. When the thread waits for readiness, it must be ready itself, so this call does both. All the error checks of `markReady()` apply.

It is possible and reasonable to call this method repeatedly: more Trieads may be added to the App later, and it's a good idea to call `readyReady()` again before communicating with these new threads. Otherwise any rowops sent before these threads become ready will never arrive to these threads.

```
$to->markDead();
```

Mark this Triead as Dead. A dead thread will not receive any more input, and any output from it will be thrown away. This notifies the harvester that it needs to join the Perl thread, so there should not be too much of a delay between making this call and exiting the Perl thread. The repeated calls have no effect.

Normally the `Triead::start()` and `startHere()` call `markDead()` automatically in their wrapper logic, and there is no need for a manual call. However if you decide to bypass them, you must call `markDead()` manually before exiting the thread, or the harvester will be stuck forever waiting for this thread to die.

In all the `mark*()` methods, the state advance is cumulative: it brings the thread through all the intermediate states. But with `markDead()` there is an even more interesting twist. Suppose there is an application with an incorrect topology, and all the Trieads in it but one are ready. That last Triead then experiences an error, and proceeds directly to call `markDead()` and then exit. This `markDead()` will involve an intermediate step marking the Triead as ready. Since it's the last Triead to be ready, it will trigger the topology check, and since the topology is incorrect, it will fail. If it happened in `markReady()`, the method would confess. But in `markDead()` confessing doesn't make a whole lot of sense: after all, the thread is about to exit anyway. So `markDead()` will catch all these confessions and throw them away, it will never fail. However the failed check will still abort the App, and the rest of the threads will wake up and fail as usual.

`markDead()` also clears and unreferences all the TrieadOwner's registered units, not waiting for the TrieadOwner object to be destroyed. This unravels any potential cyclic references where the code in a label might be referring back to the TrieadOwner.

```
$to->abort($msg);
```

Abort the App with a message. This is a convenience wrapper that translates to `App::abortBy()`.

```
$result = $to->isRqDead();
```

Check whether the thread was requested to die. For most threads, `mainLoop()` does this check automatically, and `nextXtray()` also returns the same value. However in the special cases, such as doing some long processing in response to a rowop, or doing some timeouts, it's best to do a manual check of `isRqDead()` periodically and abort the long operation if the thread has been requested to die, since any output will be thrown away anyway.

When a thread is requested to die, it gets immediately disconnected from the nexuses. It will not get any more input (though it might still go through the buffers in its input facets), and any output from it will be discarded.

Note that even when the Triead has been requested to die, it still must call `markDead()` when it actually dies (normally the `Triead::start()` or `startHere()` takes care of it in its wrapper).

```
$result = $to->isConstructed();
$result = $to->isReady();
$result = $to->isDead();
$result = $to->isInputOnly();
```

Check the state of the Triead, the same as Triead methods.

```
$facet = $to->makeNexus(@options);
```

Create, export and (optionally) import a nexus. The result is an imported Facet of this Nexus, except when the options specify the no-import mode (then the result will be `undef`). Confesses on errors.

The options are:

`name => $name`
    Name of the nexus, it will be used both as the export name and the local imported "as-name" of the facet.

`labels => [ @definitions ]`
    Defines the labels similarly to FnReturn in a referenced array. The array contains the pairs of (`label_name`, `label_definition`), with definitions in one of two forms:

```
labels => [
  name1 => $rowType1,
  name2 => $fromLabel2,
  ...
]
```

    The definition may be either a RowType, and then a label of this row type will be created, or a Label, and then a label of the same row type will be created and chained from that original label. The created label objects can be later found from Facets, and used like normal labels, by chaining them or sending rowops to them (chaining from them is probably not the best idea, although it works anyway).

    Optional, or may be an empty array; the implicit labels _BEGIN_ and _END_ will allways be added automatically if not explicitly defined.

    The labels are used to construct an implicit FnReturn in the current Triead's main unit, and this is the FnReturn that will be visible in the Facet that gets imported back. If the import mode is "none", the FnReturn will still be constructed and then abandoned (and freed by the reference count going to 0, as usual). The labels used as `$fromLabel` above must always belong to the Triead's main unit.

`rowTypes => [ @definitions ]`
    Defines the row types exported in this Nexus as a referenced array of name-value pairs:

```
rowTypes => [
  name => $rowType,
```

```
      ...
   ]
```

The types imported back into this Triead's facet will be references to the exact same type objects. Optional, or may be empty.

`tableTypes => [ @definitions ]`
    Defines the table types exported in this Nexus as a referenced array of name-value pairs:

```
tableTypes => [
  name => $tableType,
  ...
]
```

The types imported back into this Triead's facet will be references to the exact same type objects. Optional, or may be empty.

`reverse => 0/1`
    Flag: this Nexus goes in the reverse direction. The reverse nexuses are used to break up the topological loops, to prevent the deadlocks on the queueing. They have no limit on the queue size, and the data is read from them at a higher priority than from the direct nexuses. Default: 0.

`chainFront => 0/1`
    Flag: when the labels are specified as `$fromLabel`, chain them at the front of the original labels. Default: 1.

    The default is this way because chaining at the front is what is typically needed. The reasons are described at length in Section 16.3: "Multithreaded pipeline" (p. 290) , but the short gist is that you might want to send the rows from both the inputs, intermediate points, and the end of processing into an output nexus. It's most convenient to create the nexus in one go, after the whole thread's computation is defined. But the rowops from the earlier stages of computations have to come to the nexus before the rowops from the later stage. Chaining at the front ensures that each such label will send the rowop into the nexus first, and only then to the next stage of the computation.

`queueLimit => $number`
    Defines the size limit after which the writes to the queue of this Nexus block. In reality because of the double-buffering the queue may contain up to twice that many trays before the future writes block. This option has no effect on the reverse nexuses. Default: `&Facet::DEFAULT_QUEUE_LIMIT`, 500 or so.

`import => $importType`
    A string value, essentially an enum, determining how this Nexus gets immediately imported back into this Triead. The supported values are:

    • `reader` (or anything starting from "read") - import for reading

    • `writer` (or anything starting from "write") - import for writing

    • `none` (or anything starting from "no") - do not import

    Case-insensitive. The use of the canonical strings is recommended.

`$facet = $to->importNexus(@options);`

Import a nexus into this Triead. Returns the imported Facet. The repeated attempts to import the same Nexus will return references to the same Facet object. Confesses on errors. An attempt to import the same nexus for both reading and writing is an error.

The options are:

`from => "$thread_name/$nexus_name"`
    Identifier of the nexus to import, consisting of two parts separated by a slash.

The nexus name will also be used as the name of the local facet, unless overridden by the option "as". The reason for slash separator is that normally both the thread name and the nexus name parts may contain further components separated by dots, and a different separator allows to find the boundary between them. If a dot were used, in "a.b.c" it would be impossible to say, does it mean the thread "a" and nexus "b.c" in it, or thread "a.b" and nexus "c"? However "a/b.c" or "a.b/c" have no such ambiguity. Mutually exclusive with options "fromTriead" and "fromNexus".

```
fromTriead => $t
```

```
fromNexus => $n
```
    The alternative way to specify the source thread and nexus as separate options. Both options must be present or absent at the same time. Mutually exclusive with "from".

```
as => $name
```
    Specifies an override name for the local facet (and thus also for the FnReturn created in the facet). Logically similar to the SQL clause *AS*. Default is to reuse the nexus name.

```
import => $importType
```
    A string value, essentially an enum, determining how this Nexus gets imported. The supported values are the same as for `makeNexus()`, except "none", since there is no point in a no-op import:

- `reader` (or anything starting from "read") - import for reading;

- `writer` (or anything starting from "write") - import for writing.

Case-insensitive. The use of the canonical strings is recommended.

```
immed => 0/1
```
    Flag: do not wait for the thread that exported the nexus to be fully constructed. Waiting synchronizes with the exporter and prevents a race of an import attempt trying to find a nexus before it is made and failing. However if two threads are waiting for each other, it becomes a deadlock that gets caught and aborts the App. The immediate import allows to avoid such deadlocks for the circular topologies with helper threads.

The helper threads are the "blind alleys" in the topology: the "main thread" outsources some computation to a "helper thread", sending it the arguments, then later receiving the results and continuing with its logic.

With the helper threads, the parent thread will import the resul nexus of the helper as usual but the helper will import the argument nexus from the parent immediately. This can be done because the parent is required in this situation to export the argument nexus first and only then create the helper thread.

The sequence in Figure 19.1 shows it in detail.

- Thread A creates the nexus O;
- Thread A creates the helper thread B and tells it to import the nexus A/O for its input immediately and create the reverse nexus R for result;
- Thread A requests a (normal) import of the nexus B/R and falls asleep because B is not constructed yet;
    - Thread B starts running;
    - Thread B imports the nexus A/O immediately and succeeds;
    - Thread B defines its result nexus R;
    - Thread B marks itself as constructed and ready;
- Thread A wakes up after B is constructed, finds the nexus B/R and completes its import;
- Thread A can then complete its initialization, export other nexuses etc;

Figure 19.1. The use of immediate import.

Default: 0, except if importing a nexus that has been exported from the same Triead. Importing from the same Triead is not used often, since the export also imports the nexus back right away, and there is rarely any use in importing separately. But it's possible, and importing back from the same Triead is always treated as immediate to avoid deadlocks.

```
@exports = $to->exports();
```

The same as the method on the Triead, equivalent to `$to->get()->exports()`. The returned array contains the name-value pairs of the nexus names and objects.

```
@imports = $to->imports();
```

This method is different from the Triead method. It still returns an array of name-value pairs but the values are Facets (not Nexuses, as in Triead). It's a natural difference, since the facets are useful in the owner thread, and available only in it.

```
$result = $to->flushWriters();
```

Flush any data collected in the writer facets, sending them to the appropriate nexuses. The data in each facet becomes a tray that is sent to the nexus (if there was no data collected on a facet, nothing will be sent from it). Returns 1 if the flush was completed, 0 if the thread was requested to die and thus the data was discarded. The data is never sent out of a facet by itself, it always must be flushed in one of the explicit ways (`TrieadOwner::flushWriters()`, `Facet::flushWriter()`, or enqueueing a rowop on the facet's labels `_BEGIN_` and `_END_`). The flush may get stuck if this is an input-only thread and a drain is active, it will wait until the drain is released.

```
$to->requestMyselfDead();
```

Request this Triead itself to die. This is the way to disconnect from the nexuses while the thread is exiting on its own. For example, if this thread is going to dump its data before exit to a large file that takes half an hour to write, normally the data queued for this thread might fill up the queues in the nexuses, and it's a bad practice to keep the other threads stuck due to the overflowing buffers. Requesting this thread to die disconnects it from the nexuses and prevents the data from collecting. The thread could also be disconnected by marking it dead, but that would keep the harvester stuck waiting to join it while the thread completes its long write, and that's not so good either. This call provides the solution, avoiding both pitfalls.

```
$result = $to->nextXtray();
```

Process one incoming tray from a single reader nexus (any nexus where data is available, respecting the priorities). A tray essentially embodies a transaction, and "X" stands for "cross-thread". There actually is the Xtray type that represents the Tray in a special thread-safe format but it's used only inside the nexuses and not visible from outside.

If there is currently no data to process, this method will wait.

The return value is 1 normally, or 0 if the thread was requested to die. So the typical usage is:

```
while($to->nextXtray()) { ... }
```

The method `mainLoop()` encapsulates the most typical usage, and `nextXtray()` needs to be used directly only in the more unusual circumstances.

The data is read from the reverse nexuses first, at a higher priority. If any reverse nexus has data available, it will always be read before the direct nexuses. A reverse nexus typically completes a topological loop, so this priority creates the preference to cycle the data through the loop until it comes out, before accepting more data into the loop. Since all the nexuses have non-zero-length queues, obviously, there will be multiple data items traveling through the loop, in different phases, but this priority solution limits the amount of data kept in the loop's queues and allows the queue flow control to prevent too much data from entering the loop.

The raised priority of the reverse nexuses can also be used to deliver the urgent messages. Remember, there is nothing preventing you from marking any nexus as reverse (as long as it doesn't create a loop consisting of only the reverse nexuses).

The downside of having the reverse nexuses connected to a thread is that it causes an extra overhead from the check with a mutex synchronization on each `nextXtray()`. The regular-priority direct nexuses use double-buffering, with locking a

mutex only when the second buffer runs dry, and refilling it by swapping its contents with the whole collected first buffer. But the high-priority reverse nexuses have to be checked every time, even if they have no incoming data.

Within the same priority the data is processed in the round-robin order. More exactly, each refill of the double-buffering grabs the data from the first buffer of each facet and moves it to the second buffer. Then the second buffer is processed in the round-robin fashion until it runs out and another refill becomes needed.

The `nextXtray()` processes all the rowops from the incoming tray by calling them on the facet's FnReturn. Two special rowops are generated automatically even if they haven't been queued up explicitly, on the facet's labels `_BEGIN_` and `_END_` (to avoid extra overhead, they are actually generated only if there is any processing chained for them).

The `nextXtray()` automatically flushes the writers after processing a tray.

If a fatal error is encountered during processing (such as some code in a label dies), `nextXtray()` will catch the exception, discard the rest of the tray and confess itself, without flushing the writers.

```
$result = $to->nextXtrayNoWait();
```

Similar to `nextXtray()`, but returns immediately if there is no data to process. Returns 0 if there is either no input data or the thread was requested to die (the way to differentiate between these cases is to call `$to->isRqDead()`).

```
$result = $to->nextXtrayTimeout($timeout);
```

Similar to `nextXtrayNoWait()`, only if there is no data, waits for up to the length of timeout. The timeout value is floating-point seconds. Returns 0 if the timeout has expired or the thread was requested to die.

```
$result = $to->nextXtrayTimeLimit($deadline);
```

Similar to `nextXtrayNoWait()`, only if there is no data, waits until the absolute deadline. The deadline value is time since epoch in floating-point seconds, such as returned by `Triceps::now()`. Returns 0 if the wait reached the deadline or the thread was requested to die.

```
$to->mainLoop();
```

Process the incoming trays until the thread is requested to die. The exact implementation of the main loop (in C++) is:

```
void TrieadOwner::mainLoop()
{
  while (nextXtray())
    { }
}
```

19.20.3. TrieadOwner drains

The drain API of the TrieadOwner is very similar to the one in the App. The best way to do the drain is by the automatically-scoped AutoDrain class. If a drain doesn't need an automatic scoping, use the TrieadOwner API. And finally if you want to mess with drains from outside an App's Tread and thus don't have a TrieadOwner, only then use the App API.

```
$to->requestDrainShared();
$to->requestDrainExclusive();
$to->waitDrain();
$to->drainShared();
$to->drainExclusive();
$to->undrain();
$result = $to->isDrained();
```

The methods are used in exactly the same way as the similar App methods, with only the difference of the names on the shared drains.

The exclusive drains always make the exclusion for this Triead. (Only one thread can be excluded from a drain). Normally the exclusive drains should be used only for the input-only threads. They could potentially be used to exclude a non-input-only thread too but I'm not sure, what's the point, and haven't worked out if it would work reliably (it might, or it might not).

```
$result = $to->isRqDrain();
```

Check whether a drain request is active. This can be used in the threads that generate data based on the real-time clock yet aren't input-only: if they find that a drain is active, they should refrain from generating the data and go back to the waiting. There is no way for them to find when the drain is released, so they should just continue to the next timeout as usual. Such code must use `nextXtrayTimeout()` or `nextXtrayTimeLimit()` for the timeouts, or the drain would never complete. These two methods know how to handle the timeouts and also how to properly interact with the drains without causing the race conditions. The input-only threads don't have this limitation. And of course keep in mind that the better practice is to deal with the real time either in the input-only threads or by driving it from outside the model altogether.

19.20.4. TrieadOwner file interrruption

The file interruption part of the API deals with how the thread handles a request to die. It should stop its work and exit, and for the normal threads that read from nexuses, this is fairly straightforward. The difficulty is with the threads that read from the outside sources (sockets and such). They may be in the middle of a read call, with no way to tell when the next chunk of data will arrive. These long calls on the file descriptors need to be interrupted when the thread is requested to die.

The interruption is done by revoking the file descriptors (dupping a `/dev/null` into it) and sending the signal `SIGUSR2` to the thread. Even if the dupping doesn't interrupt the file operation, `SIGUSR2` does, and on restart it will find that the file descriptor now refers to `/dev/null` and return immediately. Triceps defines a `SIGUSR2` handler that does nothing, but you can override it with a custom one.

For this to work, Triceps needs to know, which file descriptors are to be revoked, which is achieved by registering them for tracking in the TrieadOwner. To avoid accidentally revoking the unrelated descriptors, the descriptors must be unregistered before closing. The normal sequence is:

- open a file descriptor;

- register it for tracking;

- do the file operations;

- unregister the file descriptor;

- close it.

The lowest-level calls deal with the raw tracking:

```
$to->trackFd($fd);
```

Register the file descriptor (obtained with `fileno()` or such) for tracking. The repeated calls for the same descriptor have no effect.

```
$to->forgetFd($fd);
```

Unregister the file descriptor. If the descriptor is not registered, the call is ignored.

The next level of the API deals with the file handles, extracting the file descriptors from them as needed.

```
$to->track(FILE);
```

Get a file descriptor from the file handle and track it.

```
$to->forget(FILE);
```

Get a file descriptor from the file handle and unregister it.

```
$to->close(FILE);
```

Unegister the file handle's descriptor then close the file handle. It's a convenience wrapper, to make the unregistering easier to remember.

The correct sequence might be hard to follow if the code involves some dying and evals catching these deaths. To handle that, the next level of the API provides the automatic tracking by scope. The scoping is done with the class Triceps::TrackedFile. Which is probably easier to describe right here.

19.20.5. TrackedFile

A TrackedFile object keeps a reference of a file handle and also knows of its file descriptor being tracked by the TrieadOwner (so yes, it has a reference to the TrieadOwner too). Until the TrackedFile is destroyed, the file handle in it will never have its reference count go to 0, so it will never be automatically closed and destroyed. And when the TrackedFile is destroyed, first it tells the TrieadOwner to forget the file descriptor and only then unreferences the file handle, preserving the correct sequence.

And the scope of the TrackedFile is controlled by the scope of a variable that keeps a reference to it. If it's a local/my variable, TrackedFile will be destroyed on leaving the block, if it's a field in an object, it will be destroyed when the object is destroyed or the field is reset. Basically, the usual Perl scope business in Triceps.

If you want to close the file handle before leaving the scope of TrackedFile, don't call `close()` on the file handle. Instead, call `close()` on the TrackedFile:

```
$trf->close();
```

This will again properly untrack the file descriptor, and then close the file handle, and remember that it has been closed, so no seconds attempt at that will be done when the TrackedFile gets destroyed.

There also are a couple of getter methods on the TrackedFile:

```
$fd = $trf->fd();
```

Get the tracked file descriptor. If the file handle has been already closed, will return -1.

```
$fh = $trf->get();
```

Get the tracked file handle. If the file handle had already been closed, will confess.

Now we get back to the TrieadOwner.

```
$trf = $to->makeTrackedFile(FILE);
```

Create a TrackedFile object from a file handle.

```
$trf = $to->makeTrackedFileFd(FILE, $fd);
```

The more low-level way to construct a TrackedFile, with specifying the file descriptor as a separate explicit argument. `makeTrackedFile()` is pretty much a wrapper that calls `fileno()` on the file handle and then calls `makeTracked-FileFd()`.

And the following methods combine the loading of a file descriptor from the App and tracking it by the TrieadOwner. They are the most typically used interface for passing around the file handles through the App.

```
($trf, $file) = $to->trackDupFile($name, $mode);
```

Load the dupped file handle from the App, create a file handle object from it according to the IO::Handle subclass received from the App and $mode (in either `r/w/a/r+/w+/a+` or `</>/>>/+</+>/+>>` format), and make a TrackedFile from

it. Returns a pair of the TrackedFile object and the created file handle. The App still keeps the original file descriptor. The `$mode` must be consistent with the original mode of the file stored into the App. `$name` is the name that was used to store the file descriptor into the App.

```
($trf, $file) = $to->trackGetFile($name, $mode);
```

Similar to `trackDupFile()` only the file descriptor is moved from the App, and the App forgets about it.

```
($trf, $file) = $to->trackDupClass($name, $mode, $class);
($trf, $file) = $to->trackGetClass($name, $mode, $class);
```

Similar to the File versions, only creates the file handle of an explicitly specified subclass of IO::Handle (per `$class`).

19.21. Nexus reference

The Nexus class is pretty much opaque. It's created and managed entirely inside the App infrastructure from a Facet, and even the public API for importing a nexus doesn't deal with the Nexus object itself, but only with its name. The only public use of the Nexus object is for the introspection and entertainment value, to see what Trieads export and import what Nexuses: pretty much the only way to get a Nexus reference is by listing the exports or imports of a Triead.

The API of a Nexus is very limited:

```
$nxname = $nx->getName();
```

Get the name of the nexus (the short name, inside the Triead).

```
$tname = $nx->getTrieadName();
```

Get the name of the Triead that exported this nexus.

```
$result = $nx->same($nx2);
```

Check whether two nexus references point to the same object.

```
$result = $nx->isReverse();
```

Check whether the nexus is reverse.

```
$limit = $nx->queueLimit();
```

Get the queue limit of the nexus.

19.22. Facet reference

A Facet represents a Nexus endpoint imported into a Triead. A facet is either a reader (reading from the nexus) or a writer (writing into the nexus).

In the Perl API the Facets are created by `TrieadOwner::makeNexus()` or `TrieadOwner::importNexus()`. After that the metadata in the Facet is fixed and is available for reading only. Of course, the rowops can then be read or written.

The reading of data from a Facet is done by `TrieadOwner::nextXtray()`. There is no way to read a tray from a particular facet, `nextXtray()` reads from all the Triead's imported reader facets, alternating in a fair fashion if more than one of them has data available.

Each Facet has an FnReturn connected to it. The reading from a reader facet happens by forwarding the incoming rowops to that FnReturn. To actually process the data, you can either chain your handler labels directly to the FnReturn labels, or push an FnBinding onto that FnReturn. An incoming Xtray is always processed as a unit, with no intermixing with the other Xtrays.

The writing to a writer facet happens by calling the labels of its FnReturn. Which then has the logic that collects all these rowops into a buffer. Then when the facet is flushed, that buffer becomes an indivisible Xtray that gets sent to the nexus as a unit, and then read by the reader facets as a unit.

The facet metadata consists of:

- a set of labels, same as for an FnReturn, used to build the facet's internal FnReturn; these labels define the data that can be carried through the nexus;

- a set of row types that gets exported through the nexus;

- a set of table types that gets exported through the nexus.

The table types must not contain any references to the Perl functions, or the export will fail. The Perl code snippets in the text format can be used instead.

There are two special labels, named _BEGIN_ and _END_. They may be defined explicitly, but if they aren't, they will be always added implicitly, with an empty row type (i.e. a row type with no fields).

When reading an Xtray, the _BEGIN_ label will always be called first, and _END_ last, thus framing the rest of the data. There are optimizations that skip the calling if there is nothing chained to these labels in FnReturn nor to the top FnBinding, and the rowop as such carries no extra data. The optimization is actually a bit deeper: the _BEGIN_ and _END_ rowops that have no extra data in them aren't even carried in the Xtray through the nexus. They are generated on the fly if there is an interest in them, or otherwise the generation is skipped.

What is meant by the "extra data"? It means, either the opcode is not OP_INSERT or there are some non-NULL fields (or both). If the _BEGIN_ and _END_ labels were auto-generated, their row type will contain no fields, so the only way to send the non-default data in them will be the non-default opcode. But if you define them explicitly with a different row type, you can also send the data in them.

When sending the data into a writer Facet, you don't have to send the _BEGIN_ and _END_ rowops, if you don't, they will be generated automatically as needed, with the default contents (opcode OP_INSERT and NULLs in all the fields). Moreover, they will really be generated automatically on the reader side, thus saving the overhead of passing them through the nexus. Another consequence of this optimization is that it's impossible to create an Xtray consisting of only a default _BEGIN_, a default _END_ and no payload rowops between them. It would be an empty Xtray, that would never be sent through the nexus. Even if you create these _BEGIN_ and _END_ rowops manually (but with the default contents), they will be thrown away when they reach the writer facet. If you want an Xtray to get through, you've got to either send the payload or put something non-default into at least one of the _BEGIN_ or _END_ rowops, at the very minimum a different opcode.

Sending the _BEGIN_ and _END_ rowops into a writer facet also has the effect of flushing it. Even if these rowops have the default contents and become thrown away by the facet, the flushing effect still works. The _BEGIN_ rowop flushes any data that has been collected in the buffer before it. The _END_ rowop gets added to the buffer (or might get thrown away) and then flushes the buffer. If the buffer happens to contain anything at the flush time, that contents forms an Xtray and gets forwarded to the nexus.

It's a long and winding explanation, but really it just does what is intuitively expected.

A Facet has two names, the *full* one and the *short* one:

- The full name is copied from the nexus and consists of the name of the thread that exported the nexus and the name of the nexus itself separated by a slash, such as "t1/nx1".

- The short name is the name with which the facet was imported. By default it's taken from the short name of the nexus. But it can also be given a different explicit name during the import, which is known as the "as-name" (because it's similar to the SQL *AS* clause). So if the full name is "t1/nx1", the default short name will be "nx1", but it can be overridden. The facet's FnReturn is named with the facet's short name.

A Facet object is returned from either a nexus creation or nexus import. Then the owner thread can work with it.

```
$result = $fa->same($fa2);
```

Check whether two references point to the same Facet object.

```
$name = $fa->getShortName();
```

Get the short name of the facet (AKA "as-name", with which it has been imported).

```
$name = $fa->getFullName();
```

Get the full name of the nexus represented by this facet. The name consists of two parts separated by a slash, "`$thread/$nexus`".

```
$result = $fa->isWriter();
```

Check whether this is a writer facet (i.e. writes to the nexus). Each facet is either a writer or a reader, so if this method returns 0, it means that this is a reader facet.

```
$result = $fa->isReverse();
```

Check whether this facet represents a reverse nexus.

```
$limit = $fa->queueLimit();
```

Get the queue size limit of the facet's nexus. For a reverse nexus the returned value will be a large integer (currently `INT32_MAX` but the exact value might change in the future). And if some different limit value was specified during the creation of the reverse nexus, it will be ignored.

```
$limit = &Triceps::Facet::DEFAULT_QUEUE_LIMIT;
```

The constant of the default queue size limit that is used for the nexus creation, unless explicitly overridden.

```
$fret = $fa->getFnReturn();
```

Get the FnReturn object of this facet. This FnReturn will have the same name as the facet's short name, and it has a special symbiotic relation with the Facet object. Its use depends on whether this is a reader or writer facet. For a writer facet, sending rowops to the labels in FnReturn (directly or by chaining them off the other labels) causes these rowops to be buffered for sending into the nexus. For a reader facet, you can either chain your logic directly off the FnReturn's labels, or push an FnBinding onto it as usual.

```
$nexus = $fa->nexus();
```

Get the facet's nexus. There is not a whole lot that can be done with the nexus object, just getting the introspection information, and the same information can be obtained directly with the facet's methods.

```
$idx = $fa->beginIdx();
```

Index (as in "integer offset", not a table index) of the _BEGIN_ label in the FnReturn's set of labels. There probably isn't much use for this method, and its name is somewhat confusing.

```
$idx = $fa->endIdx();
```

Index (as in "integer offset", not a table index) of the _END_ label in the FnReturn's set of labels. There probably isn't much use for this method, and its name is somewhat confusing.

```
$label = $fa->getLabel($labelName);
```

Get a label from FnReturn by name. This is a convenience method, equivalent to `$fa->getFnReturn()->getLabel($labelName)`. Confesses if the label with this name is not found.

```
@rowTypes = $fa->impRowTypesHash();
```

Get ("import") the whole set of row types exported through the nexus. The result is an array containing the name-value pairs, values being the imported row types. This array can be assigned into a hash to populate it. As it happens, the pairs will be ordered by name in the ASCII alphabetical order but there are no future guarantees about it.

The actual import of the types is done only once, when the nexus is imported to create the facet, and the repeated calls of the `imp*` methods will return the same objects.

```
$rt = $fa->impRowType($rtName);
```

Get ("import") one row type by name. If the name is not known, will confess.

```
@tableTypes = $fa->impTableTypesHash();
```

Get ("import") the whole set of table types exported through the nexus. The result is an array containing the name-value pairs, values being the imported table types. This array can be assigned into a hash to populate it. As it happens, the pairs will be ordered by name in the ASCII alphabetical order but there are no future guarantees about it.

The actual import of the types is done only once, when the nexus is imported to create the facet, and the repeated calls of the `imp*` methods will return the same objects.

```
$tt = $fa->impTableType($ttName);
```

Get ("import") one table type by name. If the name is not known, will confess.

```
$result = $fa-> flushWriter();
```

Flush the collected buffered rowops to the nexus as a single Xtray. If there are no collected rowops, does nothing. Returns 1 if the flush succeeded (even if there was no data to send), 0 if this thread was requested to die and thus all the collected data gets thrown away, same as for the `TrieadOwner::flushWriters()`. The rules for when this method may be called is also the same: it may be called only after calling `readyReady()`, or it will confess.

If this facet is in an input-only Tread, this call may sleep if a drain is currently active, until the drain is released.

19.23. AutoDrain reference

The AutoDrain class creates the drains on an App with the automatic scoping. When the returned AutoDrain object gets destroyed, the drain becomes released. So placing the object into a lexically-scoped variable in a block with cause the release on the block exit. Placing it into another object will cause the release on deletion of that object. And just not storing the object anywhere works as a barrier: the drain gets completed and then immediately released, guaranteeing that all the previously sent data is processed and then continuing with the processing of the new data.

All the drain caveats described in the App class apply to the automatic drains too.

```
$ad = Triceps::AutoDrain::makeShared($app);
$ad = Triceps::AutoDrain::makeShared($to);
```

Create a shared drain and wait for it to complete. A drain may be created from either an App or a TrieadOwner object. Returns the AutoDrain object.

```
$ad = Triceps::AutoDrain::makeSharedNoWait($app);
$ad = Triceps::AutoDrain::makeSharedNoWait($to);
```

Same as `makeShared()` but doesn't wait for the drain to complete before returning. May still sleep if an exclusive drain is currently active.

```
$ad = Triceps::AutoDrain::makeExclusive($to);
```

Create an exclusive drain on a TrieadOwner and wait for it to complete. Returns the AutoDrain object. Normally the excluded thread (the current one, identified by the TrieadOwner) should be input-only. Such an input-only thread is allowed to send more data in without blocking. To wait for the app become drained again after that, use the method `wait()`.

```
$ad = Triceps::AutoDrain::makeExclusiveNoWait($to);
```

Same as `makeExclusive()` but doesn't wait for the drain to complete before returning. May still sleep if a shared or another exclusive drain is currently active.

```
$ad->wait();
```

Wait for the drain to complete. Particularly useful after the `NoWait` creation, but can also be used to wait for the App to become drained again after injecting some rowops through the excluded Triead of the exclusive drain.

```
$ad->same($ad2);
```

Check that two AutoDrain references point to the same object.

Chapter 20. Triceps C++ API Reference

20.1. C++ API Introduction

Fundamentally, the C++ and Perl APIs are shaped similarly. So I won't be making a version of all the examples in C++. Please read the Perl-based documentation first to understand the spirit and usage of Triceps. The C++-based documentation is more of the reference type and concentrates on the low-level specifics and differences from Perl necessitated by this specifics.

In many cases just reading the descriptions of the methods in the `.h` files should be enough to understand the details and be able to use the API. However in some cases the class hierarchies differ, with the Perl API covering the complexities exposed in the C++ API.

Writing directly in C++ is significantly harder than in Perl, so I highly recommend sticking with the Perl API unless you have a good reason to do otherwise. Even when the performance is important, it's usually enough to write a few critical elements in C++ and then bind them together in Perl. (If you wonder about Java, and why I didn't use it instead, the answer is that Java successfully combines the drawbacks of both and adds some of its own).

The C++ Triceps API is much more sensitive to the errors. The Perl API checks all the arguments for consistency, it's principle is that the interpreter must never crash. The C++ API is geared towards the efficiency of execution. It checks for errors when constructing the major elements but then does almost no checks at run time. The expectation is that the caller knows what he is doing. If the caller sends bad data, mislays the pointers etc., the program will crash. The idea here is that most likely the C++ API will be used from another layer: either an interpreted one (like Perl) or a compiled one (like a possible future custom language). Either way that layer is responsible for detecting the user errors at either interpretation or compile time. By the time the data gets to the C++ code, it's already checked and there is no need to check it again. Of course, if you write the programs manually in C++, that checking is upon you.

The more high-level features are currently available only in Perl. For example, there are no joins in the C++ API. If you want to do the joins in C++, you have to code your own. This will change over time, as these features will solidify and move to a C++ implementation to become more efficient and available through all the APIs. But it's much easier to experiment with the initial implementations in Perl.

The C++ code is located in the `cpp/` subdirectory, and all the further descriptions refer to the subdirectories under it.

20.2. The const-ness in C++

I've been using the `const` keyword for two purposes:

- To let the compiler optimize a little better the methods that do not change the state of the objects.

- To mark the fragments of the read-only object internal state returned by the methods. This is much more efficient than making copies of them.

So if you get a `const vector<> &` returned from a method, this is a gentle reminder that you should not be modifying this vector. Of course, nothing can stop a determined programmer from doing a type cast and modifying it anyway, but be aware that such inconsistent modifications will likely cause the program to crash in the future. And if the vector contains references to other objects, these objects usually should not be modified either, even they might not be marked with const.

However all this `const` stuff is not all rainbows and unicorns but also produces a sizable amount of suffering. One consequence is that you can not use the normal iterators on the `const` vectors, you have to use the const_iterators. Another is that once in a while you get something like a `(const RowType *)` from one method and need to pass it as an argument to another method that takes a `(RowType *)`. In this case make sure that you know what you are doing and then proceed boldly with using a `const_cast`. There is just no way to get all the const-ness self-consistent without ripping it out altogether.

20.3. Memory management in the C++ API and the Autoref reference

The memory management fundamentals have been described in Section 4.3: "Memory management fundamentals" (p. 20) , and the application-level considerations have been descibed in Chapter 8: "*Memory Management*" (p. 77) . This section goes deeper into the issues specific to the C++ API.

The code related to the memory management is generally collected under `mem/`. The memory management is done through the reference counting, which has two parts to it:

- The objects that can be managed by reference counting.

- The references that do the counting.

The managed objects come in two varieties: single-threaded and multi-threaded. The single-threaded objects lead their whole life in a single thread, so their reference counts don't need locking. The multi-threaded objects can be shared by multiple threads, so their reference counts are kept thread-safe by using the atomic integers (if the NSPR library is available) or by using a lock (if NSPR is not used). That whole implementation of atomic data with or without NSPR is encapsulated in the class AtomicInt in `mem/Atomic.h`.

