NoWDB

tobias.schoofs@gmx.net

April 20, 2023

Contents

1 Introduction

NoWDB is a kind of database. It merges the concepts of *graph* and *timeseries* database. Timeseries databases typically have simple data models centred around timelines consisting of pairs of the form $(timestamp, value)$ with additional *tags* to distinguish thematically different timelines. An example may be a weather forecast application with timelines describing temperature, humidity and air pressure at certain locations. The values would reflect these measurements and timestamps would refer to the points in time when the measurements were taken. Tags would be used to distinguish timelines (temperature, humidity, pressure) and to identify the location from where the respective measurement comes.

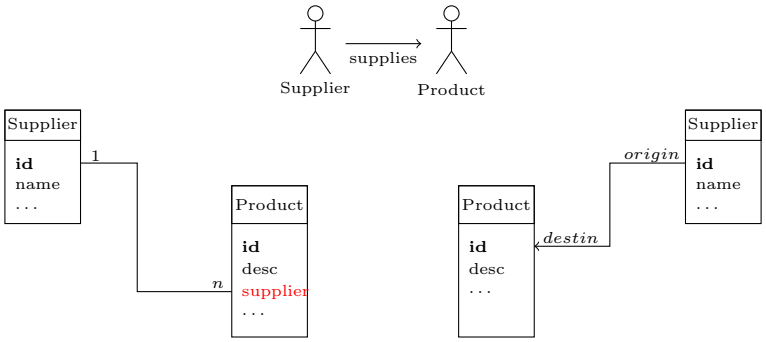
Timeseries databases shine in fast processing of large volumes of data with similar and, typically, simple structure. They are, typically, not good at complex data models like relational databases.

This is where the graph model enters the scene. Graph databases are especially good at efficiently handling data with growing complexity. They are very similar to relational databases, but replace the set-theoretic fundament of the relational world by graph theory. They, hence, do not deal with relations over sets, but with sets of vertices that are connected by edges. In a Twitter-like application, vertices may represent users, while edges may be used to model the connections between users such as *following*. Applications built on top of such a model typically focus on finding relations between vertices; a goal may be to decide whether a user A belongs to the network of a user B where a follower of a follower is considered part of the network. Another challenge may be to compute how many users have seen a certain tweet or how many users see tweets of user A in general.

Relational databases express the relation between entities in terms of *foreign keys*. That is, one entity stores a reference to another entity in its physical representation. An example may be the relation between a *product* and its *supplier*. A relational database would store the *primary key* of the supplier in the product. The relation is this way built into the product table.

Graph databases, by contrast, would express such relations by means of an edge between the two entities. The entities themselves would not be touched. There would be instead a separate entity, *viz.* the edge table, that would link product and supplier.

The following diagram shows the logical relation between supplier and product and, below on the left-hand side, its relational interpretation and, on the right-hand side, the graph interpretation:



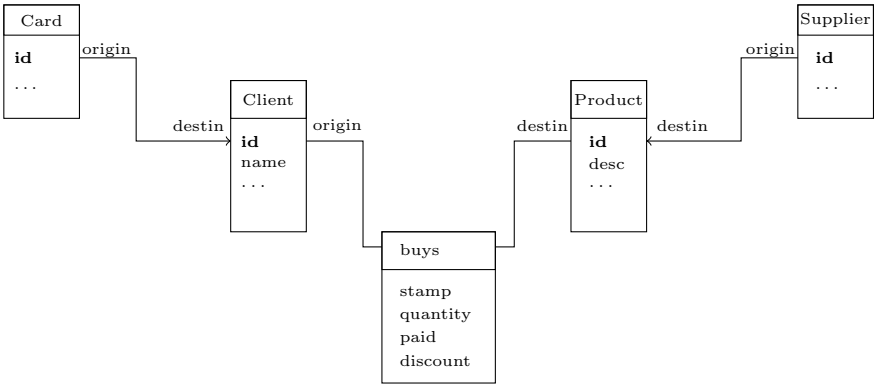
Graph databases are therefore more flexible (but also less rigid). It is easily possible to add new relations without the need to change the structure or the data of the entities involved. Also, there may be more than one relation holding between two entities, for instance, at different points in time. There is also no structural difference between $1 : n$ and $n : m$ relations like in relational databases where a master table must be used to express a many-to-many relation.

In NoWDB edges are not only used to express relations, but also to add information to vertices. This additional information is user-defined, but always timestamped. Since edges are optimised for fast processing of high data volumes, this brings the advantages of timeseries databases to the underlying graph model.

Indeed, NoWDB aims to provide the performance advantages of timeseries databases using concepts from graph databases to allow more complex data models than usually seen in the timeseries world. NoWDB can thus be applied to a wider range of applications than pure timeseries databases without losing their performance advantage.

NoWDB is in particular strong with data that can be organised in some variant of the *star* or *snowflake schema*. Relations between *fact* tables and *dimensional* tables are expressed in terms of timestamped edges between dimensional vertices of arbitrary complexity that, themselves, can be related to each other by simple, unstamped edges.

The following diagram shows part of a snowflake architecture where the dimensional vertices *Client* and *Product* are linked to the stamped edge *buys*, which can be seen as a fact table and which has additional fields, namely a timestamp (*stamp*) and information on the quantity, the price and potential disccount. *Client* and *Product* are additionally related to a (customer) *Card* and a *Supplier* respectively:



The main information encoded in this diagram may be translated to a phrase like "clients buy products (at a given time, for a certain price)"; The customer card and supplier tables add more information in the client and product dimension. That is why this specific diagram is a *snowflake* schema (and not a simpler *star* schema). One can imagine that even more information is added in the dimensions leading to always increasing complexity without affecting the vertex tables and without complicating the main relation between clients and products.

In contrast to traditional *star schema* applications, NoWDB is not limited to data analysis. Many features stress real-time data processing providing *publish and subscribe*, online data filtering and integration with big-data infrastructure.

This manual documents the main features of the database and discusses important use cases. The next chapter provides a *Quick Start* tutorial that helps understanding the concepts behind NoWDB and introduces the most important tools. The chapter will close with an overview of the remaining chapters of this document.

Throughout the document, the reader will encounter red comments like this one. These comments aim to clarify the current state of the prototype. They, in particular, draw attention to features that are not yet available or to shortcomings of their current implementation.

2 Quick Start

2.1 Getting Started

The easiest way to get started is to use the NoWDB docker containing the database server and clients.

more instructions of how to get it ...

The docker does not start the database by itself. You need to start it explicitly. There is a script called `nowstart.sh` in the root of the docker that does that. (You may want to adapt this script to your specific needs!)

Here is one way to start the docker:

```
docker run --rm -p 55505:55505 \
           -v /opt/dbs:/dbs -v /var/log:/log \
           -d nowdbdocker /bin/bash -c "/nowstart.sh"
```

This command creates the docker container and starts it. The parameters are

- `--rm` instructs the docker daemon to remove the container immediately after it will have stopped.
- `-p 55505:55505` binds the host port *55505* to the same docker port.
- `-v` maps the host path `/opt/dbs` to the docker path `/dbs` and the host path `/var/log` to the docker path `/log`.
- `-d` means the docker runs in the background (*detached*).
- `nowdbdocker` is the name of the docker image.
- `/bin/bash` is the command to be executed within the docker; `-c` passes a command to be executed to `bash`, namely `/nowstart.sh`.

The script `nowstart.sh`, contains the instructions to start the NoWDB daemon.

Looking into the script, we see it first sets some environment variables:

```
export LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib
```

This sets the search path for shared libraries.

```
export NOWDB_HOME=/usr/local
```

This sets the *home* directory, *i.e.* the directory where daemon will find resources like the Lua NOWDB script. It is here set to `/usr/local`, which means that the daemon will find the script `nowdb.lua` in the directory `/usr/local/lua`.

```
export LUA_PATH=/lua/?.lua
```

```
export LUA_CPATH=/lua/?.so
```

```
export NOWDB_LUA_PATH=*:/lua
```

These lines set the search paths for the Lua interpreter (for Lua modules and C libraries imported by Lua modules) and for *toplevel* modules imported directly by NOWDB (we will discuss that later).

Then the script starts the daemon itself:

```
nowdbd -b /dbs -l 2>/log/nowdbd.log
```

The script passes two options to the daemon: the base directory where all databases managed by this particular daemon live (`-b /dbs`) and the `-l` switch, which activates server-side Lua support.

Now the daemon is listening to port 55505 and is ready to respond to database requests. The daemon starts by printing a welcome banner to standard output:

```
+-------------------------------------------------------------+

  UTC 2018-10-09T12:03:21.631000000

  The server is ready

    - with lua support enabled
    - with python support disabled

+-------------------------------------------------------------+
  nnnn   nnnn          nnnn    nnnnnn       nnnnnn       nnnnnn
    wi  i   wi       i      i    iw           iw           er
    wi i     wi     n        n    iw         e  wi        e
    wii      wi    wi        iw    iw       e    wi      e
    wi       wi    wi        iw     iw     e      wi    e
    wi       wi     n        n       iw   e        wi  e
    wi       wi      i      i         iw e          wie
   nnnn     nnnn       nnnn             n            n
+-------------------------------------------------------------+

connections: 128
port       : 55505
domain     : any
base path  : /dbs
```

Here are some options provided by the NoWDB daemon (for more details on the server interface, please refer to chapter 5):

- `-b` The base directory, where databases are stored. (default is the current working directory).
- `-s` the binding domain, default: any. If set to a host or a domain, the server will accept only connections from that host or domain. Example: `-s localhost` does only accept connections from the server. The host or domain can be given as name (*localhost*) or as IP address (`127.0.0.1`);
- `-p` the port to which the server will listen; default is 55505, but any other (free) port may be used.
- `-c` number of connections accepted at the same time. If the argument is 0, indefinitely many simultaneous connections are accepted (until the server is not able to serve more requests); otherwise, for `-c n`, n being a positive integer, the server accepts up to n sessions at the same time and will refuse to accept more if this number is reached. The default value for this parameter is 128;

  It should be noted that, up to a certain threshold (which is not configurable) sessions are reused, that is, sessions enter a connection pool from which they are fetched, when new connection requests arrive.
- `-q` runs in quiet mode (*i.e.* no debug messages are printed to standard error);
- `-n` does not print the starting banner;
- `-y` activate server-side Python support;
- `-l` activates server-side Lua support;
- `-V` prints version information to standard output;
- `-?` or `-h` prints a help message to standard error.