The way a class selects whether it will be single-threaded or multi-threaded is by inheriting from the appropriate class:

Starget

> for single-threaded (defined in `mem/Starget.h`).

Mtarget

> for multi-threaded (defined in `mem/Mtarget.h`).

If you do the multiple inheritance, the [SM]target has to be inherited only once. Also, you can't change the choice along the inheritance chain. Once chosen, you're stuck with it. The only way around it is by encapsulating that inner class's object instead of inheriting from it.

The references are created with the template `Autoref<>`, defined in `mem/Autoref.h`. For example, if you have an object of class RowType, the reference to it will be `Autoref<RowType>`. There are are some similar references in the Boost library, but I prefer to avoid the avoidable dependencies (and anyway, I've never used Boost much).

The target objects are created in the constructors with the reference count of 0. The first time the object pointer is assigned to an Autoref, the count goes up to 1. After that it stays above 0 for the whole life of the object. As soon as it goes back to 0 (meaning that the last reference to it has disappeared), the object gets destroyed. No locks are held during the destruction itself. After all the references are gone, nobody should be using it, and destroying it is safe without any extra locks.

Starget and Mtarget are independent classes but Autoref can work transparently on both of them because Autoref doesn't modify the reference counters by itself. Instead the target class is expected to provide the methods

```
void incref() const;
int decref() const;
```

They are defined as const to allow the reference counting of even the const objects, but of course the reference counter field must be mutable. `decref()` returns the resulting counter value. When it goes down to 0, Autoref calls the destructor.

An important point is that to do all this, the Autoref must be able to execute the correct destructor when it destroys the object that ran out of references. Starget and Mtarget do not provide the virtual destructors. This allows to keep the reference-counted classes non-virtual, and save a little bit of memory in their objects, which might be important for the objects

used in large numbers. If you don't use the polymorphism for some class, you don't have to use the virtual destructors. But if you do use it, i.e. create a class B inheriting from A, inheriting from [SM]target, and then assign something like

```
Autoref<A> ref = new B;
```

then the class A (and by extension all the classes inheriting from it) must have a virtual destructor to get everything working right.

It's also possible to mess up the destruction with the use of pointers. For example, look at this sequence:

```
Autoref<RowType> rt = new RowType(...);
RowType *rtp = rt; // copies a reference to a pointer
rt = NULL; // reference cleared, count down to 0, object destroyed
Autoref <RowType> rt2 = rtp; // resurrects the dead pointer, corrupts memory
```

The lesson here is that even though you can mix the references with pointers to reduce the overhead (the reference assignments change the reference counters, the pointer assignments don't), and I do it in my code, you need to be careful. A pointer may be used only when you know that there is a reference that holds the object in place. Once that reference is gone, the pointer can't be used any more, and especially can't be assigned to another reference. Be careful.

There are more varieties of the Autoref template, also defined in `mem/Autoref.h`:

- Onceref

- const_Autoref

- const_Onceref

The Onceref is an attempt at optimization when passing the function arguments and results. It's supposed to work like the standard auto_ptr: you assign a value there once, and then when that value gets assigned to an Autoref or another Onceref, it moves to the new location, leaving the reference count unchanged and the original Onceref as NULL. This way you avoid a spurious extra increase-decrease. However in practice I haven't got around to implementing it yet, so for now it's a placeholder that is defined to be an alias of Autoref, and const_Onceref is an alias of const_Autoref.

const_Autoref is a template defined over the Autoref template:

```
template <typename Target>
class const_Autoref : public Autoref<const Target>
```

The const_Autoref is a reference to a constant object. As you can see, `const_Autoref<T>` is equivalent to `Autoref<const  T>`, only it handles the automatic type casts much better. The approach is patterned after the const_iterator. The only problem with const_Autoref is that when you try to assign a NULL to it, that blows the compiler's mind. So you have to write an explicit cast of `const_Autoref<T>::null()` or `Autoref<T>::null()` or `(T*)NULL` of `(const T*)NULL` to help it out. Either way would work as long as one or the other path for casting is selected (by default the compiler is confused by the presence of two possible paths).

Finally, const_Onceref is the const version of Onceref.

The method prototypes of Autoref and all its varieties are (Target is the argument class of the Autoref template, Ptr is the pointer to it):

```
typedef Target *Ptr;
static Ptr null();
Autoref();
Autoref(Target *t);
Autoref(const Autoref &ar);
Autoref(const Autoref<OtherTarget> &ar);
Target &operator*() const;
Target *operator->() const;
```

```
Target *get() const;
operator Ptr() const; // type conversion to pointer
bool isNull() const;
Autoref &operator=(const Autoref &ar);
Autoref &operator=(const Autoref<OtherTarget> &ar);
bool operator==(const Autoref &ar);
bool operator!=(const Autoref &ar);
bool operator==(const Autoref<OtherTarget> &ar);
bool operator!=(const Autoref<OtherTarget> &ar);
void swap(Autoref &other);
```

Since the typical usage might be hard to understand from the method prototypes, the examples follow. Autoref can be constructed with or assigned from another Autoref or a pointer, or with a NULL value:

```
T *ptr;
Autoref<T> ref1(ptr);
Autoref<T> ref2(ref1);
Autoref<T> ref3; // initialized as NULL by default
ref1 = ref2;
ref1 = ptr;
ref2 = NULL; // releases the reference
ref2 = Autoref<T>::null(); // another way to assign NULL
```

The simple NULL usually works but there are cases with ambiguity, and the static method `null()` helps to create the NULL of the correct type and remove that ambiguity.

The assignments work for exactly the same type and also for assignment to any parent in the class hierarchy:

```
Autoref<Label> = new DummyLabel(...);
```

The automatic conversion to pointers works too:

```
ptr = ref1;
```

Or a pointer can be extracted from an Autoref explicitly:

```
ptr = ref1.get();
```

The dereferencing and arrow operations work like on a pointer:

```
T val = *ref1;
ref1->method();
```

The Autorefs can also be compared for equality and inequality:

```
ref1 == ref2
ref1 != ref2
```

To compare them to pointers, use `get()`. Except for one special case: the comparison to NULL happens so often that a special method is provided for it:

```
ref1.isNull()
```

Two Autorefs may swap their values, without changing the reference counts of either value:

```
ref1.swap(ref2);
```

There is also a special variety of Autoref for referring to rows, Rowref, described in Section 20.13: "Row and Rowref reference" (p. 444) .

20.4. The many ways to do a copy

The Triceps objects need occasionally to be copied. But as the objects are connected into the deep topologies, a question arises, how much of this topology needs to be copied along with the object?

The objects that are copied the most are the IndexType and TableType. The method `copy()`, in both Perl and C++, is the general way to copy them. It copies the object, and as needed its components, but tries to share the referred objects that can be shared (such as the row types). But the multithreading support required more kinds of copying.

The Perl method `TableType::copyFundamental()` copies a table type with only a limited subset of its index types, and excludes all the aggregators. It is implemented in Perl, and if you look in `lib/Triceps/TableType.pm`, you can find that it starts by making a new table type with the same row type, and then one by one adds the needed index types to it. It needs the index types without any aggregators or nested index types, and thus there is a special method for doing this kind of copies:

```
$idxtype2 = $idxtype->flatCopy();
```

The "flat" means exactly what it looks like: copy just the object itself without any connected hierarchies.

On the C++ level there is no such method, instead there is an optional argument to `IndexType::copy()`:

```
virtual IndexType *copy(bool flat = false) const;
```

So if you want a flat copy, you call

```
idxtype2 = idxtype->copy(true);
```

There is no `copyFundamental()` on the C++ level, though it probably should be, and should be added in the future. For now, if you really want it, you can make it yourself by copying the logic from Perl.

In the implementation of the index types, this argument `flat` mostly just propagates to the base class, without a whole lot needed from the subclass. For example, this is how it works in the SortedIndexType:

```
IndexType *SortedIndexType::copy(bool flat) const
{
  return new SortedIndexType(*this, flat);
}

SortedIndexType::SortedIndexType(const SortedIndexType &orig, bool flat) :
  TreeIndexType(orig, flat),
  sc_(orig.sc_->copy())
{ }
```

The base class TreeIndexType takes care of everything, all the subclass needs to do is carry the flat argument to it.

The next kind of copying is exactly the opposite: it copies the whole table type or index type, including all the objects involved, including the row types or such. It is used for passing the table types through the nexus to the other threads.

Remember that all these objects are reference-counted. Whenever a Row object is created or deleted in Perl, it increases or decreases the reference of the RowType to which that row belongs. If the same RowType object is shared between multiple threads, they will have a contention for this atomic counter, or at the very least will be shuttling the cache line with it back and forth between the CPUs. It's more efficient to give each thread its own copy of a RowType, and then it can stay stuck in one CPU's cache.

So wherever a table type is exported into a nexus, it's deep-copied (and when a row type is exported into a nexus, it's simply copied, and so are the row types of the labels in the nexus). Then when a nexus is imported, the types are again deep-copied into the thread's facet.

But there is one more catch. Suppose we have a label and a table type that use the same row type. Whenever a row coming from a label is inserted into the table (in Perl), the row types of the row (i.e. the label's row type in this case) and of the

table are checked for a match. If the row type is the same, this check is very quick. But if the same row type gets copied and then different copies are used for the label and for the table, the check will have to go and actually compare the contents of these row types, and will be much slower. To prevent this slowness, the deep copy has to be smart: it must be able to copy a bunch of things while preserving the identity of the underlying row types. If it's given a label and then a table type, both referring to the same row type, it will copy the row type from the label, but then when copying the table type it will realize that this row type had already been seen and copied, so it will reuse the same row type copy for the table type. And the same applies even within a table type: it may have multiple references to the same row type from the aggregators, and will be smart enough to figure out if they are the same, and copy the same row type only once.

This smartness stays mostly undercover in Perl. When you import a facet, it will do all the proper copying, sharing the row types. (Though of course if you export the same row type through two separate nexuses, and then import them both into another thread, these facets will not share the types between them any more). There is a method `TableType::deepCopy()` but it was mostly intended for testing and it's self-contained: it will copy one table type with the correct row type sharing inside it but it won't do the sharing between two table types. It's not even documented in the TableType reference.

All the interesting uses of the `deepCopy()` are at the C++ level. It's used all over the place: for the TableType, IndexType, AggregatorType, SortedIndexCondition and RowSetType (the type of FnReturn and FnBinding). If you decide to create your own subclass of these classes, you need to implement the `deepCopy()` as well as the normal `copy()` for it.

Its prototype generally looks like this (substitute the correct return type as needed):

```
virtual IndexType *deepCopy(HoldRowTypes *holder) const;
```

The HoldRowTypes object is what takes care of sharing the underlying row types. To copy a bunch of objects with sharing, you create a HoldRowTypes, copy the bunch, destroy the HoldRowTypes.

For example, the Facet copies the row and table types from a Nexus like this:

```
Autoref<HoldRowTypes> holder = new HoldRowTypes;

for (RowTypeMap::iterator it = nx->rowTypes_.begin();
    it != nx->rowTypes_.end(); ++it)
  rowTypes_[it->first] = holder->copy(it->second);
for (TableTypeMap::iterator it = nx->tableTypes_.begin();
    it != nx->tableTypes_.end(); ++it)
  tableTypes_[it->first] = it->second->deepCopy(holder);
```

It also uses the same holder to copy the labels. A row type gets copied through a holder like this:

```
Autoref<RowType> rt2 = holder->copy(rt);
```

For all the other purposes, HoldRowTypes is an opaque object.

A special feature is that you can pass the holder pointer as NULL, and the deep copy will still work, only it obviously won't be able to share the underlying row types. So if you don't care about sharing, you can always use NULL as an argument. It even works for the direct copying:

```
HoldRowTypes *holder = NULL;
Autoref <RowType> rt2 = holder->copy(rt);
```

Its method `copy()` is smart enough to recognize the `this` being NULL and do the correct thing.

In the subclasses, the `deepCopy()` is typically implemented like this:

```
IndexType *SortedIndexType::deepCopy(HoldRowTypes *holder) const
{
  return new SortedIndexType(*this, holder);
```

```
}

SortedIndexType::SortedIndexType(const SortedIndexType &orig, HoldRowTypes *holder) :
  TreeIndexType(orig, holder),
  sc_(orig.sc_->deepCopy(holder))
{ }
```

The wrapper passes the call to the deep-copy constructor with a holder which in turn propagates the deep-copying to all the components using their constructor with a holder. Of course, if some component doesn't have any RowType references in it, it doesn't need a constructor with a holder, and can be copied without it. But again, the idea of the `deepCopy()` it to copy as deep as it goes, without sharing any references with the original.

20.5. String utilities

Triceps has a number of small helper functions for string handling that are used throughout the code. This description includes a few forward references but you don't really need to understand them to understand these functions.

The first two are declared in `common/Strprintf.h`:

```
string strprintf(const char *fmt, ...);
string vstrprintf(const char *fmt, va_list ap);
```

They are entirely similar to `sprintf()` and `vsprintf()` with the difference that they place the result of formatting into a newly constructed string and return that string.

The rest are defined in `common/StringUtil.h`.

```
extern const string &NOINDENT;
```

The special constant that when passed to the printing of the Type (see Section 20.10: "Types reference" (p. 438)) causes it to print without line breaks. Doesn't have any special effect on Errors, there it's simply treated as an empty string.

```
const string &nextindent(const string &indent, const string &subindent, string &target);
```

Compute the indentation for the next level when printing a Type. The arguments are:

indent
    indent string of the current level.

subindent
    characters to append for the next indent level.

target
    buffer to store the extended indent string.

The passing of `target` as an argument allows to reuse the same string object and avoid the extra construction.

The function returns the computed reference: if `indent` was NOINDENT, then reference to NOINDENT, otherwise reference to `target`. This particular calling pattern is strongly tied to how things are computed inside the type printing, but you're welcome to look inside it and do the same for any other purpose.

```
void newlineTo(string &res, const string &indent);
```

Another helper function for the printing of Type, inserting a line break. The `indent` argument specifies the indentation, with the special handling of NOINDENT: if `indent` is NOINDENT, a single space is added, thus printing everything in one line; otherwise a "\n" and the contents of `indent` are added. The `res` argument is the result string, where the line break characters are added.

```
void hexdump(string &dest, const void *bytes, size_t n, const char *indent = "");
```

Print a hex dump of a sequence of bytes (at address `bytes` and of length `n`), appending the dump to the destination string `dest`. The data will be nicely broken into lines, with 16 bytes printed per line. The first line is added directly to the end of the `dest` as-is, but if `n` is over 16, the other lines will follow after "\n". The `indent` argument allows to add indentation at the start of each following line.

```
void hexdump(FILE *dest, const void *bytes, size_t n, const char *indent = "");
```

Another version, sending the dumped data directly into a file descriptor.

The next pair of functions provides a generic mechanism for converting enums between a string and integer representation:

```
struct Valname
{
    int val_;
    const char *name_;
};

int string2enum(const Valname *reft, const char *name);
const char *enum2string(const Valname *reft, int val, const char *def = "???");
```

The reference table is defined with an array of Valnames, with the last element being `{ -1, NULL }`. Then it's passed as the argument `reft` of the conversion functions which do a sequential look-up by that table. If the argument is not found, `string2enum()` will return -1, and `enum2string()` will return the value of the `def` argument (which may be NULL).

Here is an example of how it's used for the conversion of opcode flags:

```
Valname opcodeFlags[] = {
    { Rowop::OCF_INSERT, "OCF_INSERT" },
    { Rowop::OCF_DELETE, "OCF_DELETE" },
    { -1, NULL }
};

const char *Rowop::ocfString(int flag, const char *def)
{
    return enum2string(opcodeFlags, flag, def);
}

int Rowop::stringOcf(const char *flag)
{
    return string2enum(opcodeFlags, flag);
}
```

20.6. Perl wrapping for the C++ objects

The subject of this section is not a part of the C++ API as such but the connection between the C++ and Perl APIs. You need to bother about it only if you want to write more of the components in C++ and export them into Perl.

When exporting the C++ (or any compiled language) API into Perl (or into any scripting language) there are two things to consider:

1. The script must never crash the interpterer. The interpreted program might die but the interpreter itself must never crash. If you've ever dealt with `wksh` (not thankfully long dead), you know how horrible is the debugging of such crashes.

2. Perl has its memory management by reference counting, which needs to be married with the memory management at the C++ level.

The solution to the second problem is fairly straightforward: have an intermediate wrapper structure. Perl has its reference counting for the pointer to this structure. When you construct a Perl object, return the pointer to a newly allocated instance of this structure. When the Perl reference count goes down to zero, it calls the method DESTROY for this object, and then you destroy this structure.

And inside this structure will be the C++ reference to the actual C++ object. When the wrapper structure gets created, it gets a new reference and when the wrapper structure gets destroyed, it releases this reference.

Here is a small example of how the RowType object gets created and destroyed in the Perl XS code, using the wrapper structure WrapRowType:

```
WrapRowType *
Triceps::RowType::new(...)
  CODE:
    RETVAL = NULL; // shut up the warning
    try { do {
      RowType::FieldVec fld;

      clearErrMsg();

      ...
      Onceref<RowType> rt = new CompactRowType(fld);
      Erref err = rt->getErrors();
      if (err->hasError()) {
        throw Exception::f(err, "Triceps::RowType::new: incorrect data");
      }

      RETVAL = new WrapRowType(rt);
    } while(0); } TRICEPS_CATCH_CROAK;
  OUTPUT:
    RETVAL

void
DESTROY(WrapRowType *self)
  CODE:
    // warn("RowType destroyed!");
    delete self;
```

This example also shows the way the Triceps XS code deals with the conversion of the C++ Exceptions to the Perl confessions. The `while(0)` part is a hold-over from the earlier implementation of error handling that allows to set the error message directly in a buffer and then do a `break` to jump to the end of the pseudo-loop. The new way is the throwing of Exceptions. The shown code is a hybrid that handles both. The handling itself is done in the macro TRICEPS_CATCH_CROAK.

Now let's look more in-depth at the first problem. How does Perl know to call a particular XS method for a object? From the package this object is blessed to. If the package happens to be an XS package, the XS method will be called. However it's entirely possible to re-bless the object to a completely different package. If the user does this, a completely wrong method may be called and will crash when it tries to reach a wrong object at the pointer.

Things get even worse for the other arguments of the methods. For example, the argument `other` here:

```
int
equals(WrapRowType *self, WrapRowType *other)
```

Well, Perl lets you provide a code snippet for the `typemap` file that would check that the object is as expected. But how would that snippet know? Going just by the blessed package is unreliable.

Triceps solves this problem by placing an 8-byte magic code at the front of every wrap object. Each class has its own magic value for this field. 8 bytes allow to have great many unique codes, and is quick to check because it's just one CPU word.

This magic code is defined in the C++ part in `wrap/Wrap.h` as:

```
struct WrapMagic {
  ~WrapMagic()
  {
    (*(int64_t *)v_) = 0; // makes sure that it gets invalidated
  }

  char v_[8]; // 8 bytes to make a single 64-bit comparison

  bool operator!=(const WrapMagic &wm) const
  {
    return (*(int64_t *)v_) != (*(int64_t *)wm.v_);
  }
};
```

Then the wrapper is implemented as follows:

```
template<const WrapMagic &magic, class Class>
class Wrap
{
public:
  Wrap(Onceref<Class> r) :
    magic_(magic),
    ref_(r)
  { }

  // returns true if the magic value is bad
  bool badMagic() const
  {
    return magic_ != magic;
  }

  Class *get() const
  {
    return ref_.get();
  }

  operator Class*() const
  {
    return ref_.get();
  }

public:
  WrapMagic magic_;
  Autoref<Class> ref_; // referenced value
private:
  Wrap();
};
```

The check is done with the method `badMagic()`.

And the `typemap` entry is:

```
O_WRAP_OBJECT
  if( sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG) ) {
    $var = ($type)SvIV((SV*)SvRV( $arg ));
    if ($var == 0 || $var->badMagic()) {
      croakWithMsg( \"${Package}::$func_name(): $var has an incorrect magic for $ntype
\" );
    }
  } else{
    croakWithMsg( \"${Package}::$func_name(): $var is not a blessed SV reference to $ntype
\" );
```

```
    }
```

It checks that this is an object, of an XS package, and then that the pointer to the C/C++ object is not NULL, and that the value at this pointer starts with the right magic.

`croakWithMsg()` is the Triceps function for building the confession trace and then croaking with it.

The template shown above is normally used through a macro that substitutes the repeated values from one source. For example, the definition for the RowType wrapper is:

```
DEFINE_WRAP(RowType);
```

The static definition of the magic code that the macro passes to the template is defined in `wrap/Wrap.cpp`:

```
WrapMagic magicWrapRowType = { "RowType" };
```

Just make sure that the string contains no more than 7 characters.

Some objects in Triceps are reached through the special references that know both the object and its type (such as rows and row handles). For them there is a separate template and a separate macro:

```
DEFINE_WRAP2(const RowType, Rowref, Row);
DEFINE_WRAP2(Table, Rhref, RowHandle);
```

The arguments are the type class, the reference class and finally the object class itself.

And there is one more twist: sometimes the objects are self-contained but when you use them, you must use them only with a correct parent object. Right now there is only one such class: the Tray must be used with its correct Unit, and the Perl code checks it. In this case the wrapper has the reference to both Tray and the Unit, and is defined as:

```
DEFINE_WRAP_IDENT(Unit, Tray);
```

Triceps has a substantial library of helper methods and objects for wrappig the C++ objects, located in the Perl part of the code. Many of them are located in `TricepsPerl.h`. Feel free to refer to them if you need to, but for now they are out of scope of the documentation.

Just one more thing: Perl is not happy about sharing the objects between multiple threads. Even when a C++ object is OK with multithreading, Perl requires the separate wrappers to be created for it from each thread. When a new Perl thread is started, Perl can not properly ask the XS code to split the wrappers, so instead all the objects need to be invalidated in the new thread. To do that, the XS package needs to define the method `CLONE_SKIP`:

```
int
CLONE_SKIP(...)
  CODE:
    RETVAL = 1;
  OUTPUT:
    RETVAL
```

20.7. Error reporting and Errors reference

When building some kind of a language, the complicated errors often need to be reported. Often there are many errors at a time, or an error that needs to be tracked through multiple nested layers. And then these error messages need to be nicely printed, with the indentation by nested layers. Think of the errors from a C++ compiler. Triceps is a kind of a language, so it has a class to represent such errors. It hasn't propagated to the Perl layer yet and is available only in the C++ API.

The class is Errors, defined in `common/Errors.h`, and inheriting from Starget (for single-threaded reference counting). The references to it are used so often, that `Autoref<Errors>` is defined to have its own name `Erref` (yes, that's double `r`, not triple), as well as a couple of helper methods on it.

Triceps also has the exceptions (more on them in Section 20.8: "Exception reference" (p. 435)) which also contain the Errors objects inside them.

In general an Errors object contains messages, not all of which have to be errors. Some might be warnings. But in practice is has turned out that without a special dedicated compile stage it's hard to report the warnings. Even when there is a special compile stage, and the code gets compiled before it runs, as it was in Aleri, with the warnings written to a log file, still people rarely pay attention to the warnings. You would not believe, how may people would be calling support while the source of their problem is clearly described in the warnings in the log file. Even in C/C++ it's difficult to pay attention to the warnings. I better like the approach of a separate lint tool for this purpose: at least when you run it, you're definitely looking for warnings.

Because of this, the current Triceps approach is to not have warnings. If something looks possibly right but suspicious, report it as an error but provide an option to override that error (and tell about that option in the error message).

In general, the Errors are built of two kinds of information:

- the error messages;

- the nested Errors reports.

More exactly, an Errors contains a sequence of elements, each of which may contain a string, a nested Errors object, or both. When both, the idea is that the string gives a high-level description and the location of the error in the high-level object while the nested Errors digs into the further detail. The string gets printed before the nested Errors. The nested Errors get printed with an indentation. The indentation gets added only when the errors get "printed", i.e. the top-level Errors object gets converted to a string. Until then the elements may be nested every which way without incurring any extra overhead.

Obviously, you must not try to nest an Errors object inside itself, directly or indirectly. Not only will it create a memory reference cycle, but also an endless recursion when printing.

The most typical way to construct an Errors object is with the functions taking the printf-like arguments. They have been a relatively recent addition, so there is plenty of code in Triceps that uses the older ways, but this is the latest and greatest, and recommended for all the future use.

The consistent theme in its usage is "check if the Errors reference (Erref) is NULL, if it is, allocate a new Errors, and then add a formatted error message to it". So the printf-like methods are defined not on Errors but on Erref. They check if the Erref object is NULL, allocate a new Errors object into it if needed, and then format the arguments. The simplest one is:

```
void f(const char *fmt, ...);
```

It adds a simple formatted message, always marked as an error. It is used like this:

```
Erref e; // initially NULL by default
...
e.f("a message with integer %d", n);
```

The message may be multi-line, it will be split appropriately, to make it all render with the correct indenting level. If the error message contains a "\n" at the end, that "\n" will be discarded (since the line feed after each line is implicit).

The next one is used to build the nested error messages:

```
bool fAppend(Autoref<Errors> clde, const char *fmt, ...);
```

It first checks that the child errors object is not NULL and contains an error, and if it does then it goes through the dance of allocating a new Errors object if needed, appending the formatted message and the child errors. The message goes before the child errors, unlike the method signature. So you can use it blindly like this to do the right thing:

```
Erref checkSubObject(int idx);
```

```
Erref checkObject()
{
  Erref err; // initially NULL by default
  ...
  for (int i = 0; i < sz; i++)
    err.fAppend(checkSubObject(i),  "error in the sub-object %d:", i);
  ...
  return err; // NULL on success, Errors reference on error
}
```

No additional **if**s are necessary to check the result of `checkSubObject()`, `fAppend()` contains this **if** inside it. Similar to `f()`, the error message may be multi-line.

The more basic way to create an Errors is by calling one of the constructors:

```
Errors(bool e = false);
Errors(const char *msg);
Errors(const string &msg);
Errors(const string &msg, Autoref<Errors> clde);
```

For example:

```
Erref err = Errors("error message");
```

The default constructor creates an empty object, which then needs to be filled with the messages (or not, returning an empty Errors object is also possible, meaning that no errors were found, though returning NULL is more efficient). The argument `e` is an indicator than it contains an actual error. By default it's unset but will be set when an error message is added, or when a nested Errors object with an error in it is added.

The rest of the above are the convenience constructors that make one-off Errors from one element (though of course more elements can be added afterwards). In all of them the error flag is always set, and the multi-line messages are properly handled.

After an Errors object is constructed, more elements can be added to it by one of the following methods.

```
void appendMsg(bool e, const string &msg);
```

The argument `e` shows whether the message is an error message (and it will set the error flag in Errors), otherwise just a warning. There are no special per-message indications whether it's an error or warning, the Errors keeps just a single flag per object, and the messages are eventually printed as-is. So if you want the user to recognize some message as a warning, prefix its text with something like "warning:".

The argument `msg` must be a single-line message, don't use a "\n" in it! If you want to append multiple lines, call `appendMsg()` once per line, or use `appendMultiline()` to append the whole multi-line message.

```
void appendMultiline(bool e, const string &msg);
```

The same as `appendMsg()`, only it will safely break a multi-line message into multiple single-liners and will ignore the "\n" at the end. Inside the Errors object each line will become a separate message.

```
bool append(const string &msg, Autoref<Errors> clde);
```

The `append()` method is somewhat like `Erref::fAppend()` but different, because they represent different ages of the API. When `append()` was written, the warnings were considered important, and a common way to return a no-error condition was by returning an Errors object with no contents. When `fAppend()` was written, I've pretty much given up on the use of warnings, and the preferred way to return the no-error condition had become by returning NULL.

In `append()` the following situations are possible:

- The argument `clde` is NULL or empty (that is contains no messages nor sub-objects), and the error flag in it is false. In this case `append()` does nothing, and the message `msg` is thrown away.

- The argument `clde` is not empty but the error flag is false. Both `msg` and child object will be appended to the Errors.

- The argument `clde` is not empty and the error flag is true. Both `msg` and child object will be appended to the Errors, and the error flag will be set in it.

- The argument `clde` is empty and the error flag is true. The `msg` will be appended to the Errors and the error flag will be set in it. The child object will be thrown away (it's reference-counted, so it will be freed when the last reference is gone).

Just like `appendMsg()`, the argument `msg` must be a single-line message.

The return value of `append()` will be true if the child element contained any data in it or an error indication flag. This can be used together with another method

```
void replaceMsg(const string &msg);
```

to add a complex high-level description if a child element has reported an error. By now it has become old-fashioned, and `Erref::fAppend()` is a better choice. Before `fAppend()` came along, the typical way to create an error message was by calling `strprintf()` that formats a printf-like string into a C++ string object. And it's kind of expensive to call `strprintf()` in any case, even if the child returned no error. So the optimization was to call `append()` with an empty string, and if it happened to actually append something, replace that empty message with the real one. As the following pattern shows:

```
Erref clde = someThing(...);
if (e.append("", clde)) {
    // ... generate msg in some complicated way
    e.replaceMsg(strprintf(...));
}
```

The `replaceMsg()` replaces the string portion of the last element, which owns the last added child error.

This pattern can still be used if the error message is truly difficult to generate, but usually `fAppend()` is a better choice.

It's also possible to include the contents of another Errors directly, without nesting it:

```
bool absorb(Autoref<Errors> clde);
```

The return value has the same meaning as with `append()` (but `replaceMsg()` can not be used with this method for anything reasonable). The argument `clde` may be NULL. The error flag propagates to the parent as usual.

Finally, an Errors object can be cleared to its empty state:

```
void clear();
```

Usually the clearing of an Errors object is not such a good idea. Remember, they are built into a tree. So if you later modify an Errors object, that would also modify what is seen by all the objects that refer to it. The proper approach is to treat an Errors object as append-only while it's being built. And after it has been completed and returned, treat it as read-only.

To get the number of elements in Errors, use

```
size_t size() const;
```

Remember, a child object with its associated top-level error message counts as a single element, not two of them.

However the more typical methods are:

```
bool isEmpty();
```

```
bool hasError();
```

They check whether there is nothing at all or whether there is an error. The special convenience of these methods is that they can be called on NULL pointers. Quite a few Triceps methods return a NULL Erref if there was no error. In the following example, even if er is NULL, these calls are still safe and officially supported. But **not** er->size().

```
Erref er = NULL;
er->isEmpty();
er->hasError();
parent_er->append(msg, er);
parent_er->absorb(er);
parent_er.fAppend(er, msg);
```

The data gets extracted from Erref by converting it to a string, either appending to an existing string, or creating a new one:

```
void printTo(string &res, const string &indent = "", const string &subindent = "  ");
string print(const string &indent = "", const string &subindent = "  ");
```

The res argument provides the result string to append the error messages to. print() works by constructing a new string internally, calling printTo() on it, and then returning it, so print() is more expensive than printTo().

The messages will be separated by the newline characters "\n", with the proper indentation after each of these characters. The indent argument specifies the initial indentation, subindent the additional indentation for each level.

The very first line will have nothing prepended to it, it's assumed that you've already done that indentation manually, or maybe you're appending to the end of some ongoing line. But all the following top-level lines will have the indent text following the "\n". The nested error lines will have the concatenation of indent and one copy of subindent in front of them after "\n", the second level of nesting will have an indent and two copies of subindent, and so on.

All the lines are terminated by "\n" (except if the Errors object is empty, then nothing will be appended).

Note that the constant NOINDENT has no special effect on the printing of Errors (unlike the printing of types), it's just treated here as an empty string.

20.8. Exception reference

There are different ways to report the errors. Sometimes a function would return a false value. Sometime it would return an Erref with an error in it. And there is also a way to throw the exceptions.

In general I don't particularly like the exceptions. They tend to break the logic in the unexpected ways, and if not handled properly, mess up the multithreading. The safe way of working with exceptions is with the scope-based variables. This guarantees that all the allocated memory will be freed and all the locked data will be unlocked when the block exits, naturally or on an exception. However not everything can be done with the scopes, and this results in a whole mess of try-catch blocks, and a missed catch can mess up the program in horrible ways.

However sometimes the exceptions come handy. They have been a late addition to the Triceps version 1.0 and they have blossomed in the version 2.0.

The overall approach in Triceps is that the exceptions are used for the things that Should Never Happen in a correct program, or in other words for the substantially fatal events. If the user attempts to do something that can't be executed, this qualifies for an exception. Essentially, use the exceptions for the things that qualify for the classic C abort() or assert(). The idea is that at this point we want to print an error message, print the call stack the best we can, and dump the core for the future analysis.

Why not just use an abort() then? In the C++ code you certainly can if you're not interested in the extra niceties provided by the exceptions. In fact, that's what the Triceps exceptions do by default: when you construct an exception, it prints a log message, the Triceps scheduler stack trace (the sequence of the label calls that led to the code throwing the exception) and

the C++ stack trace (using a nice feature of glibc), then calls `abort()`. The error output gives the basic idea of what went wrong and the rest can be found from the core file created by `abort()`.

However remember that Triceps is designed to be embedded into the interpreted (or compiled) languages. When something goes wrong inside the Triceps program in Perl, you don't want to get a core dump of the Perl interpreter. An interpreted program must never ever crash the interpreter. You want to get the error reported in the Perl `die()` or its nicer cousin `confess()`, and possibly intercept it in `eval`. So the Perl wrapper of the Triceps library changes the mode of Triceps exceptions to actually throw the C++ exceptions instead of aborting. Since the Perl code is not really interested in the details at the C++ level, the C++ stack trace in this case is configured not to be included into the text of the exception. The Triceps scheduler trace is still included. Eventually the XS interface does an analog of `confess()`, including the Perl stack trace. When the code goes through multiple layers of Perl and C++ code (Perl code calling the Triceps scheduler, calling the label handlers in Perl, calling the Triceps scheduler again etc.), the whole layered sequence gets nicely unwound and reported. However the state of the scheduler suffers along the way: all the scheduled rowops get freed when their stack frame is unwound, so prepare to repair the state of your model if you catch the exception. Most of the time you don't want to catch it, just let it propagate and let the program die with the error message. After all, it's still a substantially fatal event, even in Perl, just the fatality is translated to the Perl level.

If you are willing to handle the exceptions (for example, if you add elements dynamically by user description and don't want the whole program to abort because of one faulty description), you can do the same in C++. Just disable the abort mode for the exceptions and catch them. Of course, it's even better to catch your exceptions before they reach the Triceps scheduler, since then you won't have to repair the state.

The same feature comes handy in the unit tests: when you test for the detection of a fatal error, you don't want your test to abort, you want it to throw a nice catchable exception.

After all this introductory talk, on to the gritty details. The class is Exception (as usual, in the namespace Triceps or whatever custom namespace you define as TRICEPS_NS), defined in `common/Exception.h`. Inside it contains an Erref with the errors. The Exception is not intended for the reference counting, so it doesn't inherit from either of [SM]target.