2.2 First Steps

Once *nowdbd* is running, we can connect to pass queries to the server. There is a tool to do this from the command line called *nowclient*. Here is a usage example:

```
nowclient -d retail -Q "select count(*) from buys where customer=12345"
```

In this form, the client will try to connect to a server running on the same host and listening to port 55505. Furthermore, it will request to use the database *retail* and send the query indicated by the `-Q` parameter.

If successful, the client will print some processing information to standard error and the query result to standard output, *e.g.*:

```
executing "use retail"
OK
executing "select count(*) from buys where customer=12345"
59
```

With option `-q` we suppress the processing information. We would then only see the result:

```
59
```

Here are more options supported by the client tool:

- `-s` The server address or name, *e.g.* `myserver.mydomain.org` or `127.0.0.1`. Default is `127.0.0.1`;
- `-p` the port to which the database is listening. Default: 55505;
- `-d` the database to which we want to connect. Default: no database at all, which means that we cannot send queries without naming a database. Below we will look at alternatives to using this parameter;
- `-Q` the query we want the database to process;
- `-t` print some (server-side) timing information to standard error;
- `-q` quiet mode: don't print processing information to standard error.
- `-V` prints version information to standard output;
- `-?` or `-h` prints a help message to standard error.

The client tool is able to read from standard input; this way, more than one query can be processed by one call to *nowclient*. The following command processes the same query as the one above, but uses standard input instead of the options `-d` and `-Q`:

```
echo "use retail;select count(*) from buys where customer=12345;" |\
          nowclient
```

Notice that, using standard input, we need to terminate single SQL statements by a semicolon. This is even true for the last statement. Leaving the semicolon out would lead to an error.

Of course, we can do much more useful things than just getting rid of the options. The main point of reading from standard input is that we can put SQL statements into a file and *cat* it to nowclient. A useful example may be:

```
drop schema retail if exists;
create schema retail; use retail;

create large storage sales set stress=constant;
create storage statistics;

create type product (
        prod_key uint primary key,
        prod_desc text,
        prod_price float
);
create type client (
        client_key uint primary key,
        client_name text
);

create stamped edge buys (
        origin client,
        destin product,
        quantity uint,
        price  float
) storage=sales;

load '/opt/data/products.csv' into product use header;
load '/opt/data/clients.csv' into client use header;
load '/opt/data/sales.csv' into buys use header;
```

Let's assume we had this code in a file called `create_retail.sql`; then we could send it to *nowclient*:

```
cat create_retail.sql | nowclient
```

which would create the retail database.

The script shows some of the peculiarities of NoWDB. The beginning is quite regular SQL:

```
drop schema retail if exists;
create schema retail; use retail;
```

The first line drops the database retail, *i.e.* it removes all its data physically from disk. The *if exists* clause is included to avoid an error (and hence the termination of the script) in the case the database does not yet exist.

In the second line the schema 'retail' is created. The third statement (still in the second line) instructs NoWDB to use the newly created schema 'retail' in all following statements.

The next line is a bit uncommon:

```
create large storage sales set stress=constant;
```

The statement creates a *storage* object called 'sales'; A *storage* is a container for *types* and *edges* (which we will sometimes collectively call *tables*, even though this is not the exact same concept as tables in relational databases). A storage, hence, is similar to what is called *tablespace* in other databases.

The statement explicitly says that we want a *large* storage and that there will be *constant* stress (*i.e.* ingestion load) on the objects in this storage. It is good policy to put objects with equal storage parameters, into the same storage. This is especially true for sizing (*tiny*, *small*, *big*, *etc.*). On the other hand, there are good arguments to put objects with very high ingestion load into separate storage containers. This is discussed in more detail in chapters 4 and 15.

The next two blocks of code create the types 'product' and 'client'. Types describe vertices. Each database has a set of vertices that can be connected by means of edges to form graphs. These graphs structure the data in our specific application. The vertex types thus form the universe of discourse.

The attribute types used in the script are *uint*, *float* and *text*. The first is a 64bit unsigned integer; the second is a 64bit floating point number (a.k.a. *double* in languages like C); *text* is a string of up to 255 bytes, which represent UTF-8 characters. For more information on SQL types, please refer to chapter 4.

Every vertex type needs a *primary key* and that primary key must consist of only one attribute. There are no composed keys like in relational databases.

The types created above are stored in a default storage that is created and managed internally by NoWDB. Every database has two internal storage containers, one for types and one for edges. If no storage is indicated in a table (*i.e.* type or edge) definition, the table goes in either of these storage containers, depending on its type.

The next block defines an *stamped* edge. Edges are links between vertices. Regular edges have only two fields, namely: *origin* and *destination* (which is usually abbreviated to *destin* or even *dest*). A regular edge would be created by a statement of the form:

```
create edge from_a_to_b (origin a, destin b)
```

where a and b are vertex types.

Stamped edges have at least one additional field, namely *timestamp* (which is usually abbreviated to *stamp*). This field does not need to be mentioned in the edge definition (in fact, it *must* not be mentioned). Furthermore, stamped edges may have an arbitrary number of user-defined fields. (Well, not really arbitrary: there is a maximum number of 99 fields). The edge 'buys' defined above has two user-defined attributes: 'quantity', a *uint*, and 'price', a *float*.

The *origin* of buys is of type *client* and *destin* is of type *product*. We could, thus, read this definition as a sentence that says something like: "client buys product" or, since we have also a timestamp which is not explicit in the definition: "client buys product at a certain point in time". Considering the user-defined fields, we arrive at a sentence containing even more information: "client buys quantity of product for price at a certain point in time". This is the fundamental idea of NoWDB.

The definition of 'buys' contains a *storage* option. This is the way to tell NoWDB that we want this table to be stored in a particular storage and not in one of the default storages. The edge 'buys', hence, goes into the storage 'sales' (which might or might not contain also other tables).

You may have noticed that we have not created indices. It is possible to define indices on any field or combination of fields as in a relational database using the conventional *create index* statement (please refer to chapter 4 for details). Some indices, however, are created and managed internally, in particular

- the primary key of every type is powered by an index
- every edge (stamped or not) has an index on origin and
- an index on destination.

Thanks to these indices, queries on vertices using the primary key or on edges involving origin or destination are fast. Data access through indices depends mainly on the size of the result set and only marginally on the size of the table in question. One can therefore expect sub-millisecond response time for queries containing tens of thousands of elements in the result set. Obviously, processing time grows with the size of the result set. But it does not grow proportionally in the size of the table itself. Querying the data of a specific origin or destination is not significantly slower with a table containing hundreds of millions or even billions of rows compared to a small table with thousands or hundreds of thousands of rows.

The storage engine additionally favours queries on edges going *forward*, *i.e.* searching destinations for a known origin. Those queries need less disk I/O than queries with other search criteria. Searching *backwards* needs more disk I/O, but is still faster than searching with criteria that do not involve either origin or destination, *e.g.* queries that involve user-defined indices. The worst, of course, is a *fullscan*, especially on large tables. In such cases, restricting the result set by time can accelerate the query significantly. More details on query and data optimisation can be found in chapter 14.

In the final section, the script loads data from three different CSVs into the database. NoWDB provides loaders for various formats. CSV is just one example. There are many more and the loader even allows users to define their own formats using Apache Avro. With Avro it is possible to define binary formats, which can be much faster than textual representations such as CSV.

Using a serialisation system like Avro also eases interoperability of the database with external systems and applications and it significantly eases version management should data formats change over time (what they always do). For more details on data loaders, please refer to chapters 4 and 12.

Loaders, in general, are usually much more efficient than the SQL *insert* statement. The drawback of *insert* is that each statement needs the whole cycle of SQL parsing and execution, while loaders only need one cycle. Since a data source can contain millions or even billions of rows, loading is way more efficient than inserting in most cases.

Notice the *use header* clause in the loader statements. It indicates that the CSV files actually have a header line and that we want NOWDB to use this header to identify the columns. This way, the CSV is independent of the order of columns within the rows. We could also decide to *ignore* the header; in that case, the first line in the CSV will be ignored; or we could not mention headers at all. Then, the first line is interpreted as a regular line containing data. In both cases, the fields will be expected to be in *canonical* order, *i.e.* the order of fields in the definition of the type or edge (where origin, destin and timestamp are always the first three fields in stamped edges).

2.3 First Queries

The alternative to loading data is, of course, the conventional *insert* statement:

```
insert into client values (9000001, 'Popeye the Sailor');
insert into product values (100001, 'Spinach, 450g net', 1.99);
```

These two statements insert a client and a product, respectively. We can also name the attributes explicitly, like:

```
insert into product (prod_key, prod_desc, prod_price)
            values (100002, 'Candy Cigarettes, 20', 2.49);
```

Now we insert a bunch of edges:

```
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100001, '1929-01-17T09:35:12', 1, 1.99)
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100001, '1929-01-19T10:15:01', 2, 3.98)
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100001, '1929-01-20T17:12:55', 3, 5.97)
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100001, '1929-01-22T08:27:32', 1, 1.99)
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100001, '1929-01-25T12:09:59', 1, 1.99)
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100001, '1929-01-26T21:19:44', 2, 3.98)
insert into buys  (origin, destin, stamp, weight, weight2)
  values (9000001, 100002, '1929-01-22T08:27:51', 1, 2.49)
```

Worth noticing here is the time format, which follows ISO-8601.
The format is

- 4 digits for the year and hyphen
- 2 digits for the month and hyphen
- 2 digits for the day of month
- 'T' to mark the beginning of the time section
- 2 digits for the hour and colon
- 2 digits for the minute and colon
- 2 digits for the second and,
- if finer grain is necessary, a dot followed by up to 9 digits for the nanoseconds.