An Exception can be constructed in multiple ways. Just like Errors, the most typical modern way is to use the static factory methods that perform the printf-line formatting.

```
static Exception f(const char *fmt, ...);
static Exception fTrace(const char *fmt, ...);
```

The difference between them is whether the C++ stack trace will ever be included. With `Exception::fTrace()` the C++ stack trace will be added to the messages, if it is otherwise permitted by the Exception modes (see the descriptions of the flags below). With `Exception::f()`, the stack trace definitely won't ever be added. Why would you want to not add the stack trace? Two reasons:

- If you're throwing the Exception from the XS code, you know in advance that the user is interested only in the Perl stack trace, not in the C++ one. So there is no point in adding the trace. This is the case for `Exception::f()`.

- If you catch an Exception, add some information to it and re-throw a new Exception. The information from the original Exception will already contain the full stack trace, so there is no need to include the partial stack trace again. This case is not applicable to `Exception::f()` but I'll show the other methods where it's used.

These factory methods are used like this:

```
throw Exception::f("a message with integer %d", n);
```

And the similar methods for construction with the nested errors, from an Erref or another Exception:

```
static Exception f(Onceref<Errors> err, const char *fmt, ...);
static Exception fTrace(Onceref<Errors> err, const char *fmt, ...);
static Exception f(const Exception &exc, const char *fmt, ...);
```

Unlike the Erref method, these work unconditionally (since their result is normally used in `throw`, and it's too late to do anything by that time), so you better make sure in advance that there is a child error. A typical usage would be like this:

```
try {
  ...
} catch (Exception e) {
  throw Exception::f(e.getErrors(), "error at stage %d:",  n);
}

// or more directly

try {
  ...
} catch (Exception e) {
  throw Exception::f(e, "error at stage %d:",  n);
}
```

The methods that use Errors are available as `f()` and `fTrace()` while the method for a direct re-throwing of an Exception exists only as `f()` because if any stack tracing was to be done, it would be already done in the original Exception.

Just like the Errors methods, in the resulting Exception the message goes before the nested errors.

Also an Exception can be constructed by absorbing an Erref instead of wrapping it:

```
explicit Exception(Onceref<Errors> err, bool trace);
```

The `trace` flag works similarly to the difference between `f()` and `fTrace()`. The Errors object is remembered by reference, so changing it later will change the contents of the Exception.

```
explicit Exception(const string &err, bool trace);
```

A convenience constructor to make a simple string with the error. Internally creates an Errors object with the string in it.

```
explicit Exception(Onceref<Errors> err, const string &msg);
explicit Exception(Onceref<Errors> err, const char *msg);
explicit Exception(const Exception &exc, const string &msg);
```

The older ways of wrapping a nested error with a descriptive message and re-throwing it. These constructors get called inside `f()` and `fTrace()`, but the new tradition is not to use them in the user code, use `f()` and `fTrace()` instead.

```
virtual const char *what();
```

The usual, returns the text of the error messages in the Exception.

```
virtual Errors *getErrors() const;
```

Return the Errors object from the Exception.

The modes I've mentioned before are set with the class static variables:

```
static bool abort_;
```

Flag: when attempting to create an Exception, instead print the message on stderr and abort. This behavior is more convenient for debugging of the C++ programs, and is the default one. Also forces the stack trace in the error reports. The interpreted language wrappers should reset it to get the proper exceptions. Default: true.

```
static bool enableBacktrace_;
```

Flag: enable the backtrace if the constructor requests it. The interpreted language wrappers should reset it to remove the confusion of the C++ stack traces in the error reports. Default: true.

20.9. Initialization templates

Triceps has the common approach for building the complex objects in stages by the chained mehod calls (demonstrated here on a fictional class Object):

```
Autoref<Object> o = Object::make()->addOption1(arg)->addOption2(arg);
```

Here `Object::make()` is a convenience wrapper for `new Object`, because the operator `new` has an inconvenient priority. The `new` or `make()` returns a pointer to the newly constructed object, and then each method in the chain returns the same pointer (`this`, from its standpoint) to facilitate the chaining.

While the chain executes, the pointer stays a simple pointer, not a reference. So the methods in the chain can't throw any exceptions, or the memory will leak. Instead they collect the error messages in an Errors object that has to be checked afterwards, like:

```
if (o->getErrors()->hasError()) ...
```

Note that a reference to the object gets created first, so that on an error the object would be properly destroyed.

The convenience template `checkOrThrow()` allows to do the check along with the chain, and if an error is found, convert it to an Exception:

```
Autoref<Object> o = checkOrThrow(
  Object::make()->addOption1(arg)->addOption2(arg)
);
```

It does all the right things with the references.

Some objects need to be initialized after all the options have been set, since it's much easier to check things once and get the interaction of the options right rather than check on every option. And since the initialization might create the references to the object, to get it right, it has to be done after the "main" reference is created.

```
Autoref<Object> o = Object::make()->addOption1(arg)->addOption2(arg);
o->initialize();
```

The template `initialize()` allows to do it in one expression:

```
Autoref<Object> o = initialize(
  Object::make()->addOption1(arg)->addOption2(arg)
);
```

For some objects the initialization can't fail (nor any other errors can be created by the options). For the others, the errors needs to be checked afterwards, in the same way as shown above. The template `initializeOrThrow()` takes care of the whole sequence of the initialization, the check, and Exception throwing on errors:

```
Autoref<Object> o = initializeOrThrow(
  Object::make()->addOption1(arg)->addOption2(arg)
);
```

That's basically it. All these templates are defined in `common/Initialize.h`.

20.10. Types reference

Fundamentally, Triceps is a language, even though it is piggybacking on the other languages. And as in pretty much any programming language, pretty much anything in it has a type. Only the tip of that type system is exposed in the Perl API, as the RowType and TableType. But the C++ API has the whole depth. The types go all the way down to the simple types of the fields.

The classes for types are generally defined in the subdirectory `type/`. The class Type, defined in `type/Type.h` is the common base class, inheriting from Mtarget. The types are gererally fullly constructed in one thread, and then they become read-only and accessible from multiple threads.

The base class has a field that identifies, which kind of a type it is. Each subclass, every kind of type has its entry in the enum TypeId:

```
TT_VOID, // no value
TT_UINT8, // unsigned 8-bit integer (byte)
TT_INT32, // 32-bit integer
TT_INT64,
TT_FLOAT64, // 64-bit floating-point, what C calls "double"
TT_STRING, // a string: a special kind of byte array
TT_ROW, // a row of a table
TT_RH, // row handle: item through which all indexes in the table own a row
TT_TABLE, // data store of rows (AKA "window")
TT_INDEX, // a table contains one or more indexes for its rows
TT_AGGREGATOR, // user piece of code that does aggregation on the indexes
TT_ROWSET, // an ordered set of row types
```

Most of it is straightforward, and maps directly to the types described in the Perl API. TT_ROWSET is not directly exported to Perl, but internally and in the C++ API it's the underlying type for anything that uses a set of row types: an FnReturn, FnBinding, Nexus or Facet. TT_VOID is pretty much a placeholder, in case if a void type would be needed later.

The TypeId gets hardcoded in the constructor of every Type subclass. It can be obtained back with the method

```
TypeId getTypeId() const;
```

Another method finds out if the type is the simple type of a field:

```
bool isSimple() const;
```

It would be true for the types of ids TT_VOID, TT_UINT8, TT_INT32, TT_INT64, TT_FLOAT64, TT_STRING.

Generally, you can check the TypeId and then cast the Type pointer to its subclass. All the simple types have the common base class SimpleType, which is described in Section 20.11: "Simple types reference" (p. 440) .

There is also a static Type method that finds a simple type object by the type name (like "int32", "string" etc.):

```
static Onceref<const SimpleType> findSimpleType(const char *name);
```

If the type name is incorrect and the type is not found, findSimpleType() will return NULL.

There is not a whole lot of point in having many copies of the simple type objects (though if you want, you can). So there is one common copy of each simple type that can be found by name with findSimpleType(). If the type is known at the compilation time of the C++ program, you can even avoid the look-up and refer to these objects directly:

```
static Autoref<const SimpleType> r_void;
static Autoref<const SimpleType> r_uint8;
static Autoref<const SimpleType> r_int32;
static Autoref<const SimpleType> r_int64;
static Autoref<const SimpleType> r_float64;
static Autoref<const SimpleType> r_string;
```

All of them are defined in the class Type, so you refer to them like Type::r_string.

The type construction may cause errors. The construction is usually done either by a single constructor call with all the needed arguments, or a simple constructor, then additional methods to add the information in bits and pieces, then an initialization method. In both cases there is a problem of how to report the errors. They're not easy to return from a constructor and a pain to check in the bit-by-bit construction.

Instead the error information gets internally collected in an Errors object, and can be read after the construction and/or initialization is completed:

```
virtual Eref getErrors() const;
```

There also are the convenience templates that do the initialization, check the errors and throw an Exception on failure. They are described in Section 20.9: "Initialization templates" (p. 437) .

A type with errors may not be used for anything other than reading the errors (and of course it can be destroyed by deleting all the references to it). Its behavior for anything else is undefined and may cause a crash.

The rest of the common virtual methods has to do with the type comparison and print-outs. The comparison methods essentially check if two type objects are aliases for each other:

```
virtual bool equals(const Type *t) const;
virtual bool match(const Type *t) const;
```

The concept has been previously described with the Perl API in Section 5.3: "Row types equivalence" (p. 29) , but it really applies not only to the row types but to any types. The equal types are exactly the same. The matching types are the same except for the names of their elements, so it's generally safe to pass the values between these types.

`equals()` is also available as `operator==`.

The print methods create a string representation of a type, used mostly for the error messages. There is no method to parse this string representation back, at least not yet.

```
virtual void printTo(string &res, const string &indent = "", const string &subindent = "
 ") const = 0;
string print(const string &indent = "", const string &subindent = "  ") const;
```

`printTo()` appends the information to an existing string. `print()` returns a new string with the message. `print()` is a wrapper around `printTo()` that creates an empty string, does `printTo()` into it and returns it.

The printing is normally done in a multi-line format, nicely indented, and the arguments `indent` and `subindent` define the initial indent level and the additional indentation for every level.

There is also a way to print everything in one line: pass the special constant NOINDENT (defined in `com-mon/StringUtil.h`) in the argument `indent`. This is similar to using an **undef** for the same purpose in the Perl API.

The definitions of all the types are collected together in `type/AllTypes.h`.

20.11. Simple types reference

The simple types are defined as instances of the abstract class SimpleType, defined in `type/SimpleType.h`. They have one method in addition to the base Type:

```
int getSize() const;
```

It returns the size of the value of this type. For void it's 0, for string 1 (the minimal size of an empty string, consisting of only a \0 at the end), for the rest of them it's a sizeof. This size is used to extract the values from and copy the values to the compact row format.

For now this is the absolute minimum of information that makes the data usable. The list of methods will be extended over time. For example, the methods for value comparisons will eventually go here. And if the rows will ever hold the aligned values, the alignment information too.

The classes for all the actual simple types are defined in `type/AllSimpleTypes.h`:

```
VoidType
```

```
Uint8Type
Int32Type
Int64Type
Float64Type
StringType
```

You can construct the new objects of these classes but usually the pre-created shared objects are easier to use, as described in Section 20.10: "Types reference" (p. 438). The shared objects also make the type comparisons for equality and match more efficient, since the comparison gets short-circuited at the object pointer stage.

20.12. RowType reference

In the Perl API a row type is a collection of fields. Under the hood the things are more complicated. In the C++ API Triceps allows for more flexibility, more ways to represent a row. The row type is represented by the abstract base class RowType that tells the logical structure of a row and by its concrete subclasses that define the concrete layout of data in a row. To create, read or manipulate a row, all you need to know is a reference to RowType. It would refer to a concrete row type object, and the concrete row operations are accessed by the virtual methods. But when you create a row type, you need to know the concrete row type subclass.

Currently the choice is easy: there is only one such concrete subclass CompactRowType. The "compact" means that the data is stored in the rows in a compact form, one field value after another, without alignment. Perhaps some day there will be an AlignedRowType, allowing to read the values more efficiently. Or perhaps some day there will be a ZippedRowType that would store the data in the compressed format.

Naturally, RowType is a Mtarget, since it inherits from Type, and CompactRowType inherits from it. They are defined in `type/RowType.h` and `type/CompactRowType.h` but for the include purposes it's more customary to simply include `type/AllTypes.h`.

You would never use the RowType constructor directly, it's called from the subclasses. Even if you care only about the logical structure of a row but not representation, you still can't directly construct a RowType because it's an abstract class. But just construct any concrete subclass, say CompactRowType (since it's the only one available at the moment anyway), and then use its logical structure. Every subclass is expected to define a similar constructor:

```
RowType(const FieldVec &fields);
CompactRowType(const FieldVec &fields);
```

The FieldVec is the definition of fields in the row type. It's defined as simple as:

```
typedef vector<Field> FieldVec;
```

An important side note is that the field is defined within the RowType, so it's really RowType::FieldVec and RowType::Field, and you need to refer to them in your code by this qualified name. So, to create a row type, you create a vector of field definitions first and then construct the row type from it. You can throw away or modify that vector afterwards.

As usual for Type, the constructor arguments might not be correct, and any errors will be remembered and returned with `getErrors()`. Don't use a type with errors (other than to read the error messages from it, and to destroy it), it might cause your program to crash.

A Field consists of the basic information about it: the name, the type, and the array indication (remember, a Triceps field may contain an array). The array indication is either RowType::Field::AR_SCALAR for a scalar value or RowType::Field::AR_VARIABLE for a variable-sized array. The original plan was also to use the integer values for the fixed-sized array fields, but in reality the variable-sized array fields have turned out to be easier to implement and that was it. So don't use the integer values. Most probably they would work like the same variable-sized arrays but they haven't been tested, and something somewhere might crash. Use the symbolic enum AR_*.

The normal Field constructor provides all this information:

```
Field(const string &name, Autoref<const Type> t, int arsz = AR_SCALAR);
```

Or you can use the default constructor and later change the fields in the Field (or of course read them) as you please, they are all public:

```
string name_;
Autoref <const Type> type_;
int arsz_;
```

Or you can assign them later in one fell swoop:

```
void assign(const string &name, Autoref<const Type> t, int arsz = AR_SCALAR);
```

Note that even though theoretically you can define a field of any Type, in practice it has to be of a SimpleType, or the RowType constructor will return an error later. Why isn't it defined as an `Autoref<SimpleType>` then? The grand plan for the future is to allow some more interesting data structures in the rows, and this keeps the door open. In particular, the rows will be able to hold references to the other rows, just I haven't got to implementing it yet.

Once again, a RowType constructor makes a copy of the FieldVec for its use, so you can modify or destroy the original FieldVec right away. You can get back the information about the fields in RowType:

```
const vector<Field> &fields() const;
```

It returns a reference directly to the FieldVec contained in the row type, so you must never modify it! The const-ness gives a reminder about it.

There are more row type constructors (but no default one). First, each subclass variety is supposed to be able to construct its variety by copying the logical structure of any RowType:

```
CompactRowType(const RowType &proto);
CompactRowType(const RowType *proto);
```

The version with the pointer argument also works for passing an `Autoref<RowType>` as the argument which gets automatically converted to a pointer (but be careful to not destroy the last reference until after the constructor returns). And it's really the more typically used one than the &-reference version.

The resulting type will have the same logical structure but possibly a different concrete representation than the original.

The second constructor variety is a factory method:

```
virtual RowType *newSameFormat(const FieldVec &fields) const;
```

It combines the representation format from one row type and the arbitrary logical structure (the fields vector) from possibly another row type. Or course, until more of the concrete type representations become available, its use is largely theoretical.

And there is also a factory method that copies the current row type together with its concrete representation (it's really a convenience wrapper over `newSameFormat()`):

```
RowType *copy() const;
```

Let's get to some examples of constructing the row types. To reiterate, you don't construct the objects of RowType class itself, it's an abstract class. You construct the objects of the concrete subclass(es), specifically CompactRowType. Make a vector describing the fields and do the construction.

You can make the vector by either starting with an empty one and adding the fields to it or allocating a vector of the right size in advance and setting the fields in it:

```
RowType::FieldVec fields1;
fields1.push_back(RowType::Field("a", Type::r_int64)); // scalar by default
```

```
fields1.push_back(RowType::Field("b", Type::r_int32, RowType::Field::AR_SCALAR));
fields1.push_back(RowType::Field("c", Type::r_uint8, RowType::Field::AR_VARIABLE));

RowType::FieldVec fields2(2);
fields2[0].assign("a", Type::r_int64); // scalar by default
fields2[1].assign("b", Type::r_int32, RowType::Field::AR_VARIABLE);
```

You can also reuse the same vector and clean/resize is as needed to create more types.

If you're used to laying out the C structures placing the larger elements first for the more efficient alignment, know that this is not needed for the Triceps rows. The CompactRowType stores the row data unaligned, so any field order will result in the same size of the rows. And it can't make use of some fields happening to be aligned either.

You can also find the simple types by their string names:

```
fields1.push_back(RowType::Field("d", Type::findSimpleType("uint8"),
 RowType::Field::AR_VARIABLE));
```

If the type name is incorrect and the type is not found, `findSimpleType()` will return NULL, which NULL will be caught later at the row type creation times. Note that there is no automatic look-up of the array types. You can't simply pass "uint8[]" to `findSimpleType()`. You have to break it up into the simple type name as such an the array indication, like is done in `perl/Triceps/RowType.xs`. This would probably a good thing to add to RowType::Field in the future.

You can't use the type Type::r_void for the fields, it will be reported as an error.

After the fields array is created, create the row type:

```
Autoref<RowType> rt1 = new CompactRowType(fields1);
if (rt1->getErrors()->hasError())
  throw Exception(rt1->getErrors(), true);
```

Or using the inialization templates:

```
Autoref<RowType> rt1 = checkOrThrow(new CompactRowType(fields1));
```

You could also use `Autoref<CompactRowType>` but there isn't any point to it, since all the methods of CompactRowType are virtuals inherited from RowType.

Don't forget to check that the constructed type has no errors, and bail out if so. Throwing an Exception is a convenient way to abort with a nice error message, and the template `checkOrThrow()` takes care of wrapping all the details.

The RowType and its subclasses are immutable after construction, so they can be shared between threads all you want. Depending on your approach to the threads, it might be more efficient to create a separate copy of the row type for each thread. This way when the references to the row types are created and deleted, the reference count can stay in the cache of one CPU and thus be more efficient. The thread support of the Triceps library does use this approach and creates the copies whenever the row types are imported from a nexus.

There are multiple equivalent ways to create a copy:

```
Autoref<RowType> rt2 = rt1->copy();
Autoref<RowType> rt3 = rt1->newSameFormat(rt1->fields());
Autoref<RowType> rt4 = new CompactRowType(rt1);
```

Checking the errors after the copy creation is optional if the original type was correct.

If you want to extend a type with more fields, make a copy of its fields and extend it:

```
RowType::FieldVec fields3 = rt1->fields();
fields3.push_back(RowType::Field("z", Type::r_string));
Autoref<RowType> rt3 = checkOrThrow(new CompactRowType(fields3));
```

That's about it for the RowType construction.

The information about the contents of a RowType can be read back:

```
int fieldCount() const;
const vector<Field> &fields() const;
int findIdx(const string &fname) const;
const Field *find(const string &fname) const;
```

`fields()` had already been described. `fieldCount()` returns the count of fields. `findIdx()` finds the index of the field by name, so that it can then be looked up in the result of `fields()`. Or -1 if there is no such field. `find()` directly returns the pointer to the field by name, combining these two actions. (Or it returns NULL if there is no such field).

The rest of the RowType methods have to do with the manipulation of the rows. They are described in Section 20.13: "Row and Rowref reference" (p. 444) .

20.13. Row and Rowref reference

The class Row implements a data row and is fundamentally an opaque buffer. You can't do anything with it directly other than having a pointer to it. You can't even delete a Row object using that pointer. To do anything with a Row, you have to go through that row's RowType. There are some helper classes, like CompactRow, but you don't need to concern yourself with them: they are helpers for the appropriate row types and are never used directly.

That opaque buffer is internally wired for the reference counting, of the Mtarget multithreaded variety. The rows can be passed and shared freely between the threads. No locks are needed for that (other than in the reference counter), the thread safety is achieved by the rows being immutable. Once a row is created, it stays the same. If you need to change a row, just create a new row with the new contents. Basically, it's the same rules as in the Perl API.

The tricky part in the C++ API is that you can't simply use an `Autoref<Row>` for rows. As described above, it won't know how to destroy the Row when its reference counter goes to zero. Instead you use a special variety of it called Rowref, defined in `type/RowType.h`. It holds a reference both to the Row (that keeps the data) and to the RowType (that knows how to work with the Row). The RowType must be correct for the Row. It's possible to combine the completely unrelated Row and RowType, and the result will be at least some garbage data, or at most a program crash. The Perl wrapper goes to great lengths to make sure that this doesn't happen. In the C++ API you're on your own. You gain the efficiency at the price of higher responsibility.

The general rule is that it's safe to combine a Row and RowType if this RowType matches the RowType used to create that row. The matching RowTypes may have different names of the fields but the same substance. The methods described here do no type checking, and it's your responsibility to make sure that the types are matching.

A Row is created similarly to a RowType: build a vector describing the values in the row, call the constructor, you get the row. The vector type is FdataVec, and its element type is Fdata. Both of them are top-level (i.e. Triceps::FdataVec and Triceps::Fdata), not inside some other class, and both are defined in `type/RowType.h`.

An Fdata describes the data for one field. It tells whether the field is not NULL, and if so, where to find the data to place into that field. It doesn't know anything about the field types or such. It deals with the raw bytes: the pointer to the first byte of the value, and the number of bytes. As a special case, if you want the field to be filled with zeroes, set the data pointer to NULL. It is possible to specify an incorrect number of bytes, such as to create an int64 field of 3 bytes. This data will be garbage, and if it happens to be at the end of the row, might cause a crash when read. It's your responsibility to store the correct data. The same goes for the string fields: it's your responsibility to make sure that the data is terminated with a "\0", and that "\0" is included into the length of the data. On the other hand, the **uint8[]** array fields don't need a "\0" at the end, all the bytes included into them are a part of the value.

The data vector gets constructed similarly to the field vector: either start with an empty vector and push pack the elements, or allocate one of the right size and set the elements. The relevant Fdata constructors and methods are:

```
Fdata();
```

```
Fdata(bool notNull, const void *data, intptr_t len);
void setPtr(bool notNull, const void *data, intptr_t len);
void setNull();
```

If the argument `notNull` is false, the values of `data` and `len` don't matter. The default constructor sets the `notNull` to false, and thus can be used to create the NULL fields. `setNull()` is also a shortcut of `setPtr()` that sets the `notNull` to false and ignores the other fields.

The construction itself is done with the factory method `RowType::makeRow()`, and typically the result it stored in a Rowref. Or there is a convenience constructor of Rowref directly from an FdataVec. It calls `makeRow()` internally so both forms provide the same result. For example:

```
uint8_t v_uint8[10] = "123456789"; // just a convenient representation
int32_t v_int32 = 1234;
int64_t v_int64 = 0xdeadbeefc00c;
double v_float64 = 9.99e99;
char v_string[] = "hello world";

FdataVec fd1;
fd1.push_back(Fdata(true, &v_uint8, sizeof(v_uint8)-1)); // exclude \0
fd1.push_back(Fdata(true, &v_int32, sizeof(v_int32)));
fd1.push_back(Fdata()); // a NULL field
fd1.push_back(Fdata(false, NULL, 0)); // another NULL field
fd1.push_back(Fdata(true, &v_float64, sizeof(v_float64)));
fd1.push_back(Fdata(true, &v_string, sizeof(v_string)));

Rowref r1(rt1,  rt1->makeRow(fd1));
Rowref r2(rt1,  fd1);

FdataVec fd2(3);
fd2[0].setPtr(true, &v_uint8, sizeof(v_uint8)-1); // exclude \0
fd2[1].setNull();
fd2[2].setFrom(r1.getType(), r1.get(), 2); // copy from r1 field 2

Rowref r3(rt1,  fd2);
```

The field fd2[2] shows yet another feature, copying the data from a field of another row. It sets the data pointer to the location inside the original row, and the data will be copied when the new row gets created. So make sure to not release the reference to the original row until the new row is created (Fdata has a helper field for that, it will be shown later). The prototype is:

```
void setFrom(const RowType *rtype, const Row *row, int nf);
```

In the example above the vector fd2 is smaller than the number of fields in the row. The rest of fields are then filled with NULLs. They actually are literally filled with NULLs in fd2: if the size of the argument vector for `makeRow()` is smaller than the number of fields in the row type, the vector gets extended with the NULL values before anything is done with it. It's no accident that the argument of the `RowType::makeRow()` is not const, and the same applies to the Rowref constructor and assignment from FdataVec:

```
class RowType {
  virtual Row *makeRow(FdataVec &data) const;
};

class Rowref {
  Rowref(const RowType *t, FdataVec &data);
  Rowref &operator=(FdataVec &data);
};
```

It's also possible to have more elements in the FdataVec than in the row type. In this case the extra arguments are considered the "overrides": the "main" elements set the size and initial contents of the fields while the overrides copy the data fragments over that. Remember, setting the `data` argument of Fdata methods to NULL makes the elements initially set to all zeroes.

There is a special form of constructor and a setting method to build the override fields:

```
Fdata(int nf, intptr_t off, const void *data, intptr_t len);
void setOverride(int nf, intptr_t off, const void *data, intptr_t len);
```

Here `nf` is the number (AKA index) of the field (starting from 0) whose contents is to be overridden, `off` is the byte offset in it where the override will start, and `data` and `len` point to the location to copy from as usual.

The overrides are a convenient way to assemble the array fields from the fragments. The following example shows the various permutations:

```
RowType::FieldVec fields4;
fields4.push_back(RowType::Field("a", Type::r_int64, RowType::Field::AR_VARIABLE));

Autoref<RowType> rt4 = checkOrThrow(new CompactRowType(fields4));

FdataVec fd4;
Fdata fdtmp;

// allocate space for 10 int64s, initially all 0s
fd4.push_back(Fdata(true, NULL, sizeof(v_float64)*10));

// override an element at index 2
fd4.push_back(Fdata(0, sizeof(v_int64)*2, &v_int64, sizeof(v_int64)));

// fill a temporary element with setOverride and apply it at index 4
fdtmp.setOverride(0, sizeof(v_int64)*4, &v_int64, sizeof(v_int64));
fd4.push_back(fdtmp);

// manually copy an element from r1 to the index 5
fdtmp.nf_ = 0;
fdtmp.off_ = sizeof(v_int64)*5;
r1.getType()->getField(r1.get(), 2, fdtmp.data_, fdtmp.len_);
fd4.push_back(fdtmp);

Rowref r4(rt4,  fd4);
```

This creates a row type from a single field "a" at index 0, an array of int64. The data vector fd4 has the 0th element define the space for 10 elements in the array, filled by default with zeroes. It doesn't have to zero them, it could have copied the data from some location in memory. I've just done the zeroing here to show how it can be done.

The rest of elements are the overrides constructed in different ways.

The first override sets the 2nd element (counting from 0) of the array with the value from v_int64.

The second override uses the method `setOverride()` for the same purpose on the 4th element. It sets a temporary Fdata which then gets appended (copied) to the vector.

The third override shows a more involved way. It copies the value from the row r1. Since there is no ready method for this purpose (perhaps there should be?), it goes about its way manually, setting the fields explicitly. `nf_` is the same as `nf` in the methods, the field number to override. `off_` is the offset. And the location and length get filled into `data_` and `len_` by `getField()`, which takes the data from the row r1, field 2.

But wait, the field 2 of r1 has been set to NULL! Should not the NULL indication be set in the copy as well? As it turns out, no. The NULL indication (the field `notNull_` being set to false) is ignored by `makeRow()` in the override elements. However `getField()` will set the length to 0, so nothing will get copied. The value at index 5 will be left as it was initially set, which happens to be 0.

So in the end the values in the field "a" array at indexes 2 and 4 will be set to the same as v_int64, and at the other indexes 0...9 to 0.

If multiple overrides specify the overlapping ranges, they will just sequentially overwrite each other, and the last one will win.

If an override attempts to specify writing past the end of the originally reserved area of the field, it will be quietly ignored. Just don't do this. If the field was originally set to NULL, its reserved area will be zero bytes, so any overrides attempting to write into it will be silently ignored.

The summary is: the overrides allow to build the array values efficiently from the disjointed areas of memory, but if they are used, they have to be used with care.

And for the reference the fields of Fdata that can be accessed directly:

```
Rowref row_; // in case if data comes from another row, can be used
  // to keep a hold on it, but doesn't have to
  // if the row won't be deleted anyway
const char *data_; // data to store, may be NULL to just zero-fill
intptr_t len_; // length of data to store
intptr_t off_; // for overrides only: offset into the field
int nf_; // for overrides only: index of field to fill
bool notNull_; // this field is not null (only for non-overrides)
```

The field `row_` has been mentioned before. It's a convenience placeholder for copying data from another row (and possibly replacing some of the fields). You don't have to use it if you know that a reference to the original row is kept somewhere until the construction of the new row is completed. But sometimes this is difficult to do while keeping the scopes straight. Then you can make `row_` refer to the original row, and it's guaranteed to stay valid. Of course, if you want to copy the fields from more than one original row, you really have to keep the references to them in some other way.

As has been said at the start of this section, all the operations on the rows are done through the RowType's methods. Let's take a closer look at them. And to reiterate once again, the row must be created with a matching type.

```
bool isFieldNull(const Row *row, int nf) const;
```

Check whether a field is NULL, `nf` is the number (AKA index) of the field starting from 0.

```
bool getField(const Row *row, int nf, const char *&ptr, intptr_t &len) const;
```

Get a field's data, `nf` is the index (AKA "number") of the field starting from 0. The returned value will be true if the field is not NULL. The values at arguments `ptr` and `len` will be populated with the pointer to the field's data in the row and its length information. If the field is NULL, they will be set to NULL and 0.

The returned data pointer type is constant, to remind that the rows are immutable and the data in them must not be changed. However for the most types you can't refer by this pointer and get the desired value directly, because the data might not be aligned right for that data type. Because of this the returned pointer is a `char*` and not `void*`. If you have an int64 field, you can't just do:

```
int64_t *data;
intptr_t len;
if (getField(myrow, myfield, data, len)) {
  int64_t val = *data; // WRONG!
}
```

Fortunately, the type checks will catch this usage attempt right at the call of `getField()`. But there also are the convenience functions that return the values of particular types. They are implemented over `getField()` and take care of the alignment issue.

```
uint8_t getUint8(const Row *row, int nf, int pos = 0) const;
int32_t getInt32(const Row *row, int nf, int pos = 0) const;
int64_t getInt64(const Row *row, int nf, int pos = 0) const;
double getFloat64(const Row *row, int nf, int pos = 0) const;
const char *getString(const Row *row, int nf) const;
```

The convenience wrappers for getting the values of specific types. The `getString()` returns only the pointer to the start of the string, not the length, expecting that you will treat it like a C string and rely on the "\0" at the end. Obviously, if a string contains "\0" in the middle, that would not work very well, and if you care about that, use `getField()` directly instead.

If the field is NULL, these methods return the value of 0 for the numeric types, and an empty string ("") for the string type, thus simulating the Perl semantics of undefined values. If you care about the field being NULL, check it separately:

```
if (!rt1->isFieldNull(r1, nf)) {
  int64_t val = rt1->getInt64(r1, nf);
  ...
}
```

The field `pos` is the position in an array field. It's an array index, not a byte offset. These methods don't care very much about the differentiation of scalars and arrays, from their standpoint a scalar is the same thing as an array of length 1. So the default value `pos` = 0 works well for the scalars. For the arrays, if pos is beyond the end of the array, the result is the same as for a NULL field. Even more so, a NULL field is considered the same thing as an array of length 0, and it's really just a specific case of the general rule: for a NULL field any `pos` value is considered to be beyond the end of the array.

For a side note, the arguments of these calls are Row*, not Rowref. It's cheaper, and OK for memory management because it's expected that the row would be held in a Rowref variable anyway while the data is extracted from it. Don't construct an anonymous rowref object and immediately try to extract a value from it!

```
int64_t val = rt1->getInt64(Rowref(rt1, datavec), nf); // WRONG!
```

However if you have an Fdata vector, there is no point in constructing a row to extract back the same data in the first place.

The Rowref has the same-named convenience methods to access the row stored in it, using the type stored in it. They are described in detail below.

```
void splitInto(const Row *row, FdataVec &data) const;
```

A convenience for copying the rows with modification of some fields. Does the first part of the copying: splits the original row into the field data references which can then be modified (the Fdata entries, not the data itself) to change some fields. The vector referred by the `data` argument will be resized to the number of fields in this row type and filled with the information. This method **does not** modify `data.row_` in any way, if you want to put a reference to the original row in there, you have to do it manually.

```
Row *makeRow(FdataVec &data) const;
```

Create a row from the data vector, as described above. If the size of `data` is less than the number of fields in this row type, the vector will be extended with the NULL fields before creating the row. If the vector size is larger, the rest of fields will be treated as overrides.

```
void destroyRow(Row *row) const;
```

The destructor for the rows. Normally is should never be called manually, instead use Rowref to maintain the proper reference count and do the destruction when the reference count goes to 0.

```
Row *copyRow(const RowType *rtype, const Row *row) const;
```

Copy a row, essentially a combination of `splitInto()` and `makeRow()`. Here the current row type and the `rtype` argument (which is the type of the argument `row`) don't have to be completely matching, the restriction is more loose: one of them has to be a matching prefix of the other. In other words, one of the types may have extra fields. This method automatically takes care of either cutting these extra fields if the result type is shorter or adding extra NULL fields if the result type is longer.

```
static void fillFdata(FdataVec &v, int nf);
```

Fill an FdataVec, extending it with NULL fields to the size `nf`. If the vector is of an already the same length or longer, it is left unchanged. Note that this method is static. There is practically no need to call it manually, `makeRow()` calls it for you automatically.

```
bool equalRows(const Row *row1, const Row *row2) const;
```

Check two rows for an absolute equality. Returns true if the rows are equal, otherwise false. This is the method used by the FIFO indexes to find the equal rows by value. Right now it's defined to work only on the rows of the same type, including the same representation (but since only one CompactRowType representation is available, this is not a problem). When more representations become available, it will likely be extended.

```
bool isRowEmpty(const Row *row) const;
```

Check the row for emptiness. Returns true if all the fields in the row are NULL.

```
void hexdumpRow(string &dest, const Row *row, const string &indent="") const;
```

Dump the bytes of the row in a hexadecimal format, appending to the destination string `dest`. This method is rarely needed in practice, only to investigate some strange data corruptions. Since the Row object is opaque, the length of its data is hidden. This method finds the length and then passes the data to `hexdump()`, described in Section 20.5: "String utilities" (p. 427) .