This format can be used anywhere in NoWDB SQL. However, it is also possible to define custom date and time formats. How to do this is discussed in chapter 4.

Worth noticing is also the first date, January, 17, 1929, when Popeye had his first appearance in an American newspaper.

Now that we have inserted some data into our database, we are able to perform *selects*, *e.g.*:

```
nowclient -d retail -Q "select count(*) from buys \
                        where origin=9000001"
```

which would give us 7 and would count Popeye's visits to the supermarket. We can also count how often Spinach was bought:

```
select count(*) from buys
 where destin=100001
```

which shows us 6. Or we can ask how much Popeye bought and paid per type of product:

```
select destin, count(*), sum(quantity), sum(price)
  from buys
 where origin=9000001
 group by destin
```

which would give us:

```
100001;6;10;19.9000
100002;1;1;2.4900
```

Notice that the output of the client tool does not resemble the classical pretty-printed output produced by most database client tools today. The advantage of such output is that it is easier for humans to read. The CSV-like output shown above, however, is better for interoperability, for instance, when we want to combine it through pipes with other programs, like this:

```
nowclient -d retail -Q "select * from buys" | cut -d";" -f2 | ...
```

On the other hand, there are tools that produce more readable output from CSV input, such as `csvlook` from the `csvkit` package.[1] To obtain a traditional pretty-printed output we could do the following (assuming that `query.sql` contains the query we used above):

```
cat query.sql | nowclient | csvformat -d";" | \
header -a 'product,count,quantity,price' | csvlook
```

and would obtain for the grouping query used above:

```
|----------+---------+----------+---------|
|  product | count   | quantity | price   |
|----------+---------+----------+---------|
|  100001  | 6       | 10       | 19.9000 |
|  100002  | 1       | 1        | 2.4900  |
|----------+---------+----------+---------|
```

The disadvantage of the `csvkit` package is that it is somewhat slow compared to the native NoWDB tools. For this reason, there is a more complete client tool for NoWDB (which can be downloaded from xxx). This tool comes with a much larger feature set and is described in another document.

[1] Have a look at `https://github.com/jeroenjanssens/data-science-at-the-command-line`

Here is a more typical time series query illustrating the advantage of the pretty printer:

```
select destin, stamp, quantity, price
  from buys
 where origin=9000001
 order by stamp
```

which, with the same technique as above, shows:

```
|----------+---------------------+----------+---------|
|  product | stamp               | quantity | price   |
|----------+---------------------+----------+---------|
|  100001  | 1929-01-17T09:35:12 | 1        | 1.9900  |
|  100001  | 1929-01-19T10:15:01 | 2        | 3.9800  |
|  100001  | 1929-01-20T17:12:55 | 3        | 5.9700  |
|  100001  | 1929-01-22T08:27:32 | 1        | 1.9900  |
|  100002  | 1929-01-22T08:27:51 | 1        | 2.4900  |
|  100001  | 1929-01-25T12:09:59 | 1        | 1.9900  |
|  100001  | 1929-01-26T21:19:44 | 2        | 3.9800  |
|----------+---------------------+----------+---------|
```

We can also select from vertices, but instead of the edge *buys*, we use a type:

```
select prod_price from product
 where prod_key = 100001;
```

which shows

```
1.99
```

and

```
select client_name from client
 where client_key = 9000001;
```

which gives

```
Popeye the Sailor
```

Much more typical for NoWDB, however, is to use vertices together with edges. Edges connect vertices and can therefore be seen as 'relations'. What we typically want is to combine attributes of vertices and of edges. For instance, we may want to look at the attributes of the destination for a known origin; or we want to look at edges with the attributes of both vertices, origin and destin, added to them.

Both patterns are, in SQL, instances of *joins*. An instance of the first pattern would be:

```
select stamp, prod_desc, prod_price
  from buys join product on destin
 where origin = 9000001;
```

```
|--------------------+----------------------+--------|
| stamp              | product              | price  |
|--------------------+----------------------+--------|
| 1929-01-17T09:35:12 | Spinach, 450g net    | 1.9900 |
| 1929-01-19T10:15:01 | Spinach, 450g net    | 1.9900 |
| 1929-01-20T17:12:55 | Spinach, 450g net    | 1.9900 |
| 1929-01-22T08:27:32 | Spinach, 450g net    | 1.9900 |
| 1929-01-25T12:09:59 | Spinach, 450g net    | 1.9900 |
| 1929-01-26T21:19:44 | Spinach, 450g net    | 1.9900 |
| 1929-01-22T08:27:51 | Candy Cigarettes, 20 | 2.4900 |
|--------------------+----------------------+--------|
```

Note the difference in the price column. In the previous query we used *price* of buys, which (as you may have realised) is the multiplication of product price and quantity. Here, however, we use the value in *prod_price* which is the base price of one unit of that product.