And finally, as promised at the start of the section, a reference of Rowref, defined in `type/RowType.h`. Overall, it's a variety of Autoref, so all the Autoref methods (except constructors) work unchanged, and a few more methods are added.

First, let's dig into some more details of why it is needed. The rows in Triceps aren't necessarily just blobs of memory. This feature isn't used anywhere yet, but the rows in Triceps are designed to be able to contain references to the other rows and in general to the other objects. So they can't be destroyed by just freeing the memory. The destructor must know, how to release the references to these nested objects first. And knowing where these references are depends on the type of the row. And rows may be of different types. This calls for a virtual destructor.

But having a virtual destructor requires that every object has a pointer to the table of virtual functions. That adds an overhead of 8 bytes per row, and the rows are likely to be kept by the million, and that overhead adds up. So I've made the decision to save these 8 bytes and split that knowledge. It might turn out to be a premature optimization, but since it's something that would be difficult to change later, I've got it in early.

The knowledge of how to destroy a row (and also how to copy the row and to access the elements in it) is kept in the row type object. So a reference to a row needs to know two things: the row and the row's type. It's still the same extra 8 bytes of a pointer, but there are only a few row references active at a time (the tables don't use the common row references to keep the rows, instead they are implemented as a special case and have one single row type for all the rows they store).

The constructor puts both these items into a Rowref:

```
Rowref();
Rowref(const RowType *t, Row *r = NULL);
Rowref(const Rowref &ar);
```

The RowType will be referred from the new Rowref through an Autoref. So after you've created the Rowref, feel free to discard any other references to the RowType, the Rowref will keep it alive.

The RowType may be NULL if the Row is NULL.

The default constructor sets them both to NULL. Which is dangerous: make sure to assign a proper RowType when you assign a Row, or your program will crash on a NULL pointer. In fact, initially this class had no default constructor because of this danger. But eventually the convenience of the default constructor had outweighed the danger. Just be careful.

```
Rowref(const RowType *t, FdataVec &data);
```

The convenience constructor that wraps the making of the Row from FdataVec.

```
Rowref &operator=(const Rowref &ar);
```

Assign from another Rowref, copying both the RowType and Row.

```
void assign(const RowType *t, Row *r);
```

Assign the explicit RowType and Row. The Row may be NULL. The RowType might also be NULL, but only if the Row is NULL.

```
Rowref &operator=(Row *r);
```

Assign just the Row, preserving the RowType. Make sure that the previously set type is not NULL and matches the row!

```
Rowref &operator=(FdataVec &data)
```

A convenience wrapper: make the Row from FdataVec and assign it. Rowref's current RowType is used to construct the Row, so it must be not NULL.

```
Rowref &copyRow(const RowType *rtype, const Row *row);
Rowref &copyRow(const Rowref &ar);
```

More convenience wrappers: make a copy of a Row and assign it. The usual caveats of copying and assignment apply. And the original row must not be NULL.

```
Row *get() const;
const RowType *getType() const;
```

Get the row an the type back.

The usual conversion to a Row pointer and the operator `->` on it also work automatically, and `isNull()` checks if the Row is NULL. The operators `==` and `!=` check for the equality of the Row pointer, ignoring the RowType pointer.

```
bool isRowEmpty() const
```

Another convenience wrapper for checking if all the fields in the Row are NULL. The Row pointer itself must not be NULL.

```
uint8_t getUint8(int nf, int pos = 0) const;
int32_t getInt32(int nf, int pos = 0) const;
int64_t getInt64(int nf, int pos = 0) const;
double getFloat64(int nf, int pos = 0) const;
const char *getString(int nf) const;
```

These convenience wrappers work exactly the same as the methods of the RowType. The Row must not be NULL.

20.14. TableType reference

The TableType describes the type of a table, defined in `type/TableType.h`. It inherits being an Mtarget from the Type, and can be shared between the threads after it has been initialized. However like RowType, it can benefit from creating a separate copy for each thread, to keep the reference count changes local to one CPU (see the method `deepCopy()` below).

In the C++ API it's built very similarly to the current Perl API, by constructing a bare object, then adding information to it, and finally initializing it. The Perl API will eventually change to something more Perl-like, the C++ API will stay this way.

The creation goes like this:

```
Autoref<TableType> tt = initializeOrThrow( (new TableType(rt1))
  ->addSubIndex("primary", it
  )->addSubIndex("secondary", itcopy
  )
);
```

In reality the index types would also be constructed as a part of this long statement but for clarity they were assumed to be pre-created as `it` and `itcopy`.

After a table type has been initialized, nothing can be added to it any more. Just as in the Perl API, the `addSubIndex()` adds not its argument index type object but its deep copy (see more about the copy varieties in Section 20.4: "The many ways to do a copy" (p. 424)). When the table type gets initialized, these index types get tied to it.

Note that the operator `new` has to be in parenthesis to get the priorities right. It's kind of annoying, so the better-looking equivalent way to do it is to use the static method `make()`:

```
Autoref<TableType> tt = initializeOrThrow(TableType::make(rt1)
  ->addSubIndex("primary", it
  )->addSubIndex("secondary", itcopy
  )
);
```

And here is a more complete example, with making of all the components in one chain:

```
Autoref<TableType> tt = initializeOrThrow(TableType::make(rt1)
  ->addSubIndex("primary", HashedIndexType::make(
    NameSet::make()->add("a")->add("e"))
    ->setAggregator(new MyAggregatorType("onPrimary", NULL))
    ->addSubIndex("level2", new FifoIndexType)
  )
);
```

The construction-related methods are:

```
TableType(Onceref<RowType> rt);
static TableType *make(Onceref<RowType> rt);
TableType *addSubIndex(const string &name, IndexType *index);
void initialize();
```

The method `initialize()` is called by the `initializeOrThrow()` template, which then checks for errors and throws an Exception if it finds any. But you can call it manually instead if you wish.

The working of the `addSubIndex()` is such that it doesn't put its `this` object into any kind of Autoref. Because of that it's able to pass that pointer right through to its result for chaining. It doesn't check anything and can't throw any exceptions. So the TableType object created by the constructor gets through the chain of `addSubIndex()` without having any counted references to it created, its reference count stays at 0. Only when the result of the chain is assigned to an Autoref, the first reference gets created. Obviously, this chain must not be interrupted by any exceptions or the memory will leak. Any detected errors must be collected in the embedded error objects, that will be read after initialization.

The result of initialization is void for a good reason: if you call `initialize()` manually, you can't include it into this chain, and the void result forces the non-chained call. Before the initialization is called, the TableType object must be properly held in a counted reference. For example:

```
Autoref<TableType> tt = TableType::make(rt1)
  ->addSubIndex("primary", it
  )->addSubIndex("secondary", itcopy
  );
tt->initialize();
if (tt->getErrors()->hasError())
  throw Exception(tt->getErrors());
```

But `initializeOrThrow()` is smart enough to do things correctly within the chained call format. It saves the Table-Type in an Autoref, then calls `initialize()`, and on success returns that Autoref, never corrupting the references. It involves a little more overhead but the TableType construction usually happens only once at the start-up time, and thus a minor difference in efficiency doesn't matter.

It's safe to call `initialize()` multiple times, the repeated calls will simply have no effect.

The sub-indexes may be added only while the table type is not initialized, afterwards it will throw an Exception.

The methods to examine the contents of the table type are:

```
bool isInitialized() const;
```

Check whether the table type has been initialized.

```
const RowType *rowType() const;
```

Returns the row type of the table type. Since the table type is not expected to be destroyed immediately, it's OK to return a plain pointer.

```
IndexType *findSubIndex(const string &name) const;
```

Find the index by name. Returns NULL if not found. This looks only for the top-level indexes, to find the nested indexes, the similar calls have to be continued on the further levels. At the moment there is no ready method to resolve a whole index path in C++, however the same method in IndexType has a special feature that allows to do it by simple chaining:

```
Autoref<IndexType> it = tt->findSubIndex("level1")->findSubIndex("level2");
if (it.isNull()) {
  // not found
}
```

That special feature is that `findSubIndex()` in IndexType works even on a NULL reference, simply returning NULL. So if `findSubIndex("level1")` returns NULL, `findSubIndex("level2")` will simply propagate that NULL. The downside compared with the path-resolving method in Perl is that this chain doesn't tell you, which of the indexes in the path were not found.

```
IndexType *findSubIndexById(IndexType::IndexId it) const;
```

Finds the first index type of a particular kind (or NULL if none found). The ids are like IndexType::IT_HASHED, IndexType::IT_FIFO, IndexType::IT_SORTED.

```
IndexType *getFirstLeaf() const;
```

Finds the first leaf index type. This call does search through the whole depth of the index tree for the first leaf index type.

```
const IndexTypeVec &getSubIndexes() const;
```

Returns the vector with all the top-level index type references. The vector is read-only, you must not change it.

```
TableType *copy() const;
```

Copy the table type, including all its contents of IndexTypes and AggregatorTypes, because they get tied to no more than one TableType. However the RowTypes stay shared. The copied table type is always uninitialized and thus can be further extended by defining more indexes and aggregators.

In case if the table type collected errors, the errors aren't copied, and in general you should not copy such a table type. The errors will be detected again when you try to initialize the copy.

```
TableType *deepCopy(HoldRowTypes *holder) const;
```

Copy the whole strucutre all the way down, up to and including the row types. This is used for exporting the TableTypes to the other threads through a Nexus. It keeps the reference counts localized to each thread, preventing the thrashing of them between CPUs as the references are created and destroyed.

The HoldRowTypes object is what takes care of sharing the underlying row types. To copy a bunch of objects with sharing, you create a HoldRowTypes, copy the bunch, destroy the HoldRowTypes, as described in more detail in Section 20.4: "The many ways to do a copy" (p. 424) .

The Perl API contains one more copy method, `copyFundamental()`. It's not directly available in the C++ API yet but the same task can be done manually.

Ultimately, the TableType is used to construct the tables. The factory method is:

```
Onceref<Table> makeTable(Unit *unit, const string &name) const;
```

This creates a table with the given name in a given unit. If the table type is not initialized or the initialization returned an error, will return NULL.

Finally, there is a call that you don't need to use:

```
RowHandleType *rhType() const;
```

Like everything else, the RowHandles have a type. But this type is very much internal, and it knows very little about the row handles. All it knows is how much memory to allocate when constructing a new RowHandle. The rest of the knowledge about the RowHandles is placed inside the Table. So, a Table (yes, a Table, **not** a TableType) acts among the other things as a type for its RowHandles.

20.15. NameSet reference

NameSet is a helper class used as an argument for construction of other classes, most notably HashedIndexType. It's an Mtarget.

NameSet is fundamentally a reference-counted vector of strings that allows to construct them with a chain of calls. It's used to construct such things as field list for the index key. Properly, the names in the set should be different but NameSet itself doesn't check for that. The order of values in it usually matters. So its class name is slightly misleading: it's not really a set, it's a vector, but the name has been applied historically. And in the future it might include the set functionality too, by adding a quick look up of index by name.

It's defined in `type/NameSet.h` as

```
class NameSet : public Mtarget, public vector<string> { ... }
```

The typical usage is like this:

```
HashedIndexType::make(NameSet::make()->add("a")->add("e"))
```

The factory method `make()` gives the more convenient operator priority than `new()`. The varieties of constructors and `make()` are:

```
NameSet();
NameSet(const vector<string> *other);
NameSet(const vector<string> &other);
static NameSet *make();
static NameSet *make(const vector<string> *other);
static NameSet *make(const vector<string> &other);
```

This approach to the copy constructors allows to construct from a plain vector, and since NameSet is its subclass, that works as a real copy constructor too. The constructor from a pointer makes the use of Autorefs more convenient, such as:

```
Autoref<NameSet> ns1 = NameSet::make()->add("a")->add("b");
Autoref<NameSet> ns2 = NameSet::make(ns1)->add("c");
```

All the vector methods are directly accessible, plus the ones added in NameSet.

```
NameSet *add(const string &s);
```

The method for the chained construction, adds a name to the vector and returns back the same NameSet object.

```
bool equals(const NameSet *other) const;
```

Comparison for equality. Returns true if both the name vectors are equal.

20.16. IndexType reference

Very much like the Perl API, the IndexType is an abstract class, in which you can't create the objects directly, you have to create the objects with its concrete sub-classes. It has the methods common for all the index types, it is defined in `type/IndexType.h` and like all types it's an Mtarget.

The index type id, defined as enum IndexType::IndexId, allows to find out the subclass of the actual object and cast it if desired. The supported index types are:

- IT_HASHED,

- IT_FIFO,

- IT_SORTED.

There is also the semi-hidden type IT_ROOT: you can't create it directly but every table type creates one implicitly, as the root of its index tree. And the value IT_LAST is defined past the last actual type, so if you ever need to iterate through the types, you can do it as

```
for (int i = 0; i < IndexType::IT_LAST; i++) { ... }
```

The conversion between the index type id and name can be done with the methods:

```
static const char *indexIdString(int enval, const char *def = "???");
static int stringIndexId(const char *str);
```

As usual with the contant-and-name conversions, if the numeric id `enval` is invalid, the string `def` is returned, by default "???". If the string name `str` is unknown, -1 is returned.

```
IndexId getIndexId() const;
```

Returns the type id of this index type. The id can't be changed, it gets hardcoded in the subclass constructor.

```
IndexType *addSubIndex(const string &name, Onceref<IndexType> index);
```

Add a sub-index, in exactly the same way as adding an index type to the TableType.

In works like TableType in all the other ways as well: it adds a copy of the argument, not the argument itself, and it designed for chaining. For example:

```
Autoref<TableType> tt = initializeOrThrow( TableType::make(rt1)
  ->addSubIndex("primary", HashedIndexType::make(
    NameSet::make()->add("b")->add("c"))
  )->addSubIndex("limit", FifoIndexType::make()
    ->setLimit(2) // will policy-delete 2 rows
  )
);
```

Not all index types allow sub-indexes but this method works silently for all of them. Any mistakes will be detected and reported at the initialization time. The sub-indexes may be added only until the index type is initialized. Any modification attempts after that will throw an Exception.

```
bool isLeaf() const;
```

Returns true if this index type has no sub-indexes. Of course, if this type is not initialized yet, more sub-types can be added to it to make it non-leaf later.

```
IndexType *findSubIndex(const string &name) const;
IndexType *findSubIndexById(IndexId it) const;
```

Find the sub-index by name or id, works in the same way as for TableType. A special feature is that it can be applied on a NULL object reference, like this:

```
Autoref<IndexType> it; // NULL by default
Autoref<IndexType> itsub = it->findSubIndex("xxx"); // doesn't crash, returns NULL
```

The idea here is to allow the safe chaining of findSubIndex() for the look-ups of the nested types:

```
Autoref<IndexType> it = tt->findSubIndex("level1")->findSubIndex("level2");
if (it.isNull()) {
  // not found
}
```

If any of the elements in the path are missing, the end result will be NULL, conveniently. But it won't tell you, which one was missing, inconveniently.

```
const IndexTypeVec &getSubIndexes() const;
```

Returns back the whole set of sub-indexes.

```
IndexType *getFirstLeaf() const;
```

Returns the first leaf index type (if a leaf itself, will return itself).

```
const NameSet *getKey() const;
```

Get the key information from the index. It will work with any kind of index, but will return a NULL if the index doesn't support the key. There is no matching method to set it, setting the key is up to the subclasses.

```
IndexType *setAggregator(Onceref<AggregatorType> agg);
const AggregatorType *getAggregator() const;
```

Set or get an aggregator type for this index type. As usual, any setting can be done only until the index type is initialized. Any modification attempts after that will throw an Exception.

```
bool isInitialized() const;
```

Check whether the index type has been initialized. The index types are initialized as a part of the table type initialization, there is no method to initialize an index type directly.

```
TableType *getTabtype() const;
```

Returns the table type, to which this index type is tied. The tying-together happens at the initialization time, so for an initialized index type this method will return NULL.

```
IndexType *copy(bool flat = false) const;
```

Create an un-initialized copy of this index. If flat is false then the copy is done with all the sub-index and aggregator types also copied, but the row types shared. This method is used by addSubIndex() to copy its argument before adding. By the way, the usual copy constructor could theoretically be used on the index types but usually doesn't make a whole lot of a sense because the sub-types and such will end up shared by reference.

If flat is true, the copy is flat, with the sub-index types and aggregator types simply not included. It's the C++ analog of the Perl method flatCopy(). This allows to re-create the partial index hierarchies, recursively flat-copying the parts that need to be included.

```
IndexType *deepCopy(HoldRowTypes *holder) const;
```

Create a deep copy, up to and including the row types. See the Section 20.14: "TableType reference" (p. 450) and Section 20.4: "The many ways to do a copy" (p. 424)for the detailed explanation. When you deep-copy a table type, the `holder` argument propagates recurively to all the index types in it.

There are great many more methods on the IndexType, that are used to maintain the index trees, but you don't need to look at them unless you are interested in the inner workings of the Triceps tables.

20.17. Index reference

The Index object can be accessed directly only in one place, in the aggregator handlers. In case if you're not sure about the difference between the Index and IndexType, it's explained in Section 9.10: "The index tree" (p. 101) . The examples of the usage are shown in Section 20.24: "Aggregator classes reference" (p. 473) , while this section lists the available methods.

The base Index class is defined in `table/Index.h`, and its subclasses in `table/RootIndex.h`, `table/FifoIndex.h`, `table/TreeIndex.h`, `table/TreeNestedIndex.h`.

```
const IndexType *getType() const;
```

Get the type of this index.

```
RowHandle *begin() const;
```

Get the handle of the first row of the group, in the default order according to its first leaf index type. Note that here it's not the whole table's first leaf index type but the first leaf in the index type subtree under this index's type. All the iteration methods return NULL if there are no more rows.

```
RowHandle *next(const RowHandle *cur) const;
```

Get the handle of the next row (or NULL if that was the last one) in the default order. The NULL argument makes the NULL result.

```
RowHandle *last() const;
```

Get the handle of the last row in the group in the default order.

The rest of the methods of Index aren't really to be used directly.

20.18. FifoIndexType reference

The FifoIndexType is defined in `type/FifoIndexType.h`. It works the same way as in Perl, except that it provides two ways to set the configuration values: either as the constructor/factory arguments or as chainable methods:

```
FifoIndexType(size_t limit = 0, bool jumping = false, bool reverse = false);
static FifoIndexType *make(size_t limit = 0, bool jumping = false, bool reverse = false);

FifoIndexType *setLimit(size_t limit);
FifoIndexType *setJumping(bool jumping);
FifoIndexType *setReverse(bool reverse);
```

So the following are equivalent:

```
Autoref<IndexType> it1 = new FifoIndexType(100, true, true);
Autoref<IndexType> it2 = FifoIndexType::make()
  ->setLimt(100)
  ->setJumping(true)
  ->setReverse(true);
```

As usual, the settings can be changed only until the initialization. The settings can be read back at any time with:

```
size_t getLimit() const;
```

```
bool isJumping() const;
bool isReverse() const;
```

Note that the limit is unsigned, and setting it to negative values results in it being set to very large positive values. The limit of 0 means "unlimited".

All the common methods inherited from IndexType and Type work as usual.

The `equals()` and `match()` are equivalent for the FifoIndexType, and are true when all the parameters are set to the same values.

20.19. HashedIndexType reference

The HashedIndexType is defined in `type/HashedIndexType.h`. It also allows to specify its only argument, the selection of the key fields, in the constructor, or set it later with a chainable call:

```
HashedIndexType(NameSet *key = NULL);
static HashedIndexType *make(NameSet *key = NULL);
HashedIndexType *setKey(NameSet *key);
```

One way or the other, the key has to be set, or a missing key will be detected as an error at the initialization time. Obviously, all the conditions described in the Perl API apply: the key fields must be present in the table's row type, and so on.

The key can be read back using the parent class method `IndexType::getKey()`. The value returned there is a `const NameSet*`, telling you that the key NameSet must not be changed afterward.

The check for `equals()` requires that the key fields are the same (not the exact same NameSet object, but its contents being the same). The check for `match()` is more tricky and depends on whether the index type has been initialized. The match check of initialized types is correct, while for the uninitialized types it's simplified. This may lead to slightly surprising effects when the two indexes match inside the initialized table types but their uninitialized copies don't. However the comparison of the uninitialized index types is probably not that usable anyway.

The simplified `match()` check is the same as `equals()`: the list of key fields must be the same. However this has a tricky effect: if two table types have matching row types with different field names, and the same Hashed indexes differing only in the key field names (following the difference in the row types), these table types will be considered non-matching because their hashed indexes are non-matching.

After the indexes are initialized, they can do a better match check. They can find their table types, and from there the row types. And then compare if the keys refer to the matching fields or not, even if their names are different. The condition then becomes that the key fields are matching in the row type, even if their names are not.

20.20. SortedIndexType reference

The SortedIndexType is defined in `type/SortedIndexType.h`, and provides a way to define any custom sorting criteria. That is done by defining your condition class, derived from SortedIndexCondition, and passing it to the SortedIndexType. Because of this the index type itself is simple, all the complex things are in the condition:

```
SortedIndexType(Onceref<SortedIndexCondition> sc);
static SortedIndexType *make(Onceref<SortedIndexCondition> sc);
SortedIndexCondition *getCondition() const;
```

A few example of the sort conditions are discussed below, they can also be found in `type/test/t_xSortedIndex.cpp`. More examples are available in `type/test/t_TableType.cpp` and in `perl/Triceps/PerlSortCondition.*`. SortedIndexCondition provides a set of virtual methods that can be re-defined in the subclass to create a custom condition. Indeed, some of them must be re-defined, since they are pure virtual.

The following basic example defines only the absolute minimum of methods. It sorts by an int32 field, whose index (starting as usual from 0) is specified in the constructor:

```
class Int32SortCondition : public SortedIndexCondition
{
public:
  // @param idx - index of field to use for comparison (starting from 0)
  Int32SortCondition(int idx) :
    idx_(idx)
  { }

  Int32SortCondition(const Int32SortCondition *other, Table *t) :
    SortedIndexCondition(other, t),
    idx_(other->idx_)
  { }

  virtual SortedIndexCondition *copy() const
  {
    return new Int32SortCondition(*this);
  }

  virtual TreeIndexType::Less *tableCopy(Table *t) const
  {
    return new Int32SortCondition(this, t);
  }

  virtual void initialize(Erref &errors, TableType *tabtype, SortedIndexType *indtype)
  {
    SortedIndexCondition::initialize(errors, tabtype, indtype);
    if (idx_ < 0)
      errors.f("The index must not be negative.");
    if (rt_->fieldCount() <= idx_)
      errors.f("The row type must contain at least %d fields.", idx_+1);

    if (!errors->hasError()) { // can be checked only if index is within range
      const RowType::Field &fld = rt_->fields()[idx_];
      if (fld.type_->getTypeId() != Type::TT_INT32)
        errors.f("The field at index %d must be an int32.", idx_);
      if (fld.arsz_ != RowType::Field::AR_SCALAR)
        errors.f("The field at index %d must not be an array.", idx_);
    }
  }

  virtual bool equals(const SortedIndexCondition *sc) const
  {
    // the cast is safe to do because the caller has checked the typeid
    Int32SortCondition *other = (Int32SortCondition *)sc;
    return idx_ == other->idx_;
  }

  virtual bool match(const SortedIndexCondition *sc) const
  {
    return equals(sc);
  }

  virtual void printTo(string &res, const string &indent = "", const string &subindent = "
  ") const
  {
    res.append(strprintf("Int32Sort(%d)", idx_));
  }

  virtual bool operator() (const RowHandle *rh1, const RowHandle *rh2) const
  {
    const Row *row1 = rh1->getRow();
```

```
    const Row *row2 = rh2->getRow();
    {
      bool v1 = rt_->isFieldNull(row1, idx_);
      bool v2 = rt_->isFieldNull(row2, idx_);
      if (v1 > v2) // isNull at true goes first, so the direction is opposite
        return true;
      if (v1 < v2)
        return false;
    }
    {
      int32_t v1 = rt_->getInt32(row1, idx_);
      int32_t v2 = rt_->getInt32(row2, idx_);
      return (v1 < v2);
    }
  }

  int idx_;
};

...

Autoref<IndexType> it = new SortedIndexType(new Int32SortCondition(1));
```

There are four kinds of constructors for a sort condition:

- One used to create the SortedIndexType in the first place, specifying the condition argument, such as Int32SortCondition(int idx) here.

- The copy constructor, used to copy the sort condition when its index gets copied. It should copy all the elements that could be modified by someone else but can share the references for the reference-counted elements that are unchangeable. Feel free to just use the default compiler-created copy constructor, as Int32SortCondition did, unless you really need it to do something more smart.

- The deep-copy constructor, used when SortedIndexType gets deep-copied, including the copying of the RowTypes and other referenced objects that are more efficient to not share between different threads. See the discussion of the deep-copying in Section 20.14: "TableType reference" (p. 450). Most of the sort conditions would not have any row types in them, nor anything deeper to copy than in the normal copy constructor, so they don't need to bother. The base class provides the default `deepCopy()` method that simply reverts to calling `copy()`, so no separate constructor is used. If you need to define it, the typical prototype for the deep-copy constructor will be:

  ```
  MySortCondition(const MySortCondition &other, HoldRowTypes *holder);
  ```

- The table-copy constructor. The need for this one is driven by the error reporting from the sort condition. For most of the sort condition classes defined directly in C++ it doesn't matter, they just can't experience any errors. But when the interpreted code is involved, the possibility of errors arises and something has to be done about them, and thus the infrastructure has to accomodate.

  When the comparison operator runs, throwing an Exception from the middle of the STL map logic is not a good idea. I've actually looked into the GCC implementation, and it would leak memory and be unhappy in this situation. This means that the sort condition has to report the error back to the Table in some other way. Which means that it has to have a link back to the table, and that link gets provided in the field `table_`. But that means that the sort condition instance gets tied to a table. And normally a sort condition belongs to an index type which belongs to a table type which can be shared by multiple tables. This dilemma is solved by making a copy of the sort condition for each created table.

  This copying and setting of the Table pointer gets handled by the sort condition method tableCopy() which in turn calls the table-copy constructor. Most of the work is done by the base class, you just need to pass through the arguments and copy your extra fields, as shown in the example.

There are three kinds of copy methods for a sort condition:

- `copy()` copies the sort condition by calling the copy constructor, as shown above. In particular, the SortedIndexType constructor makes a private copy of the condition, and remembers that copy, not the original.

- `deepCopy()` copies the sort condition all the way deep, as described in Section 20.14: "TableType reference" (p. 450) , by calling the deep-copy constructor. Most of the sort conditions don't need to bother since they don't have any RowTypes in them, the base class implements this method by calling `copy()`. If you need one, the typical implementation looks like:

```
IndexType *MySortCondition::deepCopy(HoldRowTypes *holder) const
{
  return new MySortCondition(*this, holder);
}
```

- `tableCopy()` creates a per-table instance of the copy condition, by calling the table-copy constructor.

The method `initialize()` is called at the table type initialization time. The argument `errors` is an already allocated Errors object to return the error messages in, `tabtype` is the table type where the initialization is happening, and `indtype` is the index type that owns this condition. Also the field `rt_` gets magically initialized to the table's row type reference before the sort condition initialization is called. This method is expected to do all the initialization of the internal state, check for all the errors, and return these errors if found.

To be more systematic, the following useful fields are defined in the sort condition base class:

```
Autoref<const RowType> rt_;
Table *table_;
intptr_t rhOffset_;
```

`rt_` is set to NULL on construction and the gets properly set when the SortedIndexType is initialized, before it calls the initialization of the sort condition. `table_` is also NULL to start with and gets set in `tableCopy()`, as described above. `table_` can be used by the comparison operator to report the fatal errors back to the table, as described in more detail below. `rhOffset_` is used by the more advanced sort conditions that want to cache some per-row information in the RowHandle, I'll show another example of that. It's the offset of that cached data in the RowHandle.

`equals()` and `match()` compare two conditions for equality and match. Before they get called, the caller checks that both conditions are of the same type (i.e. have the same C++ typeid), so it's safe to cast the second condition's pointer to our type. The easiest way to define `match()` is to make it the same as `equals()`. These methods may be called on both uninitialized and initialized conditions; if not initialized then the field `rt_` will be NULL.

`printTo()` appends the printout of this index's description to a string. For the simple single-line printouts it just appends to the result string. The multi-line prints have to handle the indenting correctly, as is described in Section 20.10: "Types reference" (p. 438) .

Finally, the `operator()` implements the comparison for "Less": it gets two row handles and returns true if the first one contains a row that is "less" (i.e. goes before in the sorting order) than the second one. The reason for why it's done like this is that the SortedIndexCondition is really a Less comparator class for the STL tree that had grown a few extra methods.

This example shows how to compare a value consisting of multiple elements. Even though this sort condition sorts by only one field, it first compares separately for NULL in that field, and then for the actual value. For each element you must:

- find the values of the element in both rows;

- compare for "<", and if so, return true;

- compare for ">", and if so, return false;

- otherwise (if they are equal), fall through to the next element.

The last element doesn't have to go through the whole procedure, it can just return the result of "<". And in this case the comparison for NULL wants the NULL value go before all the non-NULL values, so the result of true must go before false, and the comparison signs are reversed. It's real important that the second comparison, normally for ">", can not be

skipped (except for the last element). If you skip it, you will make a mess of the data and will spend a lot of time trying to figure out, what is going on.

That's it for the basics, the minimal subset of the methods that has to be defined. Now let's look at some more advances subjects.

What if `operator()` needs to to report an error? It has to set the sticky error in the table and return false:

```
Erref err;
err.f("error message");
table_->setStickyError(err);
return false;
```

One more method that can be supported in the sort condition is `getKey()`:

```
virtual const NameSet *getKey() const;
```

The default implementation just returns NULL, thus saying that the key is not available. The SortedIndexType's `getKey()` is really just a wrapper for the SortedIndexCondition's `getKey()`.

To show `getKey()` and a bit more techniques, so I've done a fairly big extension of the previous example. Now it can compare multiple fields, and the fields are specified by names. The only part missing from it being a general OrderedIndexType is the support of all the field types, not just int32.

```
class MultiInt32SortCondition : public SortedIndexCondition
{
public:
  // @param key - the key fields specification
  MultiInt32SortCondition(NameSet *key):
    key_(key)
  { }

  MultiInt32SortCondition(const MultiInt32SortCondition *other, Table *t) :
    SortedIndexCondition(other, t),
    idxs_(other->idxs_),
    key_(other->key_)
  { }

  virtual SortedIndexCondition *copy() const
  {
    return new MultiInt32SortCondition(*this);
  }

  virtual TreeIndexType::Less *tableCopy(Table *t) const
  {
    return new MultiInt32SortCondition(this, t);
  }

  virtual void initialize(Erref &errors, TableType *tabtype, SortedIndexType *indtype)
  {
    SortedIndexCondition::initialize(errors, tabtype, indtype);
    idxs_.clear();

    for (int i = 0; i < key_->size(); i++) {
      const string &s = (*key_)[i];
      int n = rt_->findIdx(s);
      if (n < 0) {
        errors.f("No such field '%s'.", s.c_str());
        continue;
      }
      const RowType::Field &fld = rt_->fields()[n];
      if (fld.type_->getTypeId() != Type::TT_INT32) {
```

```
        errors.f("The field '%s' must be an int32.", s.c_str());
        continue;
      }
      if (fld.arsz_ != RowType::Field::AR_SCALAR) {
        errors.f("The field '%s' must not be an array.", s.c_str());
        continue;
      }
      idxs_.push_back(n);
    }
  }

  virtual bool equals(const SortedIndexCondition *sc) const
  {
    // the cast is safe to do because the caller has checked the typeid
    MultiInt32SortCondition *other = (MultiInt32SortCondition *)sc;

    // names must be the same
    if (!key_->equals(other->key_))
      return false;

    // and if initialized, the indexs must be the same too
    if (!rt_.isNull()) {
      if (idxs_.size() != other->idxs_.size())
        return false;

      for (int i = 0; i < idxs_.size(); i++) {
        if (idxs_[i] != other->idxs_[i])
          return false;
      }
    }

    return true;
  }

  virtual bool match(const SortedIndexCondition *sc) const
  {
    MultiInt32SortCondition *other = (MultiInt32SortCondition *)sc;
    if (rt_.isNull()) {
      // not initialized, check by names
      return key_->equals(other->key_);
    } else {
      // initialized, check by indexes
      if (idxs_.size() != other->idxs_.size())
        return false;

      for (int i = 0; i < idxs_.size(); i++) {
        if (idxs_[i] != other->idxs_[i])
          return false;
      }
      return true;
    }
  }

  virtual void printTo(string &res, const string &indent = "", const string &subindent = "
") const
  {
    res.append("MultiInt32Sort(");
    for (NameSet::iterator i = key_->begin(); i != key_->end(); ++i) {
      res.append(*i);
      res.append(", "); // extra comma after last field doesn't hurt
    }
```

```
      res.append(")");
  }

  virtual const NameSet *getKey() const
  {
    return key_;
  }

  virtual bool operator() (const RowHandle *rh1, const RowHandle *rh2) const
  {
    const Row *row1 = rh1->getRow();
    const Row *row2 = rh2->getRow();

    int sz = idxs_.size();
    for (int i = 0; i < sz; i++) {
      int idx = idxs_[i];
      {
        bool v1 = rt_->isFieldNull(row1, idx);
        bool v2 = rt_->isFieldNull(row2, idx);
        if (v1 > v2) // isNull at true goes first, so the direction is opposite
          return true;
        if (v1 < v2)
          return false;
      }
      {
        int32_t v1 = rt_->getInt32(row1, idx);
        int32_t v2 = rt_->getInt32(row2, idx);
        if (v1 < v2)
          return true;
        if (v1 > v2)
          return false;
      }
    }
    return false; // falls through on equality, which is not less
  }

  vector<int> idxs_;
  Autoref<NameSet> key_;
};
```

The key is specified as a NameSet in the constructor. Unlike the HashedIndexType, there is no changing the key later, it must be specified in the constructor, and must not be NULL. The same as with HashedIndexType, the original name set becomes referenced by this sort condition and all its copies. So don't change and don't even use the original condition any more after you've passed it to the sort condition. This sort condition doesn't care to copy the NameSet in the deep copy, just leaves the deep copy the same as the default.

The initialization translates the field names to indexes (um, that's a confusing double usage of the word "index", here it's like "array indexes") in the row type, and checks that the fields are as expected.

The equality and match checks follow the fashion of HashedIndexType: if not initialized, they rely on field names, if initialized, they take the field indexes into the consideration (for equality both the names and indexes must be equal, for matching, only the indexes need to be equal).

The printing and copying is nothing particularly new and fancy. `getKey()` simply returns back the key. This feels a bit like an anti-climax, a whole big example for this little one-liner, but again, that's not the only thing that this example shows.