We can, of course, combine joins with grouping:

```
select destin, count(*), sum(prod_price)
  from buys join product on destin
 where origin = 9000001
 group by destin;
```

```
|---------+-------+---------|
| product | count | price   |
|---------+-------+---------|
| 100001  | 6     | 11.9400 |
| 100002  | 1     | 2.4900  |
|---------+-------+---------|
```

The point about the first joining pattern is that it joins only one of the two vertices with the edge. The second pattern is a bit more complex:

```
select stamp, prod_desc, client_name
  from buys
  join product on destin
  join client on origin
 where origin = 9000001;
```

The result of this query would be:

```
|--------------------+----------------------+-------------------|
| stamp              | product              | client            |
|--------------------+----------------------+-------------------|
| 1929-01-17T09:35:12 | Spinach, 450g net    | Popeye the Sailor |
| 1929-01-19T10:15:01 | Spinach, 450g net    | Popeye the Sailor |
| 1929-01-20T17:12:55 | Spinach, 450g net    | Popeye the Sailor |
| 1929-01-22T08:27:32 | Spinach, 450g net    | Popeye the Sailor |
| 1929-01-25T12:09:59 | Spinach, 450g net    | Popeye the Sailor |
| 1929-01-26T21:19:44 | Spinach, 450g net    | Popeye the Sailor |
| 1929-01-22T08:27:51 | Candy Cigarettes, 20 | Popeye the Sailor |
|--------------------+----------------------+-------------------|
```

The joining patterns where the attributes of vertices are directly integrated with edges are so common and for the graph model so natural that there is another way to do this which does not look like relational joins at all, namely *paths*. Indeed, edges form paths through the graph and we can use them directly to walk from vertex to vertex without further indication of how to follow links.

to be continued

2.4 The Python Client(s)

Until here we always used the client *tool* to perform queries. That is certainly an important use case. Much more typical, however, is to develop application code that needs a client API to connect to the database.

NoWDB comes with a native client API that is available in different languages, among others C, C++, Go, Python (both, 2.7 and 3.5) and Lua (5.3).

For Python, there are in fact two APIs. One is very close to the underlying C implementation and, hence, very simple and very fast. The other complies with the Python DB API PEP 249; it is much slower than the simple one, but also more convenient and, most importantly, it can be used with other Python modules such as Pandas.

The module implementing the simple Python NoWDB API is called *now.py* and must be imported into the client program. For the Python interpreter to find this module, it must be in a directory in the PYTHONPATH. There may be different ideas on how to install Python modules. The NoWDB installation will copy all NoWDB-related modules to one specific folder and add this folder to the PYTHONPATH. But you also may install the NoWDB Python API using `pip`. Then, everything is handled by the Python environment and you don't need to care about these things. For more details on installation, please refer to chapter 11.

Anyway, here is a simple Python program:

```
import now

with now.Connection('localhost', '55505', 'myusr', 'mypwd') as c:
    with c.execute("use retail") as r:
        if not r.ok():
            print "cannot use retail: %s" % r.details()
            exit(1)

    with c.execute("select count(*), sum(quantity), \
                                     avg(quantity)  \
                    from buys") as cur:
        if not cur.ok():
            print "ERROR: %s" % cur.details()
            exit(1)
        for row in cur:
            print "count: %d, sum: %d, avg: %.2f" %
                  (row.field(0), row.field(1), row.field(2))
```

The program first creates a *Connection* to a database listening on port 55505 on 'localhost'. It then executes a *use* statement on this connection to indicate the database towards which the following statements are directed.

The result of a *use* statement is a *status*. A status is either OK or an error whose details can be obtained by means of the method *details()*, which returns a string. Status also supports the method *code()*, which would return a numerical error code. It is often useful to know the error code to decide what to do programmatically (abandon the program, retry, try something else, *etc.*)

Results are *resource managers*. That means that they can be used inside a *with* statement. *with* assures that all resources (in this case the result) are freed before the control leaves the *with*-block even if an exception is raised.

There is no harm in using results without a *with* statement. The Python garbage collector (GC) will invoke the method that internally frees all C resources related to results. However, the programmer has little control over when the GC is invoked. It is therefore good style to use *with* blocks. This guarantees that all resources are freed as soon as possible.

The program above then executes a query. This time the execution returns a *cursor* ('cur'). The program checks whether the cursor is in a good state. The statement may have failed on the server side. The cursor would then be in a state that is not OK. In that case, the program prints the error details and exits with return code 1.