The "less" comparison function now loops through all the fields in the key. It can't do the shortcuts in the int32 comparison part any more, so that has been expanded to the full condition. If the whole loop falls through without returning, it means that the key fields in both rows are equal, so it returns false.

And an example, of its use, the key argument is created similarly to the hashed index:

```
Autoref<IndexType> it = new SortedIndexType(new MultiInt32SortCondition(
  NameSet::make()->add("b")->add("c")
));
```

Now we get to another advanced feature that has been mentioned before in the description of the row handles but is not accessible from Perl. A row handle contains a chunk of memory for every index type in the table. It is called a "row handle section". At the very least this chunk of memory contains the iterator in an index of that type, which allows to navigate through the table and to delete the row handles from the table efficiently.

But an index type may request more memory (the same fixed amount for each row handle) to store some row-specific information. For example, the hashed index type stores the value of the hash in its section, and uses this value for the efficient comparisons.

A sort condition may request and use memory in this section of a SortedIndexType. It is done by defining a few more virtual methods that handle the row section.

I could have made an example of the Hashed index re-implementation through the Sorted interface, but it's kind of boring, since you could as well look directly at the source code of the HashedIndexType. Instead I want to show a different kind of index that doesn't use the data in the rows for comparison at all but keeps the rows in the order they were inserted. Like a more expensive variety of FIFO index type. It's also a bit of a preview of a future feature. It assigns a new auto-generated sequence number to each row handle, and uses that sequence number for ordering. Later you can find the row handle quickly if you know its sequence number. If a table contains multiple copies of a row, the sequence numbers allow you to tell, which copy you are dealing with. It comes handy for such things as results of joins without a natural primary key. Of course, the usefulness of this preview is limited by the fact that there is no place for the sequence numbers in the rowops, and thus there is no way to propagate the sequence numbers in the model. That would have to be addressed before it becomes a full feature.

Now, you might ask, why not just add an extra field and put the sequence number in there? Sure, that would work too, and also solve the issue with the propagation in the rowops. However this means that as a row goes through a table, it gets copied to set the sequence number in it, which is less efficient. So ultimately keeping the sequence numbers "on the side" is more beneficial.

The implementation is:

```
class SeqSortCondition : public SortedIndexCondition
{
protected:
  class SeqRhSection : public TreeIndexType::BasicRhSection
  {
  public:
    SeqRhSection(int64_t val) :
      seq_(val)
    { }

    int64_t seq_; // the sequence number of this row handle
  };

public:
  SeqSortCondition() :
    seq_(0)
  { }

  SeqSortCondition(const SeqSortCondition *other, Table *t) :
    SortedIndexCondition(other, t),
    seq_(other->seq_)
  { }

  virtual SortedIndexCondition *copy() const
```

```
{
  return new SeqSortCondition(*this);
}

virtual TreeIndexType::Less *tableCopy(Table *t) const
{
  return new SeqSortCondition(this, t);
}

virtual void initialize(Erref &errors, TableType *tabtype, SortedIndexType *indtype)
{
  SortedIndexCondition::initialize(errors, tabtype, indtype);
  seq_ = 0;
}

virtual bool equals(const SortedIndexCondition *sc) const
{
  return true;
}

virtual bool match(const SortedIndexCondition *sc) const
{
  return true;
}

virtual void printTo(string &res, const string &indent = "", const string &subindent = "
") const
{
  res.append("Sequenced");
}

virtual size_t sizeOfRhSection() const
{
  return sizeof(SeqRhSection);
}

virtual void initRowHandleSection(RowHandle *rh) const
{
  // initialize the Seq part, the general Sorted index
  // will initialize the iterator
    SeqRhSection *rs = rh->get<SeqRhSection>(rhOffset_);
    new(rs) SeqRhSection(seq_++);
}

virtual void clearRowHandleSection(RowHandle *rh) const
{
    // clear the iterator by calling its destructor
    SeqRhSection *rs = rh->get<SeqRhSection>(rhOffset_);
    rs->~SeqRhSection();
}

virtual void copyRowHandleSection(RowHandle *rh, const RowHandle *fromrh) const
{
    SeqRhSection *rs = rh->get<SeqRhSection>(rhOffset_);
    SeqRhSection *fromrs = fromrh->get<SeqRhSection>(rhOffset_);

    // initialize the iterator by calling its copy constructor inside the placement,
  // the sequence number gets copied too
    new(rs) SeqRhSection(*fromrs);
}
```

```
  // Helper method to read the sequence from the row handle,
  // can also be used by the end-user. The row handle must as usual
  // belong to a table of this type.
  int64_t getSeq(const RowHandle *rh) const
  {
      return rh->get<SeqRhSection>(rhOffset_)->seq_;
  }

  // Helper method to set the sequence in the row handle.
  // May be used only on the rows that are not in a table.
  void setSeq(const RowHandle *rh, int64_t val) const
  {
    if (rh->isInTable()) {
      throw Exception("Attempted to change the sequence on a row in table.", true);
    }
      rh->get<SeqRhSection>(rhOffset_)->seq_ = val;
  }

  virtual bool operator() (const RowHandle *rh1, const RowHandle *rh2) const
  {
    return getSeq(rh1) < getSeq(rh2);
  }

  mutable int64_t seq_; // the next sequence number to assign
};


...
Autoref<IndexType> it = new SortedIndexType(new SeqSortCondition());
...
```

The nested class SeqRhSection defines the structure of the section that a SortedIndexType with a SeqSortCondition allocates in the RowHandle. The section class for a sort condition must always inherit from TreeIndexType::BasicRhSection, to get the general SortedIndexType parts from it, such as the iterator. Any extra fields are owned by the sort condition.

The SeqSortCondition contains the sequence number generator seq_ (not to be confused with the same-named field seq_ in SeqRhSection), that gets initialized to 0, and will be incremented from there. Since each table will get its own copy of the condition, each of them will be counting independently.

The equals() and match() always return true because there is nothing configurable in this sort condition.

The new features start at sizeOfRhSection(). The size of each row handle in a table type is the same, and is computed by asking every index type in it at initialization time and adding up the totals (plus alignment and some fixed amount of basic data). sizeOfRhSection() does its part by telling the caller the size of SeqRhSection.

Then each row handle section must provide the ways to construct and destruct it. Naturally, to save space, a section must have no virtual table, so like for the rows, a separate method in the index type acts as its virtual destructor. And there is no such thing as a virtual constructor in C++, which gets simulated through more methods in the index type. The SortedIndexType delegates most of this work to the sort condition in it. The basic constructor is initRowHandleSection(), the copy constructor is copyRowHandleSection(), and the destructor is clearRowHandleSection().

Each of them gets the location of this index type's section in the row handle using:

```
SeqRhSection *rs = rh->get<SeqRhSection>(rhOffset_);
```

The field rhOffset_ gets initialized by the SortedIndexType machinery before either of these methods gets ever called. Here rs points to the raw bytes, on which the placement constructors and the explicit destructor are called.

The methods getSeq() and setSeq() are not virtual, they are unique to this SeqSortCondition. They allow to read the sequence from a row handle or set the sequence in it. Naturally, the sequence number may be changed only when the row handle is not in the table yet, or it would mess up the indexing horribly. It's OK to throw the exceptions from setSeq() and getSeq() since they are called directly from the user code and won't confuse any Triceps code along the way.

If you want to find a row handle in the table by its sequence number, you start with creating a new row handle (which can even use an empty row). That new row handle will have a new sequence number assigned to it, but it doesn't matter, because next you call `setSeq()` and overwrite it with your desired number. Then you use this row handle to call `find()` or `delete()` on the table as usual.

The creation goes like this:

```
Rhref rh1(table, r1);
sc->setSeq(rh1, desired_number);
```

The class Rhref, a row handle reference, is similar in spirit to Rowref, and is described in detail in Section 20.23: "RowHandle and Rhref reference" (p. 472) . Or to read the number, you do:

```
int64_t seq = sc->getSeq(rh);
```

Here `sc` is the exact initialized sort condition from the actual table type (not from the exact table, since you can't get the sort condition from a table, but the table type is good enough). If you use a wrong or uninitialized sort condition, the `rhOffset_` in it will likely be wrong, and will cause all kinds of memory corruption. You can get the sort condition from a table type like this:

```
Autoref<SortedIndexType> ixt = dynamic_cast<SortedIndexType *>(tt-
>findSubIndex("primary"));
Autoref<SeqSortCondition> sc = dynamic_cast<SeqSortCondition *>(ixt->getCondition());
```

You don't have to use the dynamic cast but it's safer, and since you'd normally do it once at the model setup time and then just keep using the value, there is no noticeable performance penalty for it.

20.21. Gadget reference

The Gadget is unique to the C++ API, it has no parallels in Perl. It had also become somewhat obsolete in the recent times but still clinging to life for now.

Gadget is a base class, a single-threaded Starget, defined in `sched/Gadget.h`, its object being a something with an output label. And the details of what this something is, are determined by the subclass. Presumably, it also has some kind of inputs but it's up to the subclass. The Gadget itself defines only the output label. To make a concrete example, a table is a gadget, and every aggregator in the table is also a gadget. However the "pre" and "dump" labels of the table are not gadgets, they are just extra labels strapped on the side.

The reasons for the Gadget creation are mostly historic by now. At some point it seemed important to have the ability to associate a particular enqueueing mode with each output label. Most tables might be using `EM_CALL` but some, ones in a loop, would use `EM_FORK`, and those that don't need to produce the streaming output would use `EM_IGNORE`. This approach didn't work out as well as it seemed at first, and now is outright deprecated: the tables have `EM_CALL` hardcoded everywhere, and there are the newer and better ways to handle the loops. The whole Gadget thing should be removed at some point but for now I'll just describe it as it is.

As the result of that history, the enqueueing mode constants are defined in the Gadget class, enum EnqMode: `EM_SCHEDULE`, `EM_FORK`, `EM_CALL`, `EM_IGNORE`.

```
static const char *emString(int enval, const char *def = "???");
static int stringEm(const char *str);
```

Convert from the enqueueing mode constant to string, and back.

```
Gadget(Unit *unit, EnqMode mode, const string &name, const_Onceref<RowType> rt = (const
 RowType*)NULL);
```

The Gadget constructor is protected, since Gadget is intended to be used only as a base class, and never instantiated directly. The row type can be left undefined if it isn't known yet and initialized later. The output label will be created as soon as

the row type is known. The enqueueing mode may also be changed later, so initially it can be set to anything. All this is intended only to split the initialization in a more convenient way, once the Gadget components are set, they must not be changed any more.

The output label of the Gadget is a DummyLabel, and it shares the name with the Gadget. So if you want to differentiate that label with a suffix in the name, you have to give the suffixed name to the whole Gadget. For example, the Table constructor does:

```
Gadget(unit, Gadget::EM_CALL, name + ".out", rowt),
```

A Gadget keeps a reference to both its output label and its unit. This means that the unit won't disappears from under a Gadget, but to avoid the circular references, the Unit must not have references to the Gadgets (having references to their output labels is fine).

```
void setEnqMode(EnqMode mode);
void setRowType(const_Onceref<RowType> rt);
```

The protected methods to finish the initialization. Once the values are set, they must not be changed any more. Calling `setRowType()` creates the output label.

```
EnqMode getEnqMode() const;
const string &getName() const;
Unit *getUnit() const;
Label *getLabel() const;
```

Get back the gadget's information. The label will be returned only after it's initialized (i.e. the row type is known), before then `getLabel()` would return NULL. And yes, it's `getLabel()`, **not** `getOutputLabel()`.

The rest of the methods are for convenience of sending the rows to the output label. They are protected, since they are intended for the Gadget subclasses (which in turn may decide to make them public).

```
void send(const Row *row, Rowop::Opcode opcode) const;
```

Construct a Rowop from the given row and opcode, and enqueue it to the output label according to the gadget's enqueueing method. This is the most typical use.

```
void sendDelayed(Tray *dest, const Row *row, Rowop::Opcode opcode) const;
```

Create a Rowop and put it into the `dest` tray. The rowop will have the enqueueing mode populated according to the Gadget's setting. This method is used when the whole set of the rowops needs to be generated before any of them can be enqueued, such as when a Table computes its aggregators. After the delayed tray is fully generated, it can be enqueued with `Unit::enqueueDelayedTray()`, which will consult each rowop's enqueueing method and process it accordingly. Again, this stuff exists for the historic reasons, and will likely be removed somewhere soon.

20.22. Table reference

The Table is defined in `table/Table.h`. It inherits from Gadget, with the table's output label being the gadget's output label. Naturally, it's an Starget and usable from one thread only.

Its constructor is not public, and it's created from the TableType with its method `makeTable()`:

```
Autoref<Table> t = tabType->makeTable(unit, "t");
```

The arguments are the unit where the table will belong and the name of the table.

Yeah, it's kind of weird that in Perl the method `makeTable()` is defined on Unit, and in C++ on TableType. But if I remember correctly, it has to do with avoiding the circular dependency in the C++ header files.

For the reference, that TableType method is:

```
Onceref<Table> makeTable(Unit *unit, const string &name) const;
```

The table has a large number of methods, grouped into multiple subsets.

```
const string &getName() const;
Unit *getUnit() const;
Label *getLabel() const;
```

These methods are inherited from the Gadget. The only special thing to remember is that `getLabel()` returns the table's output label. Technically, `getName()` has an overriding implementation in the Table, to return the table's proper name while its output label and the underlying gadget have the suffix ".out" appended to their name.

```
const TableType *getType() const;
const RowType *getRowType() const;
const RowHandleType *getRhType() const;
```

Get back the type of the table, of its rows, and of its row handles. There is not much direct use for the RowHandleType, the Table uses it internally to construct the RowHandles.

```
Label *getInputLabel() const;
Label *getPreLabel() const;
Label *getDumpLabel() const;
Label *getAggregatorLabel(const string &agname) const;
```

Get the assorted labels (remember, `getLabel()` inherited from Gadget is used for the output label). The aggregator label getter takes the name of the aggregator (as was defined in the TableType) as an argument.

```
FnReturn *fnReturn() const;
```

Get the FnReturn of this table's outputs. It gets created and remembered on the first call, and the subsequent calls return the same object. It has a few labels with the fixed names: "out", "pre" and "dump", and a label for each aggregator with the aggregator's name. It could throw an Exception if you name an aggregator to conflict with one of the fixed labels, which you should not. The FnReturn's name will be "*tableName*.fret".

Next go the operations on the table (and of course the table may also be modified by sending the rowops to its input label).

```
RowHandle *makeRowHandle(const Row *row) const;
```

Create a row handle for a row. Remember, the row handles are reference-counted, and also have the special kind of references with Rhref. So the returned pointer should be stored in an Rhref. The row handle created will not be inserted into the table yet.

```
bool insert(RowHandle *rh);
```

Insert a row handle into the table. This invokes all the row replacement policies along the way. If the handle is already in the table, does nothing and returns false. May also return false if a replacement policy refuses the row, but in practice there are no such refusing policies yet. Otherwise returns true.

It may throw an Exception. It may throw directly if the row handle doesn't belong to this table or propagate the exception up from the functions it calls: since the execution involves calling the output labels and such, an exception might be thrown from there.

```
bool insertRow(const Row *row);
```

The version that combines the row handle construction and insertion. Unlike Perl, in C++ this method is named differently instead of overloading. The comments about the replacement policies and return code, and about exceptions apply here too.

```
void remove(RowHandle *rh);
```

Remove a row handle from the table. If the handle is not in the table, silently does nothing. May throw an Exception.

```
bool deleteRow(const Row *row);
```

Find a matching row and delete it (internally this involves creating a new RowHandle for the argument row and finding it in the first leaf index by value, and disposing of that temporary row handle afterwards). Returns true if the row was found and removed, false if not found. May throw an Exception.

```
void clear(size_t limit = 0);
```

Clear the table by removing all the rows from it. The removed rows are sent as usual to the "pre" and "out" labels. If the limit is not 0, no more than that number of the rows will be removed. The rows are removed in the usual order of the first leaf index.

Next go the iteration methods. The rule of thumb is that for them a NULL row handle pointer means "end of iteration" or "not found" (or sometimes "bad arguments"). And they can handle silently the NULL row handles on the input, just returning NULL on the output.

```
RowHandle *begin() const;
```

Get the first row handle in the default order of the first leaf index. If the table is empty, returns NULL.

```
RowHandle *beginIdx(IndexType *ixt) const;
```

Get the first handle in the order of a particular index. The index type must belong to this table's type. For an incorrect index type it returns NULL (perhaps in the future this will be changed to an exception).

```
RowHandle *next(const RowHandle *cur) const;
RowHandle *nextIdx(IndexType *ixt, const RowHandle *cur) const;
```

Get the next row handle in the order of the default or specific index. Returns NULL after the last handle. It's safe to pass the current row handle as NULL, the result will be NULL, as well as on any other error.

```
RowHandle *firstOfGroupIdx(IndexType *ixt, const RowHandle *cur) const;
RowHandle *lastOfGroupIdx(IndexType *ixt, const RowHandle *cur) const;
```

Get the first or last row handle in the same group as the current row according to a non-leaf index. The NULL current handle will cause NULL returned. See the details in the description of the Perl API.

```
RowHandle *nextGroupIdx(IndexType *ixt, const RowHandle *cur) const;
```

Get the first row handle of the next group. The return will be NULL if the current group was the last one, or if the current handle is NULL.

The iteration loops may look as:

```
for (RowHandle *iter = t->begin(); iter != NULL; iter = t->next(iter)) {
  ...
}

for (Rhref iter(t, t->beginIdx(level3)); !iter->isNull();
    iter = t->nextIdx(level3, iter)) {
  ...
}
```

Note that if the table does not change during the iteration (or at the very least if the row handle at the iterator is not being affected by the changes), the iterator doesn't have to be kept in an Rhref. A pointer to RowHandle is perfectly adequate and more efficient since it doesn't modify the reference counts. However you must be perfectly sure that the table will not be changed by any of the code in any labels called from the iteration loop.

Unlike Perl, there are no methods on RowHandle that produce the next RowHandle in the table, all the `next*()` methods are defined on the Table. It's possible to define them on Rhref, and will probably be done in the future.

Next go the size operations:

```
size_t size() const;
```

Get the number of rows currently in the table.

```
size_t groupSizeIdx(IndexType *ixt, const RowHandle *what) const;
```

Get the size of the group where the handle belongs according to a non-leaf index. If any arguments are wrong, returns 0. The row handle doesn't have to be in the table. If it isn't in the table, the method will find the group where the row would belong if it were inserted and return its current size.

```
size_t groupSizeRowIdx(IndexType *ixt, const Row *what) const;
```

A convenience version that makes a row handle from a row, finds the group size and disposes of the handle.

Next go the finding methods:

```
RowHandle *find(const RowHandle *what) const;
RowHandle *findIdx(IndexType *ixt, const RowHandle *what) const;
```

Find the handle of a matching row according to the default (first leaf) or the specific index, or return NULL if not found.

```
RowHandle *findRow(const Row *what) const;
RowHandle *findRowIdx(IndexType *ixt, const Row *what) const;
```

The convenience versions that create a temporary row handle and then perform the search.

20.22.1. Data dump

The dump API of the table sends the whole contents of the table to the "dump" label, thus making any labels connected to it perform an implicit iteration over the table.

```
void clear(size_t limit = 0);
```

Deletes the rows from the table. If `limit` is 0, the whole table gets cleared. If it's greater than 0, no more than this number of rows will be deleted. The deletion happens in the usual order of the first leaf index, and the rowops are sent to the table's output label as usual. It's really the same thing as running a loop over all the row handles and removing them.

```
void dumpAll(Rowop::Opcode op = Rowop::OP_INSERT) const;
void dumpAllIdx(IndexType *ixt, Rowop::Opcode op = Rowop::OP_INSERT) const;
```

The dump can go in the order of default or specific index. The opcode argument is used for the rowops sent on the dump label. Using the argument index type of NULL makes `dumpAllIdx()` use the default index and work just like `DumpAll()`. In the future there probably will also be methods that dump only a group of records.

As usual, the general logic of the methods matches the Perl API unless said otherwise. Please refer to the Perl API description for the details and examples.

20.22.2. Sticky errors

The table internals contain a few places where the errors can't just throw an Exception because it will mess up the logic big time, most specifically the comparator functions for the indexes. The Triceps built-in indexes can't encounter any errors in the comparators but the user-defined ones, such as the Perl Sorted Index, can. Previously there was no way to report these errors other than print the error message and then either continue pretending that nothing happened or abort the program.

The sticky errors provide a way out of this sticky situation. When an index comparator encounters an error, it reports it as a sticky error in the table and then returns false. The table logic then unrolls like nothing happened for a while, but before returning from the user-initiated method it will find this sticky error and throw an Exception at a safe time. Obviously, the incorrect comparison means that the table enters some messed-up state, so all the further operations on the table will keep finding this sticky error and throw an Exception right away, before doing anything. The sticky error can't be unstuck. The only way out of it is to just discard the table and move on.

```
void setStickyError(Erref err);
```

Set the sticky error from a location where an exception can not be thrown, such as from the comparators in the indexes. Only the first error sticks, all the others are ignored since (a) the table will be dead and throwing this error in exceptions from this point on anyway and (b) the comparator is likely to report the same error repeatedly and there is no point in seeing multiple copies.

```
Errors *getStickyError() const;
```

Get the table's sticky error. Normally there is no point in doing this manually, but just in case.

```
void checkStickyError() const;
```

If the sticky error has been set, throws an Exception with it.

20.23. RowHandle and Rhref reference

RowHandle is a mostly opaque class. It's defined in `table/RowHandle.h` but you never need to include that file directly: when you include `table/Table.h`, that takes care of it. If you look at the class definition, there are two public sections. The first one is really public, the second one is really for the internal use, I just didn't want to mess too much with the lists of friend classes. The really-public part is quite small:

```
const Row *getRow() const;
bool isInTable() const;
```

The meaning of these methods is the same as in Perl, get the row contained in the handle, and check whether this handle has been inserted into its table.

The constructor of RowHandle is private. It's constructed with a factory method in the Table, but normally a better approach is to use the wrapper constructor of Rhref, as described below. For the record, that Table factory method is:

```
RowHandle *makeRowHandle(const Row *row) const;
```

There is also a side API that allows the indexes and aggregators to place their sections into the RowHandles. It's not a part of the RowHandle class as such, which is quite dumb and only knows how to carry the bytes of the data. The smart part of the RowHandle construction is contained in the API between the Table and its indexes and aggregators, and is outlined in Section 20.20: "SortedIndexType reference" (p. 457) .

The row handles have the requirements very similar to the rows. They get created by the million, so the efficiency is important. In addition, they may contain the user data that has to be properly destroyed, the index and aggregator state mentioned above. For example, when an additive Perl aggregator stores its last state, it's stored in a row handle.

So the row handle references are handled similarly to the rows. They don't have a virtual destructor but rely on the Table that owns them to destroy them right. The special reference type for them is Rhref, defined in `mem/Rhref.h`.

It follows in the exact same mold as Rowref, only uses the Table instead of a RowType:

```
Rhref(Table *t, RowHandle *r = NULL);
void assign(Table *t, RowHandle *r);
```

The rest of comparisons, assignments etc. work the same as with Rowref.

The RowHandle pointer may be NULL, creating an Rhref to a yet unknown handle in a known table. The Table pointer may be NULL too (of course, only if the RowHandle is NULL), creating a completely NULL reference. As usual in the C++ API, these rules are not checked in the functions, so if you assign the incorrect NULLs, the program will crash.

As with Rowrefs, if the Table is not NULL, a RowHandle can be assigned into the Rowref through the assignment operator, but if the Table was not set, or if assigning a row handle from another table, the `assign()` method must be used to set both.

An important point is that an Rhref contains an Autoref to the Table, safely holding the table in place while the Rhref is alive, it's the same as the relation between the Rowref and RowType.

To find out the table of a Rhref, use:

```
Table *t = rhr.getTable();
```

Why is the value returned a simple pointer to the table and not an Autoref or Onceref? Basically, because it's the cheapest way and because the row handle is not likely to go anywhere. Nobody is likely to construct a RowHandle only to get the table from it and have it immediately destroyed. And even if someone does something of the sort

```
Autoref<Table> t = Rhref(t_orig, rh).getTable();
```

then the table itself is likely to not go anywhere, there is still likely to be another reference to the table that will still hold it in place. If there isn't then of course all bets are off, and it will end up with a dead reference to corrupted memory. Just exercise a little care, and everything will be fine. The same reasoning was used for the argument of the RowHandle constructor being also a table pointer, not an Autoref or Onceref.

An Rhref constructor may also be conveniently used to construct a RowHandle for a Row or directly from the field data:

```
Rhref(Table *t, Row *row);
Rhref(Table *t, FdataVec &data);
```

In fact, this is the recommended way to construct a RowHandle, immediately holding it in a reference, instead of doing all the intermediate calls manually.

20.24. Aggregator classes reference

20.24.1. AggregatorType reference

The AggregatorType is a base class from which you derive the concrete aggregator types, similarly to how it's done for the index types. It's defined in `type/AggregatorType.h`. It has a chunk of functionality common for all the aggregator types and the virtual functions that create the aggregator objects.

```
AggregatorType(const string &name, const RowType *rt);
```

The constructor provides a name and the result row type. Remember, that AggregatorType is an abstract class, and will never be instantiated directly. Instead your subclass that performs a concrete aggregation will invoke this constructor as a part of its constructor.

As has been described in the Perl part of the manual, the aggregator type is unique in the fact that it has a name. And it's a bit weird name: each aggregator type is kind of by itself and can be reused in multiple table types, but all the aggregator types in a table type must have different names. This is the name that is used to generate the name of the aggregator's output label in a table: "*table_name.aggregator_type_name*". Fundamentally, the aggregator type itself should not have a name, it should be given a name when connected to an index in the table type. But at the time the current idea looked good enough, it's easy, convenient for error messages, and doesn't get much in the way.

The result row type might not be known at the time of the aggregator type creation. All the constructor does with it is place the value into a reference field, so if the right type is not known, just use NULL, then change later at the initialization time. If it's still NULL after initialization, this will be reported as an initialization error.

```
AggregatorType(const AggregatorType &agg);
virtual AggregatorType *copy() const;
AggregatorType(const AggregatorType &agg, HoldRowTypes *holder);
virtual AggregatorType *deepCopy(HoldRowTypes *holder) const;
```

An aggregator type must provide the copy constructors and the virtual methods that invoke them. Both kinds of copies are deep but deepCopy() is even deeper, copying even the row types. See Section 20.4: "The many ways to do a copy" (p. 424) for details.

The basic copy is the same as with the index types: when an agggregator type gets connected into a table type, it gets actually copied, and the copy must always be uninitialized.

The virtual methods are typically defined in the subclasses as follows:

```
AgregatorType *MyAggregatorType::copy() const
{
  return new MyAggregatorType(*this);
}

AgregatorType *MyAggregatorType::deepCopy(HoldRowTypes *holder) const
{
  return new MyAggregatorType(*this, holder);
}
```

Some of the fields in the AggregatorType are directly usable by the subclasses:

```
const_Autoref<RowType> rowType_; // row type of result
Erref errors_; // errors from initialization
string name_; // name inside the table's dotted namespace
int pos_; // a table has a flat vector of AggregatorGadgets in it, this is the index for
 this one (-1 if not set)
bool initialized_; // flag: already initialized, no future changes
```

rowType_
    is the row type of the result. The constructor puts the argument value there but it can be changed at any time (until the initialization is completed) later.

errors_
    is a place to put the errors during initialization. It comes set to NULL, so if you want to report any errors, you have to create an Errors object first, there are the helper functions for that.

name_
    is where the aggregator name is kept. Generally, don't change it, treat it as read-only.

pos_
    has to do with management of the aggregator types in a table type. Before initialization it's -1, after initialization each aggregator type (that becomes tied to its table type) will be assigned a sequential number. Again, treat it as read-only, and you probably would never need to even read it.

initialized_
    shows that the initialization has already happened. Your initialization should call the initialization of the base class, which would set this flag. No matter if the initialization succeesed or failed, this flag gets set. It never gets reset in the original AggregatorType object, it gets reset only in the copies.

```
const string &getName() const;
const RowType *getRowType() const;
bool isInitialized() const;
virtual Erref getErrors() const;
```

The convenience getter functions that return the data from the fields. You can override getErrors() but there probably is no point to it.

```
virtual bool equals(const Type *t) const;
virtual bool match(const Type *t) const;
```

The equality and match comparisons are as usual. The defaults provided in the base AggregatorType check that the result row type is equal or matching (or that both result row types are NULL), and that the C++ typeid of both are the same. So if your aggregator type has no parameters, this is good enough and you don't need to redefine these methods. If you do have parameters, you call the base class method first, if it returns false, you return false, otherwise you check the parameters. Like this:

```
bool MyAggregatorType::equals(const Type *t) const
{
     if (!AggregatorType::equals(t))
        return false;

    // the typeid matched, so safe to cast
    const MyAggregatorType *at = static_cast<const MyAggregatorType *>(t);
    // ... check the type-specific parameters ...
}
```

The other method that you can re-define or leave alone is `printTo()`:

```
virtual void printTo(string &res, const string &indent = "", const string &subindent = "
 ") const;
```

The default one prints "aggregator (*result row type*) *name*". If you want to print more information, such as the name of the aggregator class and its arguments, you can define your own.

```
virtual void initialize(TableType *tabtype, IndexType *intype);
```

This method is called at the TableType initialization time, as it goes through all the components. This is the place where the aggregator type parses its arguments, matches them up with the information about the table type and produces any parsed internal representations. It may also compute the aggregation result type if that was not done earlier. The `initialize()` method in the subclass must always call the method in the parent class, to let it do its part. If any errors are found, they must be reported by setting an Errors object in the field `errors_`. If the aggregator type has no parameters, and its result type was set in the constructor, it doesn't have to define the `initialize()` method.

Finally, there are methods that will produce objects that do the actual work:

```
virtual AggregatorGadget *makeGadget(Table *table, IndexType *intype) const;
virtual Aggregator *makeAggregator(Table *table, AggregatorGadget *gadget);
```

This exposes quite a bit of the inherent complexity of the aggregators. For the simpler cases you can use the subclass BasicAggregatorType that handles most of this complexity for you and just skip these "make" methods. By the way, the IndexType has a "make" method of this kind too but it was not discussed because unless you define a completely new IndexType, you don't need to worry about it: it just happens under the hood. The SortedIndexType just asks you to define a condition and takes care of the rest, like the BasicAggregatorType for aggregators.

20.24.2. AggregatorGadget reference

The Gadget concept is discussed in Section 20.21: "Gadget reference" (p. 467). Each aggregator in a table is a gadget. So whenever a table is created from a table type, each aggregator type in that table type is called to produce its gadget, and these gadgets are collected in the table. When you call `table->getAggregatorLabel("name")`, you get the output label from the appropriate gadget.

The short summary: one AggregatorGadget per table per aggregator type.

The AggregatorGadget is a subclass of Gadget that keeps the extra information typically needed by all the aggregator types. It's defined in `sched/AggregatorGadget.h`. The original grand plan was that the different aggregator types may

define their own subclasses of AggregatorGadget but in reality there appears no need to. So far all the aggregators happily live with the base AggregatorGadget. It's constructed as:

```
AggregatorGadget(const AggregatorType *type, Table *table, IndexType *intype);
```

The arguments are pretty much a pass-through from the `makeGadget()`. The AggregatorGadget will keep references to the AggregatorType and to the IndexType, and a pointer to the Table, for the future use. The reason why the Table is not referenced is that it would create cyclic references, because the table already references all its aggregator gadgets. There is normally no need to worry that the table will disappear: when the table is destroyed, it will never call the aggregator gadget again. And that would remove the references to the Aggregator and AggregatorGadget, thus likely causing them to be destroyed too (unless you hold another reference to AggregatorGadget from outside Aggregator, which you normally should not).

This information can be obtained back from the AggregatorGadget with:

```
Table *getTable() const;
const AggregatorType *getType() const;
template<class C> const C *typeAs() const;
const IndexType *getIndexType() const;
```

The AggregatorType can be extracted in two ways, as a plain base class pointer with `getType()` and with the template `typeAs()` that casts it to the appropriate subclass. For example:

```
MyAggregatorType *agt = gadget->typeAs<MyAggregatorType>();
```

Of course, the subclasses can also read the fields directly.

The AggregatorGadget also publicly exports the method `sendDelayed()` of the Gadget (which is normally protected) and provides a convenience wrapper that constructs a row from fields data and sends it:

```
void sendDelayed(Tray *dest, const Row *row, Rowop::Opcode opcode) const
void sendDelayed(Tray *dest, FdataVec &data, Rowop::Opcode opcode) const;
```

The Gadget method `send()` is not exported, and is even marked as private. The rows are always sent from the aggregators in the delayed fashion. The reasons for that are partially historic, having to do with the per-Gadget enqueueing modes, but the bigger reason is that it also helps with the error handling inside the Table and Aggregator code, separating the errors in the Aggregators themselves from the errors in the labels called by them.

20.24.3. Aggregator reference

Unlike AggregatorGadget, an Aggergator represents a concrete aggregation group, on a concrete index (not on an index type, on an index!). Whenever an index of some type is created, an aggregator of its connected type is created with it. Remember, an index contains a single aggregation group. A table with nested indexes can have lots of aggregators of a single type. The difference between an index type and an index is explained in detail in Section 9.10: "The index tree" (p. 101), and the available Index methods are enumerated in Section 20.17: "Index reference" (p. 456).

The way it works, whenever some row in the table gets deleted or inserted, the table determines for each index type, which actual index in the tree (i.e. which group) got changed. Then for aggregation purposes, if that index has an aggregator on it, that aggregator is called to do its work on the group. It produces an output row or two (or maybe none, or it can get creative and produce many rows) for that group and sends it to the aggregator gadget of the same type.

The short summary: one Aggregator object per group, produces the updates when asked, sends them to the single common gadget.

The pointers to the Table and Gadget are passed to the `makeAggregator()` method for convenience, the Aggergator object doesn't need to remember them. Whenever it will be called, it will also be given these pointers as arguments. This is done in an attempt to reduce the amount of data stored per aggregator.

The Aggregator class (defined in `table/Aggregator.h`) is the base class for the per-aggregation-group objects. Its main purpose is in the handler method:

```
virtual void handle(Table *table, AggregatorGadget *gadget, Index *index,
  const IndexType *parentIndexType, GroupHandle *gh, Tray *dest,
  AggOp aggop, Rowop::Opcode opcode, RowHandle *rh);
```

In retrospect, this method might be better off belonging to the AggregatorType class, but for now it is what it is.

You would create a subclass that would get instantiated for every aggregation group. Then the handler would be called every time this group gets modified, exactly as it was described for the Perl API. The arguments are fundamentally the same as in Perl, just structured differently: they're passed directly instead of being hidden in an aggregation context. The C++ programmers are expected to behave responsibly and not try to mess with these parameters outside of the call scope, or prepare to meet the dire consequences.

Before digging more into the arguments, a few more words about the subclass structure. The subclass may define any additional fields to keep its aggregation state. If you're doing an additive aggregation, it allows you to keep the previous results. If you're doing the optimization of the deletes, it allows you to keep the previous sent row.

What if your aggregator keeps no state? You still have to make a separate Aggregator object for every group, and no, you can't just return NULL from `makeAggregator()`, and no, the Aggregators are not reference-countable, so you have to make a new copy of it for every group. This looks decidedly sub-optimal, and eventually I'll get around to straighten it out. The good news though is that most of the real aggerators keep the state anyway, so it doesn't matter much.

Now getting back to the arguments. Probably the easiest way is to start with showing how the Perl AggregatorContext calls translate to the C++ API.

```
$result = $ctx->groupSize();
```

```
    size_t sz = parentIndexType->groupSize(gh);
```

Get the size of the group. The result is of the type size_t. This is pretty much the only method of the IndexType base class that should be called directly, and only in the aggregation; and also pretty much the only use of the arguments `parentIndexType` and `gh`. The rest of the IndexType methods should be accessed through the similar methods in the Table, and I won't even document them. However if you really, really want to, you can find the description of the other methods in `type/IndexType.h` and call them in the aggregation as well.

```
$rowType = $ctx->resultType();
```

```
    const RowType *rt = gadget->getLabel()->getType();
```

Get the result row type.

```
$rh = $ctx->begin();
```

```
    RowHandle *rhi = index->begin();
```

Get the first row handle of the group, in the order of the index (or technically, if it's not a leaf index, of its first leaf sub-index). As usual, it would return NULL if the group is empty. The aggregators are a weird place where the Index methods are called directly and not through the Table API. You *could* use the Table API as well, by getting the index type with `index->getType()`, and then using it in the table API, but then you would also need to supply a sample row from the group to the Table API, and getting that sample is also done through `index->begin()`, so there is no way around it. Note that you can't just use the `rh` argument, since it might be the handle that had already been deleted from the table. Besides, going directly to the index is slightly more efficient, since it skips the step of finding the index by its type and a sample row.

It's fine to use a pointer to RowHandle instead of a reference here, since the handles are already held in the table which is guaranteed not to change while the iterators run.

Just as in Perl's AggregatorContext, there is no `index->end()`. When the end of the group is reached, the iteration will return a NULL handle.

```
$rh = $ctx->next($rh);

    rhi = index->next(rhi);
```

Get the handle of the next row (or NULL if that was the last one) in the order of the index. The NULL argument produces the NULL result.

```
$rh = $ctx->last();

    RowHandle *rhi = index->last();
```

Get the handle of the last row in the group in the default order. Returns NULL if the group is empty.

```
$rh = $ctx->beginIdx($idxType);
$rh = $ctx->endIdx($idxType);
$rh = $ctx->lastIdx($idxType);

    RowHandle *sample = index->begin();
    RowHandle *rhend =  table->nextGroupIdx(otherIndexType, sample);
    for (RowHandle *rhit = table->firstOfGroupIdx(otherIndexType, sample); rhit != rhend;
     rhit = table->nextIdx(otherIndexType, rhit)) {
      ...
    }
```

Unlike the Perl API of AggregatorContext, there are no direct analogs of `beginIdx()` and such in the C++ API. To get them in C++, you need to translate the iteration to another index type through the Table (and of course, just like in Perl, you would need somehow to get the reference or pointer to another index type into your aggregator, and that index type better be in the subtree of the `parentIndexType`). To translate through the Table, you take any row from the group, usually the first one, and use it with the table methods that accept a sample row.

```
$ctx->send($opcode, $row);

    gadget->sendDelayed(dest, row, opcode);
```

In Perl I've named this method just `send()` but in C++ it comes with its proper name of `sendDelayed()`.

```
$ctx->makeHashSend($opcode, $fieldName => $fieldValue, ...);

    gadget->sendDelayed(dest, fields, opcode);
```

This is a convenience wrapper that builds the row from the fields and sends it on.

And here goes the honest description of the handler arguments:

```
Table *table
```
    Table where this Aggregator belongs.

```
AggregatorGadget *gadget
```
    The Gadget where this Aggregator sends its results.

```
Index *index
```
    Index that defines the group on which this Aggregator runs.

```
const IndexType *parentIndexType
```
    The IndexType of the parent Index, one level above the `index` argument. It's needed mostly because the group size is kept at that level.

GroupHandle *gh
This is an opaque object that can be used as an argument to the `parentIndexType` methods. It's an internal object that ties together all the indexes in the group under the parent index.

Tray *dest
The tray where the results will be collected. This tray is created and then processed by the Table logic.

AggOp aggop
The aggregation operation, with the same meaning as described for Perl in Section 11.2: "Manual aggregation" (p. 143) . The enum `AggOp` is defined in the Aggregator class and contains the elemends AO_BEFORE_MOD, AO_AFTER_DELETE, AO_AFTER_INSERT, AO_COLLAPSE.

Rowop::Opcode opcode
The suggested opcode for the result rowops. The meaning is also as in Perl.

RowHandle *rh
The handle that is about to or had just been inserted or removed, depending on the `aggop`. It may be NULL for the operation AO_COLLAPSE.

The methods to convert the `AO_*` constants to and from the strings are also defined in the Aggregator class:

```
static const char *aggOpString(int code, const char *def = "???");
static int stringAggOp(const char *code);
```

They work in the same way as the other constant conversion methods.

20.24.4. BasicAggregatorType reference

In many cases a simple stateless aggregation is good enough. For that, you don't need to define the whole set of classes for your aggregation, you can use the BasicAggregatorType (defined in `type/BasicAggregatorType.h`) instead.

You just define a simple C-style function to compute the aggregation and pass it to the BasicAggregatorType constructor:

```
BasicAggregatorType(const string &name, const RowType *rt, Callback *cb);
```

This function has all the arguments of Aggregator::handle forwarded to it:

```
typedef void Callback(Table *table, AggregatorGadget *gadget, Index *index,
  const IndexType *parentIndexType, GroupHandle *gh, Tray *dest,
  Aggregator::AggOp aggop, Rowop::Opcode opcode, RowHandle *rh, Tray *copyTray);
```

BasicAggregatorType takes care of the rest of the infrastructure: gadgets, aggregators etc.

If you need to pass some additional information to this function, you do it by extending the BasicAggregatorType class. Add your extra fields to your subclass, and then the callback function can find the type object from the gadget, and read the values from there.

20.24.5. Aggegator example

Since the aggregator classes are somewhat convoluted and substantially different from the Perl version, I've decided to provide not just the reference but also a couple of small examples. The full text can be found in the unit test file `table/test/t_Aggr.cpp`.

First, if your aggregator is truly stateless and fully hardcoded, the easier way to do it as by defining a plain function with the same handler arguments and building a BasicAggregatorType with it. And here is one that sums the values of an int64 field (the test case `aggBasicSum`):

```
void sumC(Table *table, AggregatorGadget *gadget, Index *index,
  const IndexType *parentIndexType, GroupHandle *gh, Tray *dest,
```

```
    Aggregator::AggOp aggop, Rowop::Opcode opcode, RowHandle *rh)
{
  // don't send the NULL record after the group becomes empty
  if (opcode == Rowop::OP_NOP || parentIndexType->groupSize(gh) == 0)
    return;

  int64_t sum = 0;
  for (RowHandle *rhi = index->begin(); rhi != NULL; rhi = index->next(rhi)) {
    sum += table->getRowType()->getInt64(rhi->getRow(), 2, 0); // field c at idx 2
  }

  // pick the rest of fields from the last row of the group
  RowHandle *lastrh = index->last();

  // build the result row; relies on the aggregator result being of the
  // same type as the rows in the table
  FdataVec fields;
  table->getRowType()->splitInto(lastrh->getRow(), fields);
  fields[2].setPtr(true, &sum, sizeof(sum)); // set the field c from the sum

  // could use the table row type again, but to exercise a different code,
  // use the aggregator's result type:
  // gadget()->getType()->getRowType() and gadget->getLabel()->getType()
  // are equivalent
  Rowref res(gadget->getLabel()->getType(), fields);
  gadget->sendDelayed(dest, res, opcode);
}

...
  Autoref<TableType> tt = TableType::make(rt1)
    ->addSubIndex("Hashed", HashedIndexType::make( // will be the default index
        (new NameSet())->add("e")
      )->addSubIndex("Fifo", FifoIndexType::make()
        ->setAggregator(new BasicAggregatorType("aggr", rt1, sumC))
      )
    );
...
```

As described above, you create the BasicAggregatorType by giving it the aggregator name, aggregator result row type, and the handler function.

In this case the handler function is completely hardcoded. It works on the int64 field at index 2. The row type I used in this example is:

```
row {
  uint8[10] a,
  int32[] b,
  int64 c,
  float64 d,
  string e,
}
```

So the field is actually named "c", and that's why the aggregator function is named "sumC". But since in this case everything is known in advance, to make it more efficient, the look-up of the field by name has been skipped, and the field index has been pre-computed and hardcoded into the function.

The general outline of the aggregator is exactly the same as in Perl: check for an empty group, then iterate through all the rows in the group and compute a the sum, fill the rest of the fields from the last row, and send the result. The difference is that there is no AggregatorContext, and the calls are done directly on the bits and pieces received as arguments.

The input row type for reading the rows from the group is found as:

```
table->getRowType()
```

The result row is built in this example by copying the last row and replacing one field. The data from the last row is split into FdataVec (the data itself is not copied at this point but the data descriptors in the construction vector are made to point to the data in the original row). Then the descriptor for the field "c" is changed to point to the computed sum. Then the new row is built from the descriptor.

In this particular case the type of the input rows and of the result rows is the same, so either could have been used to construct the result rows. There are two ways to find the result type:

```
gadget->getType()->getRowType()
gadget->getLabel()->getType()
```

They are exactly the same, just there are two paths leading to the same object.

Finally, the constructed row is sent. `sendDelayed()` takes care of constructing the rowop from the components. The version of `sendDelayed()` that also constructs the row from the fields is shown in the next example.

And now on to the next example.

Doing a proper custom aggregator is more involved, and requires making subclasses of both Aggregator and AggregatorType. The test case `aggSum` shows an example of aggregator that can sum any 64-bit field. It still could be done with subclassing the BasicAggregatorType, since it still doesn't keep any group state, but I wanted to show a full-size example as well.

The subclass of Aggregator contains only one method that is very similar to the BasicAggregator handler shown before:

```
class MySumAggregator: public Aggregator
{
public:
  // same as sumC but finds the field from the type
  virtual void handle(Table *table, AggregatorGadget *gadget, Index *index,
    const IndexType *parentIndexType, GroupHandle *gh, Tray *dest,
    Aggregator::AggOp aggop, Rowop::Opcode opcode, RowHandle *rh);
};

void MySumAggregator::handle(Table *table, AggregatorGadget *gadget, Index *index,
  const IndexType *parentIndexType, GroupHandle *gh, Tray *dest,
  Aggregator::AggOp aggop, Rowop::Opcode opcode, RowHandle *rh)
{
  // don't send the NULL record after the group becomes empty
  if (opcode == Rowop::OP_NOP || parentIndexType->groupSize(gh) == 0)
    return;

  int fidx = gadget->typeAs<MySumAggregatorType>()->fieldIdx();

  int64_t sum = 0;
  for (RowHandle *rhi = index->begin(); rhi != NULL; rhi = index->next(rhi)) {
    sum += table->getRowType()->getInt64(rhi->getRow(), fidx, 0);
  }

  // pick the rest of fields from the last row of the group
  RowHandle *lastrh = index->last();

  // build the result row; relies on the aggregator result being of the
  // same type as the rows in the table
  FdataVec fields;
  table->getRowType()->splitInto(lastrh->getRow(), fields);
  fields[fidx].setPtr(true, &sum, sizeof(sum));
```

```
  // use the convenience wrapper version
  gadget->sendDelayed(dest, fields, opcode);
}
```

The difference is that the field index is not hardcoded but taken from the aggregator type. The aggregator type is found with

```
gadget->typeAs<MySumAggregatorType>()
```

The method `fieldIdx()` is a custom addition to the MySumAggregatorType, not inherited from any base class.

The version of `AggregatorGadget::sendDelayed()` used here takes care of constructing the row from the fields and then sends it on.

Then the aggregator type needs to be defined with a fixed set of inherited virtual methods plus any needed custom parts.

```
class MySumAggregatorType: public AggregatorType
{
public:
  // @param fname - field name to sum on
  MySumAggregatorType(const string &name, const string &fname):
    AggregatorType(name, NULL),
    fname_(fname),
    fidx_(-1)
  { }
  // the copy constructor works fine
  // (might set the non-NULL row type, but it will be overwritten
  // during initialization)

  // constructor for deep copy
  // (might set the non-NULL row type, but it will be overwritten
  // during initialization)
  MySumAggregatorType(const MySumAggregatorType &agg, HoldRowTypes *holder):
    AggregatorType(agg, holder),
    fname_(agg.fname_),
    fidx_(agg.fidx_)
  { }

  virtual AggregatorType *copy() const
  {
    return new MySumAggregatorType(*this);
  }

  virtual AggregatorType *deepCopy(HoldRowTypes *holder) const
  {
    return new MySumAggregatorType(*this, holder);
  }

  virtual bool equals(const Type *t) const
  {
    if (this == t)
      return true; // self-comparison, shortcut

    if (!AggregatorType::equals(t))
      return false;

    const MySumAggregatorType *sumt = static_cast<const MySumAggregatorType *>(t);

    if (fname_ != sumt->fname_)
      return false;

    return true;
  }
```

```
  virtual bool match(const Type *t) const
  {
    if (this == t)
      return true; // self-comparison, shortcut

    if (!AggregatorType::match(t))
      return false;

    const MySumAggregatorType *sumt = static_cast<const MySumAggregatorType *>(t);

    if (fname_ != sumt->fname_)
      return false;

    return true;
  }

  virtual AggregatorGadget *makeGadget(Table *table, IndexType *intype) const
  {
    return new AggregatorGadget(this, table, intype);
  }

  virtual Aggregator *makeAggregator(Table *table, AggregatorGadget *gadget) const
  {
    return new MySumAggregator;
  }

  virtual void initialize(TableType *tabtype, IndexType *intype)
  {
    const RowType *rt = tabtype->rowType();
    setRowType(rt); // the result has the same type as the argument
    fidx_ = rt->findIdx(fname_);
    if (fidx_ < 0)
      errors_.fAppend(new Errors(rt->print()), "Unknown field '%s' in the row type:",
 fname_.c_str());
    else {
      if (rt->fields()[fidx_].arsz_ != RowType::Field::AR_SCALAR
      || rt->fields()[fidx_].type_->getTypeId() != Type::TT_INT64)
        errors_.fAppend(new Errors(rt->print()),
          "Field '%s' is not an int64 scalar in the row type:", fname_.c_str());
    }
    AggregatorType::initialize(tabtype, intype);
  }

  // called from the handler
  int fieldIdx() const
  {
    return fidx_;
  }

protected:
  string fname_; // name of the field to sum, must be an int64
  int fidx_; // index of field named fname_
};
```

The constructor accepts the aggregator name and the name of the field on which it will sum. The field name will be
translated to field index during initialization, and made available to the `MySumAggregator::handler()` via the
method `fieldIdx()`. The aggregator type starts with the result row type of NULL, with the actual row type set during
initialization. The idea here is that the result row type of this aggregator is always equal to the input row type, so rather
than specifying the result type explicitly and then having to check it for compatibility, why not just take the table's row
type when it becomes available? And it works beautifully.

The copy constructor and the constructor with HoldRowTypes are the implementations of the virtual methods `copy()` and `deepCopy()`. The `deepCopy()` is used in the multithreaded support for passing the table types through the nexus. See Section 20.4: "The many ways to do a copy" (p. 424)and Section 16.4: "Object passing between threads" (p. 297) for details.

The methods `match()` and `equals()` follow the same general shape as everywhere else. `makeGadget()` creates a generic gadget, and `makeAggregator()` creates an instance of aggregator for each group.

The interesting stuff starts happening in the initialization. The row type gets found from the table and set as the result type. Then the aggregation field is found in the row type and checked for being of the proper type. Its index is remembered for the later use.

`errors_.fAppend()` makes the error construction more convenient. It is smart enough to check the reference for NULL and allocate a new Errors if so, then append a printf-formatted message and nested errors to it.

20.25. Unit reference

The Unit is declared in `sched/Unit.h`. It represents an execution unit that controls the execution order of a sequential part of a model. An unit is an Mtarget but in reality it may be used in one thread only. The multithreading support in the references is needed to build the multithreaded applications but each Unit in the application belongs to one thread (possibly sharing it with other Units).

```
Unit(const string &name);
```

Constructs an execution unit.

```
const string &getName() const;
```

Get back the name.

```
void schedule(Onceref<Rowop> rop);
void scheduleTray(const_Onceref<Tray> tray);
void fork(Onceref<Rowop> rop);
void forkTray(const_Onceref<Tray> tray);
void call(Onceref<Rowop> rop);
void callTray(const_Onceref<Tray> tray);

void enqueue(int em, Onceref<Rowop> rop);
void enqueueTray(int em, const_Onceref<Tray> tray);
```

Schedule, fork or call a rowop or tray, like in Perl. Unlike Perl, the methods with a tray argument have different names. And the enqueueing mode is always an integer constant. These constants are defined in the enum Gadget::EnqMode (the Gadget class is described in Section 20.21: "Gadget reference" (p. 467)), and is one of `Gadget::EM_SCHEDULE`, `Gadget::EM_FORK`, `Gadget::EM_CALL` and `Gadget::EM_IGNORE`.

```
void enqueueDelayedTray(const_Onceref<Tray> tray);
```

Enqueue (schedule, fork or call) the rowops from a tray built by Gadget's method `sendDelayed()`. The Rowop has a semi-undocumented field to store the enqueuing mode for that particular Rowop, that gets populated by `Gadget::sendDelayed()`. The `enqueueDelayedTray()` uses the value from this field to determine the enqueueing method individually for each Rowop. The whole arraggement is largely obsolete: it used to be used for the aggregators in a Table, mixing the outputs from multiple aggregators with potentially different enqueueing methods in one tray. But now everything just uses `EM_CALL` and there is no need for this any more.

```
bool empty() const;
```

Check whether all the Unit's frames are empty.

```
bool isFrameEmpty() const;
```

Check whether the current inner frame is empty.

```
bool isInOuterFrame() const;
```

Check whether the unit's current inner frame is the same as its outer frame, which means that the unit is not in the middle of a call.

```
void callNext();
void drainFrame();
```

Execute the next rowop from the current (inner) frame, or all the rowops on the current frame. The semantics is the same as in the Perl code.

```
void setMark(Onceref<FrameMark> mark);
```

Set a mark on the current frame, same as in Perl.

```
void loopAt(FrameMark *mark, Onceref<Rowop> rop);
void loopTrayAt(FrameMark *mark, const_Onceref<Tray> tray);
```

Enqueue a rowop or a tray at the marked frame.

```
void callAsChained(const Label *label, Rowop *rop, const Label *chainedFrom);
```

This method was introduced in Triceps version 2, and hasn't propagated to Perl yet. I'm not even sure that I want it visible in Perl, since it's kind of low-level. It executes a label call, assuming that it was chained from another label (before version 2 the functionality itself had obviously existed but was not visible in the API).

The row types of all the arguments must be matching, or the method would likely crash.

The label will be called with rowop, just as if this label was chained from another label. Since the chaining might be multi-level, the `chainedFrom` label is not necessarily the one in the rowop, it can itself be chained from the label in the rowop, directly or indirectly.

All the tracing for the chained calls is automatically invoked. Keeping the consistency of the tracing is up to you: if you use the `chainedFrom` argument label that hasn't actually been called through the chain, the trace will look surprising.

This method is used in Triceps in such places as calling a label in FnBinding through an FnReturn. You can use it directly as well, just be very careful.

```
void clearLabels();
```

Clear all the unit's labels, same semantics as when called from Perl.

```
void rememberLabel(Label *lab);
```

The method that connects a label to the unit. Normally you don't need to call it manually, the label constructor calls it (and that's why it's not in the Perl API). The only real reason to use this method manually is if you've disconnected a label manually from the unit, and want to reconnect it back (and I'm not sure if anyone would ever want that). Calling this method repeatedly with the same label and unit will have no effect. Remembering the same label in multiple units is not a good idea.

```
void  forgetLabel(Label *lab);
```

Make the unit forget a label, so on `clearLabels()` that label won't be cleared. This is another dangerous low-level method, since only the unit will forget about the label but the label will still keep the pointer to the unit, unless it's cleared. Because of the danger, it's also not in the Perl API. The reason to use it would be if you want to disassemble and discard a part of the unit without disturbing the rest of it. However a safer alternative is to just create multiple units in one thread and discard by a whole unit.

```
RowType *getEmptyRowType() const;
```

A convenience method to get a reference to a row type with no fields. Such row type is useful for creation of pseudo-labels that have the user-defined clearing handlers that clear some user data.

Each unit has its own instance of an empty row type. The separate instances (instead of having a single global one) are purely for the conveniece of memory management in the threads, they are all equivalent. You could also create your own empty row type.

```
void setMaxStackDepth(int v);
int maxStackDepth() const;
void setMaxRecursionDepth(int v);
int maxRecursionDepth() const;
```

Set and get the maximal unit stack depth and recursion depth, works the same as in Perl.

```
int getStackDepth() const;
```

Returns the current depth of the execution stack. 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.

```
void clearLabels();
```

Clear all the labels in the unit, then drop the references from unit to them. Normally should be called only when the unit is about to be destroyed, and is called automatically when the unit gets destroyed. However in case of cyclic references to the unit, may need to be called manually or through a UnitClearingTrigger object to break these cycles. For more information, see Section 8.1: "Reference cycles" (p. 77) .

The Unit has a few helper classes: the tracers and the clearing triggers.

The tracer classes inherit from the nested virtual class Unit::Tracer. They will be described in Section 20.26: "Unit Tracer reference" (p. 487) . The Unit class contains the definition of the enums of the tracers and provides a way to set the tracer in the unit.

```
enum TracerWhen {
  // The values go starting from 0 in before-after pairs
  TW_BEFORE, // before calling the label's execution as such
  TW_AFTER, // after all the execution is done
  TW_BEFORE_CHAINED, // after execution, before calling the chained labels (if they are
 present)
  TW_AFTER_CHAINED, // after calling the chained labels (if they were present)
  TW_BEFORE_DRAIN, // before draining the label's frame if it's not empty
  TW_AFTER_DRAIN, // after draining the label's frame if was not empty
};
```

This enum describes the point of the tracer invocation. The meaning of the values is described in detail in Section 7.10: "Tracing the execution" (p. 63) .

```
static const char *tracerWhenString(int when, const char *def = "???");
static int stringTracerWhen(const char *when);
```

The usual conversions between the integer constants and their string representations. def is as usual the default placeholder that will be used for an invalid value. And the conversion from string would return a -1 on an invalid value.

```
static const char *tracerWhenHumanString(int when, const char *def = "???");
static int humanStringTracerWhen(const char *when);
```

The same kind of conversion as above, only the strings are more human-readable. Basically, the same thing, only in the lowercase words. For example, TW_BEFORE_CHAINED would become "before-chained". This conversion is used by the provided tracer classes in their traces.

```
static bool tracerWhenIsBefore(int when);
static bool tracerWhenIsAfter(int when);
```

Check a TracerWhen value for whether it represents a point before or after calling a label. Right now all the valid values fall into one or the other classification. But more trace points that are neither "before" or "after" could get added in the future, so a good practice is to check explicitly for both conditions rather than a simple if/else with one condition.

```
void setTracer(Onceref<Tracer> tracer);
Onceref<Tracer> getTracer() const;
```

Set and get back the tracer object for the Unit. NULL means that no tracer is set. It's possible to use the same tracer object for multiple units, as long as these units are in the same threads; that will combine the traces from these units into one trace. If you really want, it's even possible to use the same tracer object for the units in different threads but for that you must define your own tracer class that does the proper synchronization. None of the pre-defined tracers have the synchronization, since using them from multiple threads is usually not a good idea.

The clearing trigger objects are constructed with:

```
UnitClearingTrigger(Unit *unit);
```

The trigger is an Mtarget, so the typical use would be:

```
{
  Autoref<UnitClearingTrigger> ctrig = new UnitClearingTrigger(myunit);
  ...
}
```

At the block exit when the Autoref will get destroyed, it will destroy the trigger, which would in turn cause the clearing of the unit. Of course, you can also place the Autoref into another object, and then the destruction of that object would cause the clearing, instead of the end of the block.

20.26. Unit Tracer reference

Unlike Perl, in C++ the tracer is defined by inheriting from the class Unit::Tracer. The base class inherits from the Mtarget, and in the subclass you need at the minimum to define your virtual method:

```
virtual void execute(Unit *unit, const Label *label, const Label
  *fromLabel, Rowop *rop, TracerWhen when);
```

It gets called at the exactly same points as the Perl tracer (the C++ part of the UnitTracerPerl forwards the calls to the Perl level). The arguments are also the same as described in the Perl docs:

- unit is the unit being traced.

- label is the current label being traced.

- fromLabel is the parent label in the chaining (would be NULL if the current label is called directly, without chaining from anything).

- rop is the current row operation.

- when is a constant showing the point when the tracer is being called. It's value may be one of Unit::TW_BEFORE, Unit::TW_AFTER, Unit::TW_BEFORE_DRAIN, Unit::TW_AFTER_DRAIN, Unit::TW_BEFORE_CHAINED, Unit::TW_AFTER_CHAINED; the prefix TW stands for "tracer when".

Here is a simple example of a tracer:

```
class SampleTracer : public Unit::Tracer
{
public:
```

```
  virtual void execute(Unit *unit, const Label *label,
    const Label *fromLabel, Rowop *rop, Unit::TracerWhen when)
  {
    printf("trace %s label '%s' %c\n", Unit::tracerWhenHumanString(when),
      label->getName().c_str(), Unit::tracerWhenIsBefore(when)? '{' : '}');
  }
};
```

The Unit methods shown are documented in Section 20.25: "Unit reference" (p. 484) .

The SampleTracer above was just printing the trace right away. Usually a better idea is to save the trace in the tracer object and return it on demand. Unit::Tracer provides the means to do that: an Erref object is used as a buffer, where the data can be added efficiently line-by-line, and later read back. The buffer is defined protected, so that the subclasses can access it but the end users can't:

```
protected:
  Erref buffer_; // buffer for collecting the trace
```

On each call the tracer's `execute()` would build a string, and append it to the buffer:

```
buffer_->appendMsg(false, traceString);
```

The public interface for the buffer is:

```
virtual Erref getBuffer();
virtual void clearBuffer();
```

These methods are virtual to let the subclasses define their own version if they want. The methods do exactly what their names say. `getBuffer()` returns the reference to the buffer in `buffer_`, after which the end user has the full access to it.

The pattern of reading the buffer contents works like this:

```
string tlog = trace->getBuffer()->print();
trace->clearBuffer();
```

The log string can then be printed, or used in any other way.

The clearing of the buffer is done by discarding the old one and allocating a new one. An interesting consequence is that if you call `getBuffer()`, save the reference, then call `clearBuffer()`, the original buffer is still available at your saved reference and won't be changed any more, for example:

```
Erref buf = trace->getBuffer();
trace->clearBuffer();
string tlog = buf->print();
```

Of course, if a tracer subclass defines its own version of virtual methods, it may change this semantics.

It can be quite useful to include the row being processed into the trace, as the Perl examples show in Section 7.10: "Tracing the execution" (p. 63) . The C++ part of Triceps doesn't provide a ready row printer yet but Unit::Tracer provides a way for you to use your own row printer. Even two ways. One is to re-define the public method `printRow()`:

```
virtual void printRow(string &res, const RowType *rt, const Row *row);
```

The job of this method is to append the information from the row `row` of the type `rt` to the result string `res`. *Append*, not replace.

The second way is by providing a pointer to a simple C-style function of the type:

```
typedef void RowPrinter(string &res, const RowType *rt, const Row *row);
```

The arguments and functionality are exactly the same as for the method `printRow()`. The base class implementation of `printRow()` works by calling this function pointer of it's not NULL. This pointer can be given to the Tracer constructor:

```
Tracer(RowPrinter *rp = NULL);
```

In general, the subclasses should also take this argument and pass it through to the parent class. This allows the separation of the tracer itself and of the row printing: the user becomes capable of specifying any row printer. Since the default row printer is NULL, the row contents doesn't get printed by default.

Triceps provides a couple of stock tracer classes:

StringTracer
    Collects the trace in a buffer, identifying the objects as addresses. This is not exactly easy to read normally but may come useful if you want to analyze a core dump.

StringNameTracer
    Similar but prints the object identification as names. More convenient but prone to the duplicate names used for different objects.

Everything but the constructors of these classes follows the general Tracer interface. The constructors are:

```
StringTracer(bool verbose = false, RowPrinter *rp = NULL);
StringNameTracer(bool verbose = false, RowPrinter *rp = NULL);
```

The `verbose` flag enables the tracing at all points, if it's set to `false` then only the `TW_BEFORE` points are traced. The argument `rp` lets the end user supply a row printer function.

The tracing does not have to be used just for tracing. It can also be used as a breakpoint: check in your tracer for an arbitrary condition and stop if it has been met.

There is only one tracer per unit at a time. However if you want, you can implement the chaining in your own tracer (particularly useful if it's a breakpoint tracer): accept a reference to another tracer object, and after doing your own part, call that object's `execute()` method.

Even though the Tracer class inherits from Mtarget, none of its methods have the synchronization in them. Thus each object must be used in only one thread. If you really want to use your tracer object from multiple threads, define your own subclass and re-implement all the virtual methods in it with synchronization.

20.27. Label reference

In C++ the custom labels are defined by defining your own class that inherits from Label (which is defined in `sched/Label.h`). Like this small example from a unit test:

```
class ForkingLabel : public Label
{
public:
  ForkingLabel(Unit *unit, Onceref<RowType> rtype, const string &name,
      Onceref<Label> next) :
    Label(unit, rtype, name),
    next_(next)
  { }

  virtual void execute(Rowop *arg) const
  {
    unit_->fork(next_->adopt(arg));
  }

  Autoref<Label> next_;
};
```

The subclass needs to define its own execution method:

```
virtual void execute(Rowop *arg) const;
```

The base class takes care of all the general execution mechanics, chaining etc. All you need to do in this method is perform your user-defined actions. By the way, this method is protected and should never be called directly. The labels must always be called through a unit, which will then execute them in the correct way.

The subclass may (though doesn't have to) also define the custom clearing method:

```
virtual void clearSubclass();
```

Currently this method is called by the public method `clear()` after the label is marked as cleared but before clearing of the chain (though this order may change in the future).

The base class constructor is also protected, it's always constructed from a subclass. You can not instantiate the base Label class because it contains an abstract `execute()` method.

```
Label(Unit *unit, const_Onceref<RowType> rtype, const string &name);
```

The arguments are similar to the Perl version, only the Unit is passed as an argument to the Label constructor.

The constructed label keeps a reference to its row type, and a pointer (not reference, to avoid the circular references!) to the unit. The unit automatically keeps a reference to the label, so there is no need to call `Unit::rememberLabel()` manually. On the other hand, if you do not want the label to be remembered by its unit (though why?), the only way to achieve that is to call `Unit::forgetLabel()` after its construction.

Now, the public methods of Label.

```
const string &getName() const;
const RowType *getType() const;
Unit *getUnitPtr() const;
```

Read back the information from the constructor. The method `getUnitPtr()` is named this way and not `getUnit()` to emphasize that the Label has only a pointer to the Unit, not a reference. After the label gets cleared, `getUnitPtr()` will return NULL.The reason is that after that the label doesn't know any more whether the unit still exists or has been deleted, and doesn't want to return a pointer to a potentially freed memory.

```
const string &getUnitName() const;
```

A convenience method for the special case of getting the label's unit name. It's used in many error message. You can't just say `label->getUnitPtr()->getName()` because `getUnitPtr()` might return a NULL if the label has been already cleared. `getUnitName()` takes care of it and returns a special string "[label cleared]" if the label has been cleared.

```
void clear();
```

Clears the label. After that the label stops working. Note that clearing a label doesn't disconnect it from its unit. Well, the label won't tell you its unit any more (the pointer will be reset to NULL) but the unit will still have a reference to the label! Use the unit's method `forgetLabel()` to disconnect it (but that won't clear the label itself, so you have to call both `unit->forgetLabel()` and `label->clear()`). Of course, if you call `unit->clearLabels()`, that would take care of everything.

Clearing cleans the chaining list of this label but doesn't call recursively `clear()` on the formerly chained labels. If you need that, you have to do it yourself.

```
bool isCleared() const;
```

Check if the label has been cleared.

```
void setNonReentrant();
bool isNonReentrant() const;
```

Mark the label as non-reentrant, and check this flag. There is no way to unset this flag. The meaning of it is described in Section 7.13: "Recursion control" (p. 74) .

```
Erref chain(Onceref<Label> lab, bool front = false);
```

Chain another label to this one (so when this label is executed, the chained labels will also be executed in order). This label will keep a reference of the chained label. The circular chainings are forbidden and will throw an Exception. If the argument `front` is `false` (default), the chaining will be done at the back of the chain, if `true` then at the front, same as the method `chainFront()` in the Perl API. In the C++ API both kinds of chainings are done with the same method.

```
typedef vector<Autoref<Label> > ChainedVec;
const ChainedVec &getChain() const;
```

Get back the information about the chained labels. This returns a reference to the internal vector, so if the chainings are changed afterwards, the changes will be visible in the vector.

```
bool hasChained() const;
```

A quick check, whether there is anything chained.

```
void clearChained();
```

Clear the chaining list of this label. (But doesn't call `clear()` on these labels!)

```
Rowop *adopt(Rowop *from) const;
```

A convenient factory method for adopting the rowops. Treat it similarly to a constructor: the returned Rowop will be newly constructed and have the reference count of 0; the returned pointer must be stored in an Autoref (or Onceref). This method by itself doesn't check whether the original Rowop has a matching type, it simply makes a copy with the label reference replaced. It's up to you to make sure that the labels are correct.

A special subclass of the Label is DummyLabel: it's a label that does nothing. It's `execute()` method is empty. It's constructed very similarly to the normal Label:

```
DummyLabel(Unit *unit, const_Onceref<RowType> rtype, const string &name);
```

The dummy labels are convenient for chaining the other labels to them.

20.28. Rowop reference

The Rowop class is defined in `sched/Rowop.h` though usually there is no need to include it directly, just include `sched/Unit.h` which will pull in the rest of the scheduling classes.

The Rowop in C++ consists of all the same parts as in Perl API: a label, a row, and opcode.