Otherwise, if the result was fine, the program iterates over the cursor printing for each row the fields 0-2. Cursors are iterators that allow simple iteration using *for*. The *for*

statement additionally frees the resources related to *row* results as soon as possible. It, hence, fulfills a similar function as the *with* statement.

Moreover, the last iteration of the *for* statement frees the cursor resources. Strictly speaking, the *with* statement around the *execute* is not really necessary. The *for* statement already handles resource management. We could do it like this:

```
for row in c.execute("..."):
  # do something with row
```

There is a convenience interface that eliminates the explicit error handling present in the code above. It is called *rexecute* where the 'r' stands for *raise* indicating that errors will be raised and not returned. Indeed, whenever the result is not OK, *rexecute* raises an error. This way, only successful results are finally passed on to the caller.

Here is an example of an *insert* statement using the *rexecute* interface (we assume that the connection, 'c', was already established):

```
with c.rexecute("insert into buys (origin, product, timestamp) \
                            values (9000001, 100002, \
                                    '1929-01-23T08:45:00')") as rep:
    print "rows affected: %d" % rep.affected()
    print "running time : %d" % rep.runTime()
```

In the case of data manipulation (DML) or data loading (DLL) statements, the result (if there was no error) is a report A report has the methods *affected()*, which indicates the number of rows affected by this statement, *runTime()*, which indicates the running time of the statement in microseconds, and *errors()*, which, in the case of *load*, indicates the number of rows that resulted in an error.

In many cases, however, the programmer is not really interested in the result. She only wants to know if the statement was successful. For this very common use case, there is a convenient interface implemented on top of *rexecute* which is called *rexecute_*. The underscore indicates that the function does not return a result at all and the 'r', as for *rexecute*, indicates that errors are raised. We can therefore simplify the code above to

```
c.rexecute("insert into buys (origin, product, timestamp) \
           values (9000001, 100002, '1929-01-23T08:45:00')")
```

For more details on the Python client and on the PEP 249 compliant API please refer to chapter 8. The underlying C client API is described in chapter 7. Client APIs for other languages are described in language-specific documentations.

2.5 Lua in the Database

Like many other databases, NoWDB supports stored procedures and stored functions, *i.e.* user code that is executed within the database. The main server-side language for implementing stored procedures and stored functions is Lua (version 5.3 or higher). Another option is Python, but the NoWDB Lua environment provides better performance, better integration and more support packages for database development and is therefore the recommended language to implement procedures and functions.

The difference between stored procedures and stored functions is that stored functions run inside an SQL context. They can be used in a *select* clause, for instance. They are limited, however, in what they are allowed to do. In a *select* clause, for instance, they are not allowed to execute DDL, DLL or DML statements. It would indeed be very strange when a query would suddenly change the database.

Stored procedures, on the other hand, cannot run in SQL context. They are executed explicitly by the *exec* statement. In exchange for this limitation, they get a lot of power: stored procedures are allowed to run any SQL code including DML, DLL and even DDL.

Stored procedures are composed of two elements: there interface (or *signature*) and their implementation. The interface defines how the procedure is to be called; the implementation defines what it does.

The interface is created by the SQL statement *create procedure*, *e.g.*:

```
create procedure sales.revenue(pDay time) language lua
```

This statement defines the interface of a procedure called 'revenue' that takes one argument, a timestamp called 'pDay'. Furthermore, the procedure is written in Lua and is implemented in the Lua module 'sales'. The statement does not define a return value for the procedure. But all procedures in NoWDB return a polymorphic *result type* that can be either a status, a report, a row or a cursor.

The *toplevel* module 'sales', which needs to contain a global function called 'revenue' that accepts one parameter, must be located in a directory NoWDB knows about, namely in a path in the environment variable `NOWDB_LUA_PATH`. The entries in this variable have the form: *db*:*path*, where *db* is the name of a database and *path* a common UNIX path. Entries are separated by semicolon, *e.g.*:

```
export NOWDB_LUA_PATH="retail:/opt/retail/lua;otherdb:/my/path/to/lua"
```

The drawback of using an environment variable to define the Lua path is of course that no paths can be added interactively through SQL. On the other hand, declining this possibility eliminates a whole range of security concerns related to executable code in the database server. In fact, there is currently no way to inject Lua code into the database if this option is *not explicitly configured*. There is a dynamic interpreter that

can be called through a predefined interface and allows executing any Lua code sent to the server. But this interpreter must be set up explicitly and should not be used for database servers that are directly reachable from the internet.

Modules imported into toplevel modules via *require* are regularly handled by the Lua interpreter, *i.e.* they must be located in a directory in the regular Lua path.

Once the function *revenue* is created and its code is accessible to the database, it can be executed by an *exec* statement:

```
exec revenue('1929-01-25')
```

Let's look into the module *sales* to see how the procedure is implemented:

```
function revenue(day)
  local td = nowdb.round(day, nowdb.day)
  local tm = td + nowdb.day
  local stmt = string.format(
    [[select origin, sum(price) from buys
       where stamp >= %d
         and stamp < %d
       group by origin]], td, tm)

  local mx = {0,0,0,0,0,0}
  local cur = nowdb.execute(stmt)
  for row in cur.rows() do
      local x = row.field(1)
      if x > mx[2] then
         mx[1] = row.field(0)
         mx[2] = x
      elseif x > mx[4] then
         mx[3] = row.field(0)
         mx[4] = x
      elseif x > mx[6] then
         mx[5] = row.field(0)
         mx[6] = x
      end
  end
  cur.release()
  local typs = {nowdb.UINT, nowdb.FLOAT,
                nowdb.UINT, nowdb.FLOAT,
                nowdb.UINT, nowdb.FLOAT}
  return nowdb.array2row(typs, mx)
end
```

First, you may have noticed that there are no imports. Indeed, the basic NOWDB functionality is present in the global table `nowdb`. All functions and constants defined in this table can be used just like the functionality in the Lua standard packages (*string*, *table*, *os*, *etc.*).

The function revenue starts with the definition of two local variables *td* (for "today") and *tm* (for "tomorrow"). The variable *td* is derived from the parameter by means of the *nowdb.round* function, which takes two parameters: the first is a *time*, the second is a time unit to which we want to round. The *nowdb* package defines several such units, namely second, minute, hour and day. The *round* function provides a simple means to round a time down to one of those units such that the remainder is reduced to 0. Rounding down to day means, for instance, that hour, minute and second are 0. But, caution: the function is quite simple. It does not take care of leap seconds and other subtleties related to time arithmetic. If such functionalities are needed, it is better to work directly with the SQL time functions that actually are aware of more subtle implications of the notion of time.

The variable *tm* is then derived from *td* by just adding one more day.

Both variables are used in the following SQL statement, which sums up the prices paid "today" grouped by clients (*origin*).

Then we create an array with six elements, the local variable *mx*, and, by means of the *nowdb.execute* function, the cursor *cur*, which holds the result of the query.

The function *execute* behaves similar to the *rexecute* function we saw in the Python client. That is, it returns the result of the statement passed in if no error occurred during execution. Otherwise, it "raises" an error, *i.e.* it calls the Lua *error* function. The NOWDB environment ensures that this error is caught and returned to the client as proper result type. This can also be done explicitly in user code by means of the *nowdb.error* function, which expects an error code and an error messages and returns a result representing an error. This, however, is rarely necessary. More typical in error handling is to "raise" an error using the *nowdb.raise* function, which, likewise, expects an error code and an error message, but does not create an error result, but calls Lua's *error* function creating a string composed of error code and error message.

Cursor results have a *rows* method, a generator that can be used to initialise a *for*-loop over all rows in the cursor. In the *for*-loop here we compare the current value of $field(1)$, which holds the sum of the price for this client, with the value stored in *mx[2], mx[4]* and *mx[6]*. In other words, we are searching for the three clients with greatest revenue for this day and store these client keys in *mx[1]*, *mx[3]* and *mx[5]* respectively.

At the end, after the loop, we release the cursor. This is not strictly necessary, since the Lua GC will release all C resources associated with results. However, it is good style to release results when they are not needed anymore. The Lua GC does not know about the C resources and this may lead to situations where a lot of memory is allocated in a session, before the GC is invoked. Explicit release is in particular a good policy for long-running sessions.

As for the Python *for*-block, The Lua iterator releases row results internally; it is therefore not necessary to release the rows allocated in the loop explicitly.

Finally, *revenue* returns the result of a call to *nowdb.array2row*. This function expects two arguments: a list of NoWDB types and an array with values corresponding to these types. The types are *uints*, representing the client key values in the *mx* array, and *floats*, representing the revenue for this day for this client.

nowdb.array2row creates a row result using the array and the types. Internally, it calls the *nowdb.makerow* function and, for each element in the array, it calls the method *add2row* provided for row results. This method expects two parameters: a NoWDB type and a corresponding value. Finally, it closes the result by calling the *closerow* method.

When we call *revenue* from the client like

```
nowclient -d retail -Q "exec revenue('1929-01-22')"
```

the result would look something like this (assuming that we have loaded much more clients and buys into our retail database):

```
9000004;1837.45;9000012;17.00;9000035;1486.57
```

Of course, we can call *exec* also from a Python client and then handle the result as cursor or row, *e.g.*:

```
with Connection("127.0.0.1", "55505", None, None) as c:
  with c.rexecute("exec revenue('1929-01-22')") as row:
    print("1) client %d: %.2f" % (row.field(0), row.field(1)))
    print("2) client %d: %.2f" % (row.field(2), row.field(3)))
    print("3) client %d: %.2f" % (row.field(4), row.field(5)))
```

The execution environment of server-side Lua is the session. Each session has its own Lua interpreter and all packages are loaded into this local interpreter. Programs running in different sessions cannot communicate directly with each other. However, there are inter-session communication means like locks, events and queues for advanced use-cases.

Another