It has one more item that's not really visible in the Perl API, the enqueueing mode, but it's semi-hidden in the C++ API as well. The only place where it's used is in `Unit::enqueueDelayedTray()`. This allows to build a tray of rowops, each with its own enqueueing mode, and then enqueue all of them appropriately in one go. This feature is kind of historic and will be removed somewhere soon.

The Rowop class inherits from Starget, usable in one thread only. Since it refers to the Labels, that are by definition single-threaded, this makes sense. A consequence is that you can't simply pass the Rowops between the threads. The passing-between-threads is handled by the Nexuses, performing the translation of the Labels along the way.

The opcodes are defined in the enum Rowop::Opcode, so you normally use them as `Rowop::OP_INSERT` etc. The opcodes actually contain a bitmap of individual flags, defined in the enum Rowop::OpcodeFlags: `Rowop::OCF_INSERT` and `Rowop::OCF_DELETE`. You don't really need to use these flags directly unless you really, really want to.

Besides the 3 already described opcodes (OP_NOP, OP_INSERT and OP_DELETE) there is another one, OP_BAD. It's a special value returned by the string-to-opcode conversion method instead of the -1 returned by the other similar method.

The reason is that OP_BAD is specially formatted to be understood by all the normal opcode type checks as NOP (i.e. both flags OCF_INSERT and OCF_DELETE are reset in it), while -1 would be seen as a combination of INSERT and DELETE. So if you miss checking the result of conversion on a bad string, at least you would get a NOP and not some mysterious operation. The reason why OP_BAD is not exported to Perl is that in Perl an `undef` is used as the indication of the invalid value, and works even better.

There is a pretty wide variety of Rowop constructors:

```
Rowop(const Label *label, Opcode op, const Row *row);
Rowop(const Label *label, Opcode op, const Rowref &row);

Rowop(const Label *label, Opcode op, const Row *row, int enqMode);
Rowop(const Label *label, Opcode op, const Rowref &row, int enqMode);

Rowop(const Rowop &orig);
Rowop(const Label *label, const Rowop *orig);
```

The constructors with the explicit `enqMode` are best not be used outside of the Triceps internals, and will eventually be obsoleted. The last two are the copy constructor, and the adoption constructor which underlies `Label::adopt()` and can as well be used directly.

Once a rowop is constructed, its components can not be changed any more, only read.

```
Opcode getOpcode() const;
const Label *getLabel() const;
const Row *getRow() const;
int getEnqMode() const;
```

Read back the components of the Rowop. Again, the `getEnqMode()` is on the way to obsolescence. And if you need to check the opcode for being an insert or delete, the better way is to use the explicit test methods, rather than getting the opcode and comparing it for equality:

```
bool isInsert() const;
bool isDelete() const;
bool isNop() const;
```

Check whether the opcode requests an insert or delete (or neither).

The same checks are available as static methods that can be used on the opcode values:

```
static bool isInsert(int op);
static bool isDelete(int op);
static bool isNop(int op);
```

And the final part is the conversion between the strings and integer values for the Opcode and OpcodeFlags enums:

```
static const char *opcodeString(int code);
static int stringOpcode(const char *op);
static const char *ocfString(int flag, const char *def = "???");
static int stringOcf(const char *flag);
```

As mentioned above, `stringOpcode()` returns OP_BAD for the unknown strings, not -1.

20.29. Tray reference

A Tray in C++, defined in `shed/Tray.h`, is simply a deque of Rowop references, plus an Starget, so that it can be reference-counted like the rest of Triceps classes:

```
class Tray : public Starget, public deque< Autoref<Rowop> >
```

All it really defines is the constructors:

```
Tray();
Tray(const Tray &orig);
```

The operations on the Tray are just the usual deque operations.

Yes, you can copy the trays by constructing a new one from an old one:

```
Autoref<Tray> t1 = new Tray;
t1->push_back(op1);

Autoref<Tray> t3 = new Tray(*t1);
```

Afterwards `t3` will contain references to the same rowops as `t1` (but will be a different Tray than `t1`!).

The assignments (`operator=`) happen to just work out of the box because the `operator=` implementation in Starget does the smart thing and avoids the corruption of the reference counter. So you can do things like

```
*t3 = *t1;
```

It's worth noting once more that unlike Rows and Rowops, the Trays are mutable. If you have multiple references to the same Tray, modifying the Tray will make the new contents visible through all the references!

An important difference from the Perl API is that in C++ the Tray is not associated with a Unit. It's constructed simply by calling its constructor, and there is no Unit involved. It's possible to create a tray that contains a mix of rowops for different units. If you combine the C++ and Perl code, and then create such mixes in the C++ part, the Perl part of your code won't be happy.

And there is actually a way to create the mixed-unit trays even in the Perl code, in the tray of FnBinding. But this situation would be caught when trying to get the tray from FnBinding into the Perl level, and the workaround for handling them is to use the method `FnBinding:callTray()`.

The reason why Perl associates the trays with a unit is to make the check of enqueueing a tray easy: just check that the tray belongs to the right unit, and it's all guaranteed to be right. At the C++ level no such checks are made. If you enqueue the rowops on labels belonging to a wrong unit, they will be enqueued quietly, will attempt to execute, and from there everything will likely go very wrong. So be disciplined. And maybe I'll think of a better way for keeping the unit consistency in the future.

20.30. FrameMark reference

The FrameMark (defined in `sched/FrameMark.h`) marks the unit's frame at the start of the loop, to fork there the rowops for the next iterations of the loop. It's pretty simple:

```
FrameMark(const string &name);
```

The constructor. It gives the mark a name. A FrameMark is an Starget, so it may be used only in one thread.

```
const string &getName() const;
```

Read back the name.

```
Unit *getUnit() const;
```

This method is different from `getUnit()` on most of the other classes. It returns the pointer to the unit, on which it has been set. A freshly created FrameMark would return NULL. Internally a FrameMark doesn't keep a reference to the unit, it's just a pointer, and a way for the Unit to check in `loopAt()` that the mark has been indeed set on this unit (it would refuse to fork the rowops there otherwise). And you can use it for the entertainment purposes too. Normally when the frame marked with this mark gets popped from the Unit's stack, the mark becomes unset, and its `getUnit()` will return NULL.

All the actions on the FrameMark are done by passing it to the appropriate methods of the Unit. When a mark is set on a frame, the frame has a reference to it, so the mark won't be destroyed until the frame is freed.

20.31. RowSetType reference

RowSetType, defined in `types/RowSetType.h`, is another item that is not visible in Perl. Maybe it will be in the future but at the moment things look good enough without it. It expresses the type ("return type" if you want to be precise) of a streaming function (FnReturn and FnBinding classes). Naturally, it's a sequence of the row types, and despite the word "set", the order of its elements matters.

A RowSetType is one of these objects that get assembled from many parts and then initialized, like this:

```
Autoref<RowSetType> rst = initializeOrThrow(RowSetType::make()
  ->addRow("name1", rt1)
  ->addRow("name2", rt2)
);
```

Of course, nothing stops you from adding the row types one by one, in a loop or in some other way, and then calling `initialize()` manually. And yes, of course you can keep a reference to a row set type as soon as it has been constructed, not waiting for initialization. You could do instead:

```
Autoref<RowSetType> rst = new RowSetType();
rst->addRow("name1", rt1);
rst->addRow("name2", rt2);
rst->initialize();
if (rst->getErrors()->hasError()) {
  ...
}
```

You could use the `initializeOrThrow()` template after the piecemeal construction as well, just I also wanted to show the way for the manual handling of the errors. And you can use the `new` or `make()` interchangeably.

All that the initialization does is fixate the row set, forbid the addition of the further row types to it. Which makes sense at the moment but I'm not so sure about the future, in the future the dynamically expandable row sets might come useful. We'll see when we get there.

```
RowSetType();
static RowSetType *make();
```

Construct a row set type. The method `make()` is just a wrapper around the constructor that is more convenient to use with the following `->addRow()`, because of the way the operator priorities work in C++. Like any other type, RowSetType is unnamed by itself, and takes no constructor arguments. Like any other type, RowSetType is an Mtarget and can be shared between multiple threads after it has been initialized.

```
RowSetType *addRow(const string &rname, const_Autoref<RowType>rtype);
```

Add a row type to the set. All the row types are named, and all the names must be unique within the set. The order of the addition matters too. See the further explanation of why it does in the description of the FnReturn in Section 20.32: "FnReturn reference" (p. 495). If this method detects an error (such as duplicate names), it will append the error to the internal Errors object, that can be read later by `getErrors()`. A type with errors must not be used.

The row types may not be added after the row set type has been initialized.

```
void initialize();
```

Initialize the type. Any detected errors can be read afterwards with `getErrors()`. The repeated calls of `initialize()` are ignored.

```
bool isInitialized() const;
```

Check whether the type has been initialized.

```
typedef vector<string> NameVec;
const NameVec &getRowNames() const;
typedef vector<Autoref<RowType> > RowTypeVec;
const RowTypeVec &getRowTypes() const;
```

Read back the contents of the type. The elements will go in the order they were added.

```
int size() const;
```

Read the number of row types in the set.

```
int findName(const string &name) const;
```

Translate the row type name to index (i.e. the order in which it was added, starting from 0). Returns -1 on an invalid name.

```
RowType *getRowType(const string &name) const;
```

Find the type by name. Returns NULL on an invalid name.

```
const string *getRowTypeName(int idx) const;
RowType *getRowType(int idx) const;
```

Read the data by index. These methods check that the index is in the valid range, and otherwise return NULL.

The usual methods inherited from Type also work: `getErrors()`, `equals()`, `match()`, `printTo()`.

The row set types are considered equal if they contain the equal row types with equal names going in the same order. They are considered matching if they contain matching row types going in the same order, with any names. If the match condition seems surprising to you, think of it as "nothing will break if one type is substituted for another at execution time".

```
void addError(const string &msg);
Erref appendErrors();
```

The ways to add extra errors to the type's errors. It's for convenience of the users of this type, the thinking being that since we already have one Errors object, we can as well use it for everything, and also keep all the errors reported in the order of the fields, rather than first all the errors from the type then all the errors from its user. The FnReturn and FnBinding use it.

20.32. FnReturn reference

FnReturn, defined in `sched/FnReturn.h`, is generally constructed similarly to the RowSetType:

```
ret = initializeOrThrow(FnReturn::make(unit, name)
  ->addLabel("lb1", rt1)
  ->addFromLabel("lb2", lbX)
);
```

Or of course piece-meal. As it gets built, it actually builds a RowSetType inside itself.

```
FnReturn(Unit *unit, const string &name);
static FnReturn *make(Unit *unit, const string &name);
```

The constructor and convenience wrapper. The unit will be remembered only as a pointer, not reference, to avoid the reference loops. However this pointer will be used to construct the internal labels. So until the FnReturn is fully initialized, you better make sure that the Unit object has a reference and doesn't get freed. FnReturn is an Starget, and must be used in only one thread.

```
const string &getName() const;
Unit *getUnitPtr() const;
const string &getUnitName() const;
```

Get back the information from the constructor. Just like for the Label class, it reminds you that the Unit is only available as a pointer, not a reference, here. The FnReturn also has a concept of clearing: it has the special labels inside, and once any of these labels gets cleared, the FnReturn is also cleared by setting the unit pointer to NULL and forgetting the FnContext (see more on that below). So after the FnReturn is cleared, `getUnitPtr()` will return NULL. And again similarly to the Label, there is a convenience function to get the unit name for informational printouts. When FnReturn is cleared, it returns the constant string "[fn return cleared]".

```
FnReturn *addFromLabel(const string &lname, Autoref<Label>from, bool front = true);
```

Add a label to the return by chaining it off another label. `lname` is the name within the return. The full name of the label will be "return_name.label_name". The label names within a return must be unique and not empty, or it will be returned as an initialization error. The label type will be copied (actually, referenced) from the `from` label, and the new label will be automatically chained off it. If the argument `front` is `false`, the chaining will be done at the back of the chain, if `true` (default) then at the front. The front chaining is convenient if you want to pass both the original request and the result into the return. Usually you would define the result computation and then define the return. With the chaining at the back, this would lead to the computation chained off the input label first and the return going after it. This would lead to the result coming out before the argument, and special contortions would be needed to avoid it. With chaining at the front, the return will go in the chain before the computation, even if the return was defined last. The labels can be added only until the return is initialized, or it will throw an Exception.

```
FnReturn *addLabel(const string &lname, const_Autoref<RowType>rtype);
```

Add a new independent label to the return. Works very similarly to `addFromLabel()`, only uses the explicit row type and doesn't chain to anything. The label can be later found with `getLabel()` and either chained off something or used to send the rows to it explicitly. The labels can be added only until the return is initialized.

```
class FnContext: public Starget
{
public:
  virtual ~FnContext();
  virtual void onPush(const FnReturn *fret) = 0;
  virtual void onPop(const FnReturn *fret) = 0;
};
FnReturn *setContext(Onceref<FnContext> ctx);
```

Set the context with handlers for the pushing and popping of the bindings in the FnReturn. FnContext is a top-level class, not nested in FnReturn. I.e. **not** Triceps::FnReturn::FnContext but Triceps::FnContext. Triceps generally tries to follow the C++ tradition of using the virtual methods for the callbacks, with the user then subclassing the base class and replacing the callback methods. However subclassing FnReturn is extremely inconvenient, because it gets connected to the other objects in a quite complicated way. So the solution is to make a separate context class for the callbacks, and then connect it. The callbacks will be called just before the binding is pushed or popped, but after the check for the correctness of the push or pop. They can be used to adjust the state of the streaming function by pushing or popping its stack of local variables, like was shown in the Perl examples in Section 15.10: "Streaming functions and more recursion" (p. 271) . The context may be set only until the return is initialized.

```
template<class C> C *contextIn() const;
```

Get back the context. Since the context will be a subclass of FnContext, this also handles the correct type casting. Use it like:

```
Autoref<MyFnCtx> ctx = fret1->contextIn<MyFnCtx>();
```

The type is converted using the `static_cast`, and you need to know the correct type in advance, or your program will break in some horrible ways. If the context has not been set, it will return a NULL.

```
void initialize();
```

Initialize the FnReturn. Very similar to the Type classes, it will collect the errors in an Errors object that has to be checked afterwards, and an FnReturn with errors must not be used. The initialization can be called repeatedly with no ill effects. After initialization the structure of the return (labels and context) can not be changed any more.

```
Erref getErrors() const;
```

Get the errors that were detected during construction and initialization. Normally called after initialization but can also be called at any stage, as the errors are collected all the way through the object construction.

```
bool isInitialized() const;
```

Check whether the return is initialized.

```
RowSetType *getType() const;
```

Get the type of the return, which gets built internally by the return. The names of the row types in the set will be the same as the names of labels in the return, and their order will also be the same. This call can be made only after initialization, or it will throw an Exception.

```
int size() const;
```

Get the number of labels in the return. Can be called at any time.

```
const RowSetType::NameVec &getLabelNames() const;
const RowSetType::RowTypeVec &getRowTypes() const;
const string *getLabelName(int idx) const;
RowType *getRowType(const string &name) const;
RowType *getRowType(int idx) const;
```

Get the piecemeal information about the label names and types. These are really the convenience wrappers around the RowSetType. Note that they return pointers to be able to return NULL on the argument that is out of range. A somewhat special feature is that even though the row set type can be read only after initialization (after it becomes frozen and can not be messed with any more), these wrappers work at any time, even when the return is being built.

```
bool equals(const FnReturn *t) const;
bool match(const FnReturn *t) const;
bool equals(const FnBinding *t) const;
bool match(const FnBinding *t) const;
```

Convenience wrappers that compare the equality or match of the underlying row set types.

```
Label *getLabel(const string &name) const;
int findLabel(const string &name) const;
Label *getLabel(int idx) const;
```

Get the label by name or index, or the index of the label by name. Return a NULL pointer or -1 index on an invalid argument.

```
typedef vector<Autoref<RetLabel> > ReturnVec;
const ReturnVec &getLabels() const;
```

Get the whole set of labels. FnReturn::RetLabel is a special private label type with undisclosed internals. You need to treat these labels as being a plain Label.

```
void push(Onceref<FnBinding> bind);
```

Push a binding on the return stack. The return must be initialized, and the binding must be of a matching type, or an Exception will be thrown. The reference to the binding will be kept in the FnReturn until it's popped.

```
void pushUnchecked(Onceref<FnBinding> bind);
```

Similar to `push()`, only the type of the binding is not checked. This method is not available in Perl, it's an optimization for the automatically generated code that does all the type checks up front at the generation time. The manually written code probably should not be using it.

```
void pop(Onceref<FnBinding> bind);
```

Pop a binding from the return stack. The binding argument specifies, which binding is expected to be popped. It's not strictly necessary (use the `pop()` without arguments to skip it) but allows to catch any mess-ups with the return stack early. If the stack is empty or the top binding is not the same as the argument, throws an Exception.

```
void pop();
```

The unchecked version. It still checks and throws if the stack is empty. This method may come handy occasionally, but in general the checked version should be preferred. Pretty much the only reason to use it would be if you try to restore after a major error and want to pop everything from all your FnReturns untill their stacks become empty. But there is much trouble with this kind of restoration.

```
int bindingStackSize() const;
```

Get the current size of the return stack (AKA the stack of bindings). Useful for debugging.

```
typedef vector<Autoref<FnBinding> > BindingVec;
const BindingVec &bindingStack() const;
```

Get the current return stack. Useful for debugging.

```
bool isFaceted() const;
```

Returns `true` if this FnReturn object is a part of a Facet.

20.33. FnBinding reference

FnBinding is defined in `sched/FnBinding.h`, and substantially matches the Perl version. It inherits from Starget, and can be used in only one thread.

Like many other classes, it has the constructor and the static `make()` function:

```
FnBinding(const string &name, FnReturn *fn);
static FnBinding *make(const string &name, FnReturn *fn);
```

The binding is constructed on a specific FnReturn and obtains (references) the RowSetType from it. The FnReturn must be initialized before it can be used to create the bindings. Later the FnBinding can be pushed onto any matching FnReturn, not just the one it was constructed with.

It's generally constructed in a chain fashion:

```
Autoref<FnBinding> bind = FnBinding::make(fn)
  ->addLabel("lb1", lb1, true)
  ->addLabel("lb2", lb2, false);
```

Each method in the chain returns the same FnBinding object. The method `addLabel()` adds one concrete label that gets connected to the FnReturn's label by name. The other chainable method is `withTray()` which switches the mode of collecting the resulting rowops in a tray rather than calling them immediately.

The errors encountered during the chained construction are remembered and can be read later with the method:

```
Erref getErrors() const;
```

You must check the bindings for errors before using it. A binding with errors may not be used.

Or you can use the `checkOrThrow()` wrapper from `common/Initialize.h` to automatically convert any detected errors to an Exception:

```
Autoref<FnBinding> bind = checkOrThrow(FnBinding::make(fn)
  ->addLabel("lb1", lb1, true)
```

```
  ->addLabel("lb2", lb2, false)
  ->withTray(true)
);
```

Continuing with the details of chainable methods:

```
FnBinding *addLabel(const string &name, Autoref<Label> lb, bool autoclear);
```

Adds a label to the binding. The name must match a name from the FnReturn, and there may be only one label bound to a name (some names from the return may be left unbound, and the rowops coming to them will be ignored). The label must have a type matching the named FnReturn's label. The autoclear flag enables the automatic clearing of the label (and also forgetting it in the Unit) when the binding gets destroyed. This allows to create and destroy the bindings dynamically as needed. So, basically, if you've created a label just for the binding, use `autoclear==true`. If you do a binding to a label that exists in the model by itself and can be used without the binding, use `autoclear==false`.

In principle, nothing stops you from adding more labels later (though you can't remove nor replace the labels that are already added). Just make sure that their types match the expected ones.

The labels in the FnBinding may belong to a different Unit than the FnReturn. This allows to use the FnReturn/FnBinding coupling to connect the units.

```
FnBinding *withTray(bool on);
```

Changes the tray collection mode, the argument `on==true` enables it, `on==false` disables. Can be done at any time, not just at construction. Disabling the tray mode discards the current tray. If the tray mode is enabled, whenever the binding is pushed onto a return and the rowops come into it, the labels in this binding won't be called immediately but they would adopt the incoming rowops, and the result will be queued into a tray, to be executed later.

```
Onceref<Tray> swapTray();
```

Used with the tray collection mode, normally after some rowops have been collected in the tray. Returns the current tray and replaces it in the binding with a new clean tray. You can call the returned tray afterwards. If the tray mode is not enabled, will return NULL, and won't create a new tray.

```
Tray *getTray() const;
```

Get the current tray. You can use and modify the tray contents in any usual way. If the tray mode is not enabled, will return NULL.

```
void callTray();
```

A convenience combination method that swaps the tray and calls it. This method is smart about the labels belonging to different units. Each rowop in the tray is called with its proper unit, that is found from the rowop's label. Mixing the labels of multiple units in one binding is probably still not such a great idea, but it works anyway.

```
const string &getName() const;
```

Get back the binding's name.

```
RowSetType *getType() const;
```

Get the type of the binding. It will be the same row set type object as created in the FnReturn that was used to construct this FnBinding.

```
int size() const;
```

Get the number of labels in the row set type (of all available labels, not just the ones that have been added).

```
const RowSetType::NameVec &getLabelNames() const;
const RowSetType::RowTypeVec &getRowTypes() const;
```

```
const string *getLabelName(int idx) const;
RowType *getRowType(const string &name) const;
RowType *getRowType(int idx) const;
```

The convenience wrappers that translate to the same methods in the RowSetType.

```
Label *getLabel(const string &name) const;
int findLabel(const string &name) const;
Label *getLabel(int idx) const;
```

Methods similar to FnReturn that allow to translate the names to indexes and get the labels by name or index. The same return values, the index -1 is returned for an unknown name, and a NULL label pointer is returned for an unknown name, an incorrect index and an undefined label at a correct name or index.

```
typedef vector<Autoref<Label> > LabelVec;
const LabelVec &getLabels() const;
```

Return all the labels as a vector. This is an internal vector of the class, so only a const reference is returned. The elements for undefined labels will contain NULLs.

```
typedef vector<bool> BoolVec;
const BoolVec &getAutoclear() const;
```

Return the vector of the autoclear flags for the labels.

```
bool isAutoclear(const string &name) const;
```

Get the autoclear flag for a label by name. If the name is unknown, will quietly return `false`.

```
bool equals(const FnReturn *t) const;
bool match(const FnReturn *t) const;
bool equals(const FnBinding *t) const;
bool match(const FnBinding *t) const;
```

Similarly to the FnReturn, the convenience methods that compare the types between the FnReturns and FnBindings. They really translate to the same methods on the types of the returns or bindings.

20.34. ScopeFnBind and AutoFnBind reference

A couple more of helper classes are defined in `sched/FnReturn.h`.

ScopeFnBind does a scoped pushing and popping of a binding on an FnReturn. Its only method is the constructor:

```
ScopeFnBind(Onceref<FnReturn> ret, Onceref<FnBinding> binding);
```

It's used as:

```
{
    ScopeFnBind autobind(ret, binding);
    ...
}
```

It will pop the binding at the end of the block. An unpleasant feature is that if the return stack get messed up, it will throw an Exception from a destructor, which is a big no-no in C++. However since normally in the C++ code the Triceps Exception is essentially an abort, this works good enough. If you make the Exception catchable, such as when calling the C++ code from an interpreter, you better make very sure that the stack can not get corrupted, or do not use ScopeFnBind.

AutoFnBind is a further extension of the scoped binding. It does three additional things:

• It allows to push multiple bindings on multiple returns as a group, popping them all on destruction.

- It's a reference-counted Starget object, which allows the scope to be more than one block.

- It also has a more controllable way of dealing with the exceptions.

This last two properties allow it to be used from the Perl code, making the scope of a Perl block, not C++ block, and to pass the exceptions properly back to Perl.

```
AutoFnBind();
AutoFnBind *make();
```

The constructor just creates an empty object which then gets filled with bindings.

```
AutoFnBind *add(Onceref<FnReturn> ret, Autoref<FnBinding> binding);
```

Add a binding, in a chainable fashion. The simple-minded way of using the AutoFnBind is:

```
{
    Autoref<AutoFnBind> bind = AutoFnBind::make()
        ->add(ret1, binding1)
        ->add(ret2, binding2);
    ...
}
```

However if any of these `add()`s throw an Exception, this will leave an orphaned AutoFnBind object, since the throwing would happen before it had a chance to do the reference-counting. So the safer way to use it is:

```
{
    Autoref<AutoFnBind> bind = new AutoFnBind;
    bind
        ->add(ret1, binding1)
        ->add(ret2, binding2);
    ...
}
```

Then the AutoFnBind will be reference-counted first, and if an `add()` throws later, this will cause a controlled destruction of the Autoref and of AutoFnBind.

But it's not the end of the story yet. The throws on destruction are still a possibility. To catch them, use an explicit clearing before the end of the block:

```
void clear();
```

Pops all the bindings. If any Exceptions get thrown, they can get caught nicely. It tries to be real smart, going through all the bindings in the backwards order and popping each one of them. If a `pop()` throws an exception, its information will be collected but `clear()` will then continue going through the whole list. At the end of the run it will make sure that it doesn't have any references to anything any more, and then will re-throw any collected errors as a single Exception. This cleans up the things as much as possible and as much as can be handled, but the end result will still not be particularly clean: the returns that got their stacks corrupted will still have their stacks corrupted, and some very serious application-level cleaning will be needed to continue. Probably a better choice would be to destroy everything and restart from scratch. But at least it allows to get safely to this point of restarting from scratch.

So, the full correct sequence will be:

```
{
  Autoref<AutoFnBind> bind = new AutoFnBind;
  bind
    ->add(ret1, binding1)
    ->add(ret2, binding2);
  ...
  bind->clear();
}
```

Or if any code in "..." can throw anything, then something like this snippet(whis is not actually tested, so use with caution):

```
{
  Autoref<AutoFnBind> bind = new AutoFnBind;
  bind
    ->add(ret1, binding1)
    ->add(ret2, binding2);
  try {
  ...
  } catch (Triceps::Exception e) {
    try {
      bind->clear();
    } catch (Triceps::Exception ee) {
      e->getErrors()->append("Unbinding errors triggered by the last error:", ee-
>getErrors());
    }
    throw;
  } catch (exception e) {
    bind->clear(); // might use a try/catch around it as well
    throw;
  }
}
```

It tries to be nice if the exception thrown from "..." was a Triceps one, and add nicely any errors from the binding clearing to it.

Finally, a little about how the Perl AutoFnBind translates to the C++ AutoFnBind:

The Perl constructor creates the C++-level object and adds the bindings to it. If any of them throw, it destroys everything nicely and translates the Exception to Perl. Otherwise it saves a reference to the AutoFnBind in a wrapper object that gets returned to Perl.

The Perl destructor then first clears the AutoFnBind and catches if there is any Exception. However there is just no way to return a Perl exception from a Perl destructor, so it simply prints the error on stderr and calls `exit(1)`. If no exception was thrown, the AutoFnBind gets destroyed nicely by removing the last reference.

For the nicer handling, there is a Perl-level method `clear()` that does the clearing and translates the exception to Perl.

20.35. App reference

As usual, I won't be repeating the descriptions from the Perl reference, but just point out the methods and differences of the C++ version. And there are more detailed descriptions directly in the header files. The App class is defined in `cpp/app/App.h`. Naturally, it's an Mtarget.

The static part of the API is:

```
static Onceref<App> make(const string &name);
static Onceref<App> find(const string &name);
static void drop(Onceref<App> app);

typedef map<string, Autoref<App> > Map;
static void listApps(Map &ret);
```

The App constructor is private, and the Apps get constructed with `make()`. `make()` throws an Exception if an App with this name already exists, `find()` throws an exception if an App with this name does not exist, and `drop()` just does nothing if its argument App had already been dropped.

All the operations on the global list of apps are internally synchronized and thread-safe. `listApps()` clears its argument map and fills it with the copy of the current list. Of course, after it returns and releases the mutex, other threads may create

or delete apps, so the returned list (well, map) may immediately become obsolete. But since it all is done with the reference counters, the App objects will continue to exist and be operable as long as the references to them exist.

The App instance API is as follows. It's also all internally synchronized.

```
const string &getName() const;
```

Get the name of the App.

```
Onceref<TrieadOwner> makeTriead(const string &tname, const string &fragname = "");
```

Define a new Triead. This method is called either from the OS thread where the Triead will run or from its parent thread (unlike Perl, in C++ it's perfectly possible to pass the reference to the TrieadOwner from a parent OS thread to the thread that will run it). Either way, the OS thread that will run this Triead ends up with the TrieadOwner object reference, and no other thread must have it. The arguments are the thread name and fragment name, and the empty fragment name means that this thread won't belong to any fragment.

And just to reiterate, this does not create the OS thread. Creating the OS thread is your responsibility (although Triceps provides the helper classes for the Posix threads). This call only creates the Triceps Triead management structures to put it under control of the App.

If a thread with this name has already been defined, or if the thread name is empty, throws an Exception.

```
void declareTriead(const string &tname);
```

Declare a new thread. Declaring a thread more than once, or declaring a thread that has been already defined, is perfectly OK.

```
void defineJoin(const string &tname, Onceref<TrieadJoin> j);
```

This is a call without an analog in the Perl API. This defines a way for the harvester to join the thread. The Perl API happens to be hardcoded to join the Perl threads. But the C++ API can deal with any threads: POSIX ones, Perl ones, whatever. If a thread wants to be properly joined by the harvester, it must define its join interface, done as a TrieadJoin object. Each kind of threads will define its own subclass of TrieadJoin.

If there is no join defined for a Triead, then when it exits, the harvester will just update the state and manage the Triead object properly but won't do any joining. Which is useful in case if the OS thread is detached (not the best idea but doable) or if the Triead is created from the parent OS thread, and then the actual OS thread creation fails and there is nothing to join.

If the thread is not declared nor defined yet, `defineJoin()` throws an exception.

It's possible (though unusual) to call this method multiple times for the same thread. That would just replace the joiner object. The joiner is a reference-counted object, so the old object will just have itse reference count decreased. It's possible to pass the joiner as NULL, that would just drop the existing joiner, if any was defined.

```
typedef map<string, Autoref<Triead> > TrieadMap;
void getTrieads(TrieadMap &ret) const;
```

List the Trieads in this App. Same as with listing the Apps, the argument map gets cleared and then filled with the current contents.

```
void harvester(bool throwAbort = true);
bool harvestOnce();
void waitNeedHarvest();
```

The harvester API is very similar to Perl, with confession replaced by Exception. The only difference is how the argument specifies the throwing of an Exception on detecting the App abort (the Exception will still be thrown only after joining all the App's threads).

App reference 503

The result of `harvestOnce()` is `true` if the App is dead. The Exceptions in `harvestOnce()` originate from the `TrieadJoin::join()` method that performs the actual joining. All the caveats apply in the same way as in Perl.

```
bool isDead();
```

Returns `true` if the App is dead.

```
void waitDead();
```

Wait for the App to become dead.

```
bool isAborted() const;
```

Returns `true` if the App is aborted.

```
string getAbortedBy() const;
string getAbortedMsg() const;
```

Get the thread name and message that caused the abort. If the App is not aborted, will return the empty strings.

```
void abortBy(const string &tname, const string &msg);
```

Abort the app, by the thread tname, with the error message msg.

```
bool isShutdown();
```

Returns true if the App has been requested to shut down.

```
void shutdown();
```

Request the App to shut down. This involves interrupting all the threads in case if they are sleeping. The interruption is another functionality of the TrieadJoin object. It's possible for the TrieadJoin interruptor to encounter an error and throw an Exception. If this happens, `shutdown()` will still go through all the Trieads and interrupt them, and then repackage the error messages from all the received Exceptions into one Exception and re-throw it.

Technically, this means that in the Perl API the shutdown might also confess, when its underlying C++ call returns an Exception. This should theoretically never happen, but practically you never know.

```
void shutdownFragment(const string &fragname);
```

Shut down a fragment. All the logic described in the Perl API applies. Again, this involves interruption of all the threads in the fragment, and if any of them throw Exceptions, these will be re-thrown as a single Exception.

```
enum {
    DEFAULT_TIMEOUT = 30,
};
void setTimeout(int sec, int fragsec = -1);
```

Set the readiness timeouts (main and fragment), in seconds. If the fragment timeout argument is <0, it gets set to the same value as the main timeout.

```
void setDeadline(const timespec &dl);
```

Set the deadline (unlike timeout, with fractional seconds) for the main readiness.

```
void refreshDeadline();
```

Explicitly refresh the deadline, using the fragment timeout.

```
void requestDrain();
```

Request a shared drain.

```
void requestDrainExclusive(TrieadOwner *to);
```

Request an exclusive drain, with the argument TrieadOwner. Unlike Perl API, the C++ API supports the methods for requesting the exclusive drain on both App and TrieadOwner classes (the TrieadOwner method is really a wrapper for the App method). In general, using the TrieadOwner method for this purpose probably looks nicer.

Since the TrieadOwner reference is really private to the OS thread that runs it, this method can be called only from that OS thread. Of course, in C++ you could pass it around to the other threads, but don't, TrieadOwner is not thread-safe internally and any operations on it must be done from one thread only.

```
void waitDrain();
```

Wait for the drain (either shared or exclusive) to complete.

```
void drain();
```

A combination of `requestDrain()` and `waitDrain()`.

```
void drainExclusive(TrieadOwner *to);
```

A combination of `requestDrainExclusive()` and `waitDrain()`.

```
bool isDrained();
```

Quickly check if the App is currently drained (should be used only if the App is known to be requested to drain).

```
void undrain();
```

End the drain sequence.

The file descriptor store/load API is really of not much use in C++ as such, in C++ it's easy to pass and share the file descriptors and file objects between the threads as-is. It has been built into the App class for the benefit of Perl and possibly other interpreted languages.

```
void storeFd(const string &name, int fd);
void storeFd(const string &name, int fd, const string &className);
```

Store a file descriptor. Unlike the Perl API, the file descriptor is **not** dupped before storing. It's stored as is, and if you want dupping, you have to do it yourself. The class name argument is optional, if not specified, it's set to an empty string. Throws an Exception if a file descriptor with this name is already stored.

```
int loadFd(const string &name, string *className = NULL) const;
```

Load back the file descriptor. Again, no dupping, returns the stored value as-is. If the name is unknown, returns -1. If the class name argument is not NULL, it will be filled with the stored class name.

```
bool forgetFd(const string &name);
```

Forget the file descriptor. Returns true if this name was known and became forgotten, or false if it wasn't known. Normally, you would load the descriptor, take over its ownership, and then tell the App to forget it.

```
bool closeFd(const string &name);
```

Close the file descriptor (if it was known) and then forget it. Returns true if this name was known and became forgotten, or false if it wasn't known.

The rest of the Perl App methods have no analogs in C++. They are just purely Perl convenience wrappers.

20.36. Triead reference

Triead is a class that can be referenced from multiple threads and inherits from Mtarget. It's defined in `app/Triead.h`.

The meaning of the C++ methods is exactly the same as in Perl, only the format of values is slightly different. Obviously, the `start*()` methods are Perl-only, in C++ the Trieads are defined by `App::makeTriead()` with the help of `App::defineJoin()`.

```
const string &getName() const;
```

Get the Triead's name.

```
const string &fragment() const;
```

Get the name of the Triead's fragment. If the Triead doesn't belong to a fragment, returns an empty string "".

```
bool isConstructed() const;
```

Check whether the Triead has been constructed. For the explanation of the Triead lifecycle states, see Section 16.2: "The Triead lifecycle" (p. 288) .

```
bool isReady() const;
```

Check whether the Triead is ready.

```
bool isDead() const;
```

Check whether the Triead is dead.

```
bool isInputOnly() const;
```

Check whether the Triead is input-only, that is, it has no reader nexuses imported into it. When the Triead is created, this flag starts its life as `false`, and then its correct value is computed when the Triead becomes ready. So, to check this flag correctly, you must first check that the Triead is ready.

```
typedef map<string, Autoref<Nexus> > NexusMap;
void exports(NexusMap &ret) const;
void imports(NexusMap &ret) const;
void readerImports(NexusMap &ret) const;
void writerImports(NexusMap &ret) const;
```

Get the list of nexuses exported from this Triead, or imported, or only the reader imports, or only the writer imports. In the result of `imports()` there is no way to tell, which nexuses are imported for reading and which for writing, use the specialized methods for that. In all these methods the argument map gets cleared and then filled with the new returned contents.

```
Onceref<Nexus> findNexus(const string &srcName,
  const string &appName, const string &name) const;
```

This is a method unique to the C++ API. It looks up an exported nexus by name without involving the overhead of getting the whole map. Here `name` is the name of the nexus to look up (its short part, without the thread's name in it). If there is no such nexus, an Exception will be thrown.

The `srcName` and `appName` are used for the error message in the Exception: `srcName` is the name of the thread that requested the look-up, and `appName` is the name of the App where the threads belong. (It might seem surprising, but a Triead object has no reference to its App, and doesn't know the App's name. It has to do with the avoidance of the circular references).

20.37. TrieadOwner reference

The TrieadOwner is defined in `app/TrieadOwner.h`. Its constructor is protected, the normal way of constructing is by calling `App::makeTriead()`. This is also called "defining a Triead".

TrieadOwner is an Starget, and must be accessed only from the thread that owns it (though it's possible to create it in the parent thread and then pass to the actual owner thread, as long as the synchronization between the threads is done properly).

```
Triead *get() const;
```

Get the public side if the Triead. In C++, unlike Perl, the Triead methods are not duplicated in TrieadOwner. So they are accessed through `get()`, and for example to get the Triead name, call `to->get()->getName()`. The TrieadOwner holds a reference to Triead, so the Triead object will never get destroyed as long as the TrieadOwner is alive.

```
App *app() const;
```

Get the App where this Triead belongs. The TrieadOwner holds a reference to App, so the App object will never get destroyed as long as the TrieadOwner is alive.

```
bool isRqDead() const;
```

Check whether this triead has been requested to die.

```
void requestMyselfDead();
```

Request this thread itself to die (see the extended description in the Perl reference).

```
Unit *unit() const;
```

Get the main unit of this Triead. It has the same name as the Triead itself.

```
void addUnit(Autoref<Unit> u);
```

Keep track of an additional unit. Adding a unit multiple times has no effect. See the other implications in the Perl reference.

```
bool forgetUnit(Unit *u);
```

Forget about an additional unit. If the unit is already unknown, has no effect. The main unit can not be forgotten.

```
typedef list<Autoref<Unit> > UnitList;
const UnitList &listUnits() const;
```

List the tracked units. The main unit is always included at the front of the list.

```
void markConstructed();
```

Advance the Triead state to Constructed. Repeated calls have no effect.

```
void markReady();
```

Advance the Triead state to Ready. Repeated calls have no effect. The advancement is cumulative: if the Triead was not constructed yet, it will be automatically advanced first to Constructed and then to Ready. If this is the last Triead to become ready, it will trigger the App topology check, and if the check fails, abort the App and throw an Exception.

```
void readyReady();
```

Advance the Triead to the Ready state, and wait for all the Trieads to become ready. The topology check applies in the same way as in `markReady()`.

```
void markDead();
```

Advance the Triead to the Dead state. If this Triead was not Ready yet and it's the last one to become so, the topology check will run. If the topology check fails, the App will be aborted but `markDead()` will not throw an exception.

This method is automatically called from the TrieadOwner destructor, so most of the time there is no need to call it explicitly.

It also clears all the tracked units.

```
void abort(const string &msg) const;
```

Abort the App. The name of this thread will be forwarded to the App along with the error message. The error message may be multi-line.

```
Onceref<Triead> findTriead(const string &tname, bool immed = false);
```

Find a Triead in the App by name. The flag `immed` controls whether this method may wait. If `immed` is false, and the target thread is not constructed yet (but at least declared), the method will sleep until it becomes constructed, and then returns it. If the target thread is not declared, it will throw an Exception. If `immed` is true, the look-up is immediate: it will return the thread even if it's not constructed but is at least defined. If it's not defined (even if it's declared), an Exception will be thrown. The look-up of this Triead itself is always immediate, irrespective of the `immed` flag, to avoid deadlocking itself.

An Exception may also be thrown if a circular sequence of Trieads deadlocks waiting for each other.

```
Onceref<Facet> exportNexus(Autoref<Facet> facet, bool import = true);
```

Export a Nexus. The Nexus definition is constructed as a Facet object, which is then used by this method to construct and export the Nexus. The same argument Facet reference is then returned back as the result. The `import` flag tells, whether the Nexus is to be also imported back by connecting the same original Facet object to it. If import is false, the original Facet reference is still returned back but it can't be used for anything, and can only be thrown away. The direction of the import (reading or writing) is defined in the Facet, as well as the nexus name and all the other information.

Throws an Exception on any errors, in particular on the duplicate facet names within the Triead.

```
Onceref<Facet> exportNexusNoImport(Autoref<Facet> facet);
```

A convenience wrapper around exportNexus() with `import=false`.

```
Onceref<Facet> importNexus(const string &tname,
  const string &nexname, const string &asname,
  bool writer, bool immed = false);
```

Import a Nexus from another Triead. `tname` is the name of the other thread, `nexname` is the name of nexus exported from it, `asname` is the local name for the imported facet ("" means "same as `nexname`"), the `writer` flag determines if the import is for writing, and the `immed` flag has the same meaning as in `findTriead()`. The import of a nexus involves finding its exporting thread, and the `immed` flag controls, how this finding is done.

Throws an Exception if anything is not found, or the local import name conflicts with another imported facet.

```
Onceref<Facet> importNexusImmed(const string &tname,
  const string &nexname, const string &asname, bool writer);
Onceref<Facet> importReader(const string &tname,
  const string &nexname, const string &asname = "", bool immed=false);
Onceref<Facet> importWriter(const string &tname,
  const string &nexname, const string &asname = "", bool immed=false);
Onceref<Facet> importReaderImmed(const string &tname,
  const string &nexname, const string &asname = "");
Onceref<Facet> importWriterImmed(const string &tname,
  const string &nexname, const string &asname = "");
```

Convenience wrappers for `importNexus()`, providing the default arguments and the more mnemonic names.

```
typedef map<string, Autoref<Nexus> > NexusMap;
void exports(NexusMap &ret) const;
```

Get the nexuses exported here. The map argument will be cleared and refilled with the new values.

```
typedef map<string, Autoref<Facet> > FacetMap;
void imports(FacetMap &ret) const;
```

Get the facets imported here. The map argument will be cleared and refilled with the new values.

```
NexusMaker *makeNexusReader(const string &name);
```

```
NexusMaker *makeNexusWriter(const string &name);
NexusMaker *makeNexusNoImport(const string &name);
```

A convenient way to build the nexuses for export in a chained fashion. The name argument is the nexus name. The Nexus-Maker is an opaque class that has the same building methods as a Facet, plus the method `complete()` that finishes the export. This call sequence is more convenient than building a Facet and then exporting it. For example:

```
Autoref<Facet> myfacet = ow->makeNexusReader("my")
  ->addLabel("one", rt1)
  ->addFromLabel("two", lb2)
  ->setContext(new MyFnContext)
  ->setReverse()
  ->complete();
```

Only one nexus may be built like this at a time, since there is only one instance of NexusMaker per TrieadOwner that gets reused over and over. It keeps the Facet instance being built in it. If you don't complete the build, that Facet instance will be left sitting around until another `makeNexus*()` call, when it will get thrown away. But in general if you do the calling in the sequence as shown, you can't forget to call `complete()` at the end, since otherwise the return type would not match and the compiler will fail.

```
bool flushWriters();
```

Flush all the writer facets. Returns true on the successfull completion, false if the thread was requested to die, and thus all the output was thrown away.

```
bool nextXtray(bool wait = true, const struct timespec &abstime =
  *(const struct timespec *)NULL);
```

Read and process the next Xtray. Automatically calls the `flushWriters()` after processing. The rest works in the same way as described in Perl, however this is a combination of Perl's `nextXtray()`, `nextXtrayNoWait()` and `nextXtrayTimeLimit()` in one method. If `wait` is false, this method will never wait. If `wait` is true and the `abstime` reference is not NULL, it might wait but not past that absolute time. Otherwise it will wait until the data becomes available or the thread is requested to die.

Returns true if an Xtray has been processed, false if it wasn't (for any reason, a timeout expiring or thread being requested to die).

```
bool nextXtrayNoWait();
```

A convenience wrapper over `nextXtray(false)`.

```
bool nextXtrayTimeout(int64_t sec, int32_t nsec);
```

Another convenience wrapper over `nextXtray()`: read and process an Xtray, with a timeout limit. The timeout value consists of the seconds and nanoseconds parts.

```
void mainLoop();
```

Run the main loop, calling `nextXtray()` repeatedly until the thread is requested to die.

```
bool isRqDrain();
```

Check if a drain is currently requested by any thread (and applies to this thread). In case if an exclusive drain is requested with the exclusion of this thread, this method will return false.

```
void requestDrainShared();
void requestDrainExclusive();
void waitDrain();
bool isDrained();
void drainShared();
void drainExclusive();
void undrain();
```

The drain control, same as in Perl. These methods are really wrappers over the corresponding App methods. And generally a better idea is to do the scoped drains with AutoDrain rather than to call these methods directly.

The C++ API provides no methods for the file descriptor tracking as such. Instead these methods are implemented in the class FileInterrupt. TrieadOwner has a public field

```
Autoref<FileInterrupt> fileInterrupt_;
```

to keep an instance of the interruptor. TrieadOwner itself has no use for it, nor does any other part of Triceps, it's just a location to keep this reference for the convenience of the application developer. The Perl API makes use of this location.

But how does the thread then get interrupted when it's requested to die? The answer is that the TrieadJoin object also has a reference to the FileInterrupt object, and even creates that object in its own constructor. So when a joiner method is defined for a thread, that supplies the App with the access to its interruptor as well. And to put the file descriptors into the FileInterrupt object, you can either keep a direct reference to it somewhere in your code, or copy that reference into the TrieadOwner object.

Here is for example how the Perl thread joiner is created for the TrieadOwner inside the Perl implementation:

```
string tn(tname);
Autoref<TrieadOwner> to = appv->makeTriead(tn, fragname);
PerlTrieadJoin *tj = new PerlTrieadJoin(appv->getName(), tname,
  SvIOK(tid)? SvIV(tid): -1,
  SvIOK(handle)? SvIV(handle): 0,
  testfail);
to->fileInterrupt_ = tj->fileInterrupt();
appv->defineJoin(tn, tj);
```

This is somewhat cumbersome, but the native C++ programs can create their threads using the class BasicPthread that takes care of this and more.

20.38. Nexus reference

The Nexus object is an Mtarget, and safe to access from multiple threads. It's defined in `app/Nexus.h`. The Nexus class is pretty much opaque. It's created and managed entirely inside the App infrastructure from a Facet, and even the public API for importing a nexus doesn't deal with the Nexus object itself, but only with its name. The only public use of the Nexus object is for the introspection and entertainment value, to see what Trieads export and import what Nexuses: pretty much the only way to get a Nexus reference is by listing the exports or imports of a Triead.

The API of a Nexus is very limited:

```
const string &getName() const;
```

Get the name of the nexus (the short name, inside the Triead).

```
const string &getTrieadName() const;
```

Get the name of the Triead that exported this nexus.

```
bool isReverse() const;
```

Check whether the nexus is reverse.

```
int queueLimit() const;
```

Get the queue limit of the nexus.

20.39. Facet reference

The general functioning of a facet is the same in C++ as in Perl, so please refer to the Perl reference for this information.

However the construction of a Facet is different in C++. And the export is different: first you construct a Facet from scratch and then give it to `TrieadOwner::exportNexus()` to create a Nexus from it. The import is still the same: you call `TrieadOwner::importNexus()` method or one of its varieties and it returns a Facet.

In the C++ API the Facet has a notion of being imported, very much like the FnReturn has a notion of being initialized. When a Facet is first constructed, it's not imported. Then `exportNexus()` creates the nexus from the facet, exports it into the App, and also imports the nexus information back into the facet, marking the facet as imported. It returns back a reference to the exact same Facet object, only now that object becomes imported. Obviously, you can use either the original or returned reference, they point to the same object. Once a Facet has been imported, it cannot be modified any more. The Facet object returned by the `importNexus()` is also marked as imported, so it cannot be modified either. And also an imported facet cannot be exported again.

However `exportNexus()` has an exception. If the export is done with the argument `import` set to false, the facet object will be left unchanged and not marked as imported. A facet that is not marked as imported cannot be used to send or receive data. Theoretically, it can be used to export another nexus, but practically this would not work because that would be an attempt to export another nexus with the same name from the same thread. In reality such a Facet can only be thrown away, and there is not much use for it. You can read its components and use them to construct another Facet but that's about it.

It might be more convenient to use the `TrieadOwner::makeNexus*()` methods to build a Facet object rather than building it directly. In either case, the methods are the same and accept the same arguments, just the Facet methods return a Facet pointer while the NexusMaker methods return a NexusMaker pointer.

The Facet class is defined in `app/Facet.h`. It inherits from Mtarget for an obscure reason that has to do with App topology analysis but it's intended to be used from one thread only.

You don't have to keep your own references to all your Facets. The TrieadOwner will keep a reference to all the imported Facets, and they will not be destroyed while the TrieadOwner exists (and this applies to Perl as well).

```
enum {
  DEFAULT_QUEUE_LIMIT = 500,
};
```

The default value used for the nexus queue limit (as a count of Xtrays, not rowops). Since the reading from the nexus involves double-buffering, the real queue size might grow up to twice that amount.

```
static string buildFullName(const string &tname, const string &nxname);
```

Build the full nexus name from its components.

```
Facet(Onceref<FnReturn> fret, bool writer);
```

Create a Facet, initially non-imported (and non-exported). The FnReturn object `fret` defines the set of labels in the facet (and nexus), and the name of FnReturn also becomes the name of the Facet, and of the Nexus. The `writer` flag determines whether this facet will become a writer (if true) or a reader (if false) when a nexus is created from it. If the nexus gets created without importing the facet back, the writer flag doesn't matter and can be set either way.

The FnReturn should generally be not initialized yet. The Facet constructor will check if FnReturn already has the labels _BEGIN_ and _END_ defined, and if either is missing, will add it to the FnReturn, then initialize it. If both _BEGIN_ and _END_ are already present, then the FnReturn can be used even if it's already initialized. But if any of them is missing, FnReturn must be not initialized yet, otherwise the Facet constructor will fail to add these labels.

The same FnReturn object may be used to create only one Facet object. And no, you cannot import a Facet, get an FnReturn from it, then use it to create another Facet.

If anything goes wrong, the constructor will not throw but will remember the error, and later the `exportNexus()` will find it and throw an Exception from it.

```
static Facet *make(Onceref<FnReturn> fret, bool writer);
```

Same as the constructor, used for the more convenient operator priority for the chained calls.

```
static Facet *makeReader(Onceref<FnReturn> fret);
static Facet *makeWriter(Onceref<FnReturn> fret);
```

Syntactic sugar around the constructor, hardcoding the writer flag.

Normally the facets are constructed and exported with the chained calls, like:

```
Autoref<Facet> myfacet = to->exportNexus(
  Facet::makeWriter(FnReturn::make("My")->...)
  ->setReverse()
  ->exportTableType(Table::make(...)->...)
);
```

Or with a similar chain of calls starting with `TrieadOwner::makeNexus()`.

Because of this, the methods that are used for post-construction return the pointer to the original Facet object. They also almost never throw the Exceptions, to prevent the memory leaks through the orphaned Facet objects. The only way an Exception might get thrown is on an attempt to use these methods on an already imported Facet. Any other errors get collected, and eventually `exportNexus()` will find them and properly throw an Exception, making sure that the Facet object gets properly disposed of.

```
Facet *exportRowType(const string &name, Onceref<RowType> rtype);
```

Add a row type to the Facet. May throw an Exception if the the facet is already imported. On other errors remembers them to be thrown on an export attempt.

```
Facet *exportTableType(const string &name, Autoref<TableType> tt);
```

Add a table type to the Facet. May throw an Exception if the the facet is already imported. On other errors remembers them to be thrown on an export attempt. The table type must also be deep-copyable and contain no errors. If the deep copy cannot proceed (say, a table type involves a Perl sort condition with a direct reference to the compiled Perl code) the `deepCopy()` method must still return a newly created object but remember the error inside it. Later when the table type is initialized, that object's initialization must return this error. The `exportTableType()` does a deep copy then initializes the copied table type. If this detects any errors, they get remembered and cause an Exception later in `exportNexus()`.

```
Facet *setReverse(bool on = true);
```

Set (or clear) the nexus reverse flag. May throw an Exception if the the facet is already imported.

```
Facet *setQueueLimit(int limit);
```

Set the nexus queue limit. May throw an Exception if the the facet is already imported.

```
Erref getErrors() const;
```

Get the collected errors, so that they can be found without an export attempt.

```
bool isImported() const;
```

Check whether this facet is imported.

The rest of the methods are the same as in Perl. They can be used even if the facet is not imported.

```
bool isWriter() const;
```

Check whether this is a writer facet (or if returns false, a reader facet).

```
bool isReverse() const;
```

Check whether the underlying nexus is reverse.

```
int queueLimit() const;
```

Get the queue size limit of the nexus. Until the facet is exported, this will always return the last value set by `setQueue-Limit()`. However if the nexus is reverse, on import the value will be changed to a very large integer value, currently `INT32_MAX`, and on all the following calls this value will be returned. Technically speaking, the queue size of the reverse nexuses is not unlimited, it's just very large, but in practice it amounts to the same thing.

```
FnReturn *getFnReturn() const;
```

Get the FnReturn object. If you plan to destroy the Facet object soon after this method is called, make sure that you put the FnReturn pointer into an Autoref first.

```
const string &getShortName() const;
```

Get the short name, AKA "as-name", which is the same as the FnReturn's name. Do not destroy the facet while using the returned reference.

```
const string &getFullName() const;
```

Get the full name of the nexus imported through this facet. If the facet is not imported, will return an empty string. Do not destroy the facet while using the returned reference.

```
typedef map<string, Autoref<RowType> > RowTypeMap;
const RowTypeMap &rowTypes() const;
```

Get the map of the defined row types. Returns the reference to the Facet's internal map object.

```
typedef map<string, Autoref<TableType> > TableTypeMap;
const TableTypeMap &tableTypes() const;
```

Get the map of defined table types. Returns the reference to the Facet's internal map object.

```
RowType *impRowType(const string &name) const;
```

Find a single row type by name. If the name is not known, returns NULL.

```
TableType *impTableType(const string &name) const;
```

Find a single table type by name. If the name is not known, returns NULL.

```
Nexus *nexus() const;
```

Get the nexus of this facet. If the facet is not imported, returns NULL.

```
int beginIdx() const;
int endIdx() const;
```

Return the indexes (as in "integer offset") of the _BEGIN_ and _END_ labels in FnReturn.

```
bool flushWriter();
```

Flush the collected rowops into the nexus as a single Xtray. If there is no data collected, does nothing. Returns true on a successful flush (even if there was no data collected), false if the Triead was requested to die and thus all the data gets thrown away.

20.40. AutoDrain reference

The scoped drain in C++ has more structure than in Perl. It consists of the base class AutoDrain and two subclasses: AutoDrainShared and AutoDrainExclusive. They are all defined in `app/AutoDrain.h`. They are all Stargets, and can be used in only one thread.

The base AutoDrain cannot be created directly but is convenient for keeping a reference to any kind of drain:

```
Autoref<AutoDrain> drain = new AutoDrainShared(app);
```

The constructor of the subclass determines whether the drain is shared or exclusive, and the rest of the methods are defined in the base class.

It's also possible to use the a direct local variable:

```
{
  AutoDrainShared drain(app);
  ...
}
```

Just remember not to mix the metaphors, if you create a local variable, don't try to create the references to it.

The constructors are:

```
AutoDrainShared(App *app, bool wait = true);
AutoDrainShared(TrieadOwner *to, bool wait = true);
```

Create a shared drain from an App or TrieadOwner. The `wait` flag determines if the constructor will wait for the drain to complete, otherwise it will return immediately. Well, usually immediately, if there is no other incompatible drain active at the moment. The shared drain requests the draining of all the Trieads, and multiple threads may have their shared drains active at the same time (the release will happen when all these drains become released). A shared drain will wait for all the preceding exclusive drains to be released before it gets created.

```
AutoDrainExclusive(TrieadOwner *to, bool wait = true);
```

Create an exclusive drain from a TrieadOwner. The `wait` flag makes the constructor wait for the completion of the drain. Only one exclusive drain at a time may be active, from only one thread. An exclusive drain will wait for all the preceding shared and exclusive drains to be released before it gets created.

The common method is:

```
void wait();
```

Wait for the drain to complete. Can be called repeatedly. If more data has been injected into the model through the excluded Triead, will wait for that data to drain.

20.41. Sigusr2 reference

When a thread is requested to die, its registered file descriptors become revoked, and the signal SIGUSR2 is sent to it to interrupt any ongoing system calls. For this to work correctly, there must be a signal handler defined on SIGUSR2, because otherwise the default reaction to it is to kill the process. It doesn't matter what signal handler, just some handler must be there. The Triceps library defines an empty signal handler but you can also define your own instead.

In Perl, the empty handler for SIGUSR2 is set when the module Triceps.pm is loaded. You can change it afterwards.

In C++ Triceps provides a class Sigusr2, defined in app/Sigusr2.h, to help with this. All the methods of Sigusr2 are static. If you use the class BasicPthread, you don't need to deal with Sigusr2 directly: BasicPthread takes care of it.

```
static void setup();
```

Set up an empty handler for SIGUSR2 if it hasn't been done yet. This class has a static flag (synchronized by a mutex) showing that the handler had been set up. On the first call it sets the handler and sets the flag. On the subsequent calls it checks the flag and does nothing.

```
static void markDone();
```

Just set the flag that the setup has been done. This allows to set your own handler instead and still cooperate with the logic of Sigusr2 and BasicPthread.

If you set your custom handler before any threads have been started, then set up your handler and then call `markDone()`, telling Sigusr2 that there is no need to set the handler any more.

If you set your custom handler when the Triceps threads are already running (not the best idea but still a possibility), there is a possibility of a race with another thread calling `setup()`. To work around that race, set up your handler, call `markDone()`, then set up your handler again.

```
static void reSetup();
```

This allows to replace a custom handler with the empty one. It always forcibly sets the empty handler (and also the flag).

20.42. TrieadJoin reference

TrieadJoin is the abstract base class that tells the harvester how to join a thread after it had finished. Obviously, it's present only in the C++ API and not in Perl. It's defined in `app/TrieadJoin.h`.

Currently TrieadJoin has two subclasses: PerlTrieadJoin for the Perl threads and BasicPthread for the POSIX threads in C++. I won't be describing PerlTriedJoin, since it's in the internals of the Perl implementation, never intended to be directly used by the application developers, and if you're interested, you can always look at its source code. BasicPthread is described in Section 20.44: "BasicPthread reference" (p. 516) .

Well, actually there is not a whole lot of direct use for TrieadJoin either: you need to worry about it only if you want to define a joiner for some other kind of threads, and this is not very likely.

So, if you want to define a joiner for some other kind of threads, you define a subclass of it, with an appropriately defined method `join()`.

TrieadJoin is an Mtarget, naturally referenced from multiple threads (at the very least it's created in the thread to be joined or its parent, and then passed to the harvester thread by calling `App::defineJoin()`). The methods of TrieadJoin are:

```
TrieadJoin(const string &name);
```

The constructor. The `name` is the name of the Tread, used for the error messages. Due to the various syncronization reasons, this makes the life of the harvester much easier, than trying to look up the name from the Tread object.

```
virtual void join() = 0;
```

The Most Important joining method to be defined by the subclass. The subclass object must also hold the identity of the thread in it, to know which thread to join. The harvester will call this method.

```
virtual void interrupt();
```

The method that interrupts the target thread when it's requested to die. It's called in the context of the thread that triggers the App shutdown (or otherwise requests the target thread to die). By default the TrieadJoin carries a FileInterrupt object in it (it gets created on TrieadJoin construction, and then TrieadJoin keeps a reference to it), that will get called by this method to revoke the files. But everything else is a part of the threading system, and the base class doesn't know how to do it, the subclasses must define their own methods, wrapping the base class.

Both PerlTrieadJoin and BasicPthread add sending the signal `SIGUSR2` to the target thread. For that they use the same target thread identity kept in the object as used by the `join()` call.

```
FileInterrupt *fileInterrupt() const;
```

Get a pointer to the FileInterrupt object defined in the TrieadJoin. The most typical use is to pass it to the TrieadOwner object, so that it can be easily found later:

```
to->fileInterrupt_ = fileInterrupt();
```

Though of course it could be kept in a separate local Autoref instead.

```
const string &getName() const;
```

Get back the name of the joiner's thread.

20.43. FileInterrupt reference

FileInterrupt is the class that keeps track of a bunch of file descriptors and revokes them on demand, hopefully interrupting any ongoing operations on them (and if that doesn't do the job, a separately sent signal will). It's not visible in Perl, being intergrated into the TrieadOwner methods, but in C++ it's a separate class. It's defined in `app/FileInterrupt.h`, and is an Mtarget, since the descriptors are registered and revoked from different threads.

```
FileInterrupt();
```

The constructor, absolutely plain. Normally you would not want to construct it directly but use the object already constructed in TrieadJoin. The object keeps the state, whether the interruption had happened, and is obviously initialized to the non-interrupted state.

```
void trackFd(int fd);
```

Add a file descriptor to the tracked interruptable set. If the interruption was already done, the descriptor will instead be revoked right away by dupping over from `/dev/null`. If the attempt to open `/dev/null` fails, it will throw an Exception.

```
void forgetFd(int fd);
```

Remove a file descriptor to the tracked interruptable set. You must do it before closing the descriptor, or a race leading to the corruption of random file descriptors may occur. If this file descriptor was not registered, the call will be silently ignored.

```
void interrupt();
```

Perform the revocation of all the registered file descriptors by dupping over them from `/dev/null`. If the attempt to open `/dev/null` fails, it will throw an Exception.

This marks the FileInterrupt object state as interrupted, and any following `trackFd()` calls will lead to the immediate revocation of the file descriptors in them, thus preventing any race conditions.

```
bool isInterrupted() const;
```

Check whether this object has been interrupted.

20.44. BasicPthread reference

Building a new Triead is a serious business, containing many moving part. Doing it every time from scratch would be hugely annoying and error prone. The class BasicPthread, defined in `app/BasicPthread.h`, takes care of wrapping all that complicated logic.

It originated as a subclass of pw::pwthread, and even though it ended up easier to copy and modify the code (okay, maybe this means that pwthread can be made more flexible), the usage is still very similar to it. You define a new subclass of BasicPthread, and define the virtual function `execute()` in it. Then you instantiate the object and call the method `start()` with the App argument.

For a very simple example:

```
class MainLoopPthread : public BasicPthread
```

```
{
public:
  MainLoopPthread(const string &name):
    BasicPthread(name)
  { }

  // overrides BasicPthread::execute
  virtual void execute(TrieadOwner *to)
  {
    to->readyReady();
    to->mainLoop();
  }
};

...

Autoref<MainLoopPthread> pt3 = new MainLoopPthread("t3");
pt3->start(myapp);
```

It will properly create the Triead, TrieadOwner, register the thread joiner and start the execution. The TrieadOwner will pass through to the `execute()` method, and its field `fi_` will contain the reference to the FileInterrupt object. After `execute()` returns, it will take care of marking the thread as dead.

It also wraps the call of `execute()` into a try/catch block, so any Exceptions thrown will be caught and cause the App to abort. In short, it's very similar to the Triead management in Perl.

You don't need to keep the reference to the thread object afterwards, you can even do the construction and start in one go:

```
(new MainLoopPthread("t3"))->start(myapp);
```

The internals of BasicPthread will make sure that the object will be dereferenced (and thus, in the absence of other references, destroyed) after the thread gets joined by the harvester.

Of course, if you need to pass more arguments to the thread, you can define them as fields in your subclass, set them in the constructor (or by other means between constructing the object and calling `start()`), and then `execute()` can access them. Remember, `execute()` is a method, so it receives not only the TrieadObject as an argument but also the BasicPthread object as `this`.

BasicPthread is implemented as a subclass of TrieadJoin, and thus is an Mtarget. It provides the concrete implementation of the joiner's virtual methods, `join()` and `interrupt()`. `Interrupt()` calls the method of the base class, then sends the signal `SIGUSR2` to the target thread.

And finally the actual reference:

```
BasicPthread(const string &name);
```

Constructor. The name of the thread is passed through to `App::makeTriead()`. The Triead will be constructed in `start()`, the BasicPthread constructor just collects together the arguments.

```
void start(Autoref<App> app);
```

Construct the Triead, create the POSIX thread, and start the execution there.

```
void start(Autoref<TrieadOwner> to);
```

Similar to the other version of `start()` but uses a pre-constructed TrieadOwner object. This version is useful mostly for the tests, and should not be used much in the real life.

```
virtual void execute(TrieadOwner *to);
```

Method that must be redefined by the subclass, containing the threads's logic.

```
virtual void join();
virtual void interrupt();
```

Methods inherited from TrieadJoin, providing the proper implementations for the POSIX threads.

Chapter 21. Release Notes

21.1. Release 2.0.0

Major:

- Documentation for the C++ API.

- Streaming functions.

- Multithreading.

Minor:

- No more copy trays in the tables, they've got replaced by treating the table as a streaming function (the automatically generated FnReturn in a Table).

- 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.

- In Table added `clear()`, sticky error methods, fixed the handling of errors in the index comparators to produce the sticky errors.

- The Unit ignores the attempts to remember, forget or clear labels while it's already clearing labels.

- In Unit added `isFrameEmpty`, `isInOuterFrame()`.

- In Rowop added an optional argument to `printP()`.

- 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 the C++ API AggregatorGadget added the `typeAs()` template, the version of `sendDelayed()` that constructs the row from the fields, and `getIndexType()`.

- In C++ added a Rhref constructor directly from a FdataVec argument.

- In C++ added the method `Label::adopt()`, making it easier to remember.

- The `Label::adopt()` in Perl now allows the cross-unit adoption.

- Out-of-the-box compilation with a wider range of GCC compiler versions.

- The default constructor of Fdata sets the value to NULL.

- In C++ addded the convenience `get*` methods on Rowref.

- In C++ addded the `initializeOrThrow()`, `checkOrThrow()` templates.

- The tables no longer have an enqueuing mode associated with them.

- The potentially reusable examples have been exported in the packages under Triceps::X.

- The `Triceps::X::SimpleServer::outBuf()` now checks for the client disconnection before sending data to it.

- The "dump" label added to the tables.

- The `chainFront()` method added in the Label in the Perl API and an extra argument with the same meaning added to `chain` in the C++ API.

- The `*Safe()` methods added to the Perl API.

- Finiding an index by its keys with `TableType::findIndexPathForKeys()`.

- Passing through of arbitrary options, and manipulation of the option lists.

- A fast way to check whether a row is empty.

21.2. 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.

21.3. 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.

21.4. 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

A

abort, 435
AggOp, 477, 479
aggregation, 92, 98, 141, 219, 299
   additive, 157
   arguments, 161
   context, 149
   count, 150
   first, 149
   floating point error, 160, 163
   helper table, 145, 150
   initialization, 156
   iteration, 149
   last, 149
   manual, 93, 143, 225
   multiple indexes, 163
   of DELETEs, 146
   opcode, 150
   operation, 159, 479
   optimization, 151, 155, 157
   state, 156, 159
Aggregator, 477
   handler, 456, 477, 478
AggregatorContext, 149, 375, 477
AggregatorGadget, 475
AggregatorType, 148, 369, 452, 473
Aleri, 2, 33, 41, 43, 46, 58, 151, 179, 229, 233, 235, 432
AllTypes, 440
App, 287, 290, 315, 395, 502
arrays, 28
   empty, 28
AtomicInt, 422
AutoDrain, 418, 513
AutoFnBind, 251, 263, 394, 500
Autoref, 20, 422, 431, 449

B

BasicAggregatorType, 475, 479, 479
BasicPthread, 515, 516
batch, 235
binding, 251
Braced, 284, 386
build, 12
   documentation, 12
   environment, 11
bundling, 45, 53, 83, 233

C

C++, 19, 421
call, 69
Carp, 19

case sensitivity, 29
CCL, 2, 3, 45, 117, 234, 235
CEP, 1
chat server, 305
chunks, 241
ClearingLabel, 79, 361
closure, 140
code, 21
code generation, 98, 131
Collapse, 235, 249, 251, 266, 385
collation, 324
CompactRowType, 441
confess, 8, 19, 436
const, 421
constants, 22, 39, 66, 113, 360
const_Autoref, 423
const_iterator, 421, 423
const_Onceref, 423
copy tray, 114
copying, 425
Coral8, 2, 33, 41, 43, 45, 58, 88, 117, 145, 150, 179, 202, 229

D

data flow, 1, 33
diamond, 199, 231, 323
die, 19
dispatch table, 58, 59
DocBook, 12
download, 11
drain, 293, 311
draining, 44, 69, 70
DumbClient, 62
DummyLabel, 33, 361, 491
dup2, 302

E

EnqMode, 467
enqueue, 69, 360
Erref, 431, 436
error handling, 8, 19, 23, 37, 44, 86, 87, 98, 112, 139, 151
   in C++, 431, 435, 437, 439
   multithreaded, 306
Errors, 431, 436
   multi-line, 432
ESP, 1
Esper, 7, 229
examples, 13, 25
Exception, 435, 436, 438, 440, 451, 454
   modes, 437
execution model, 2
execution order, 229, 231
expect, 318

Colophon

This manual has been typeset using the Docbook tools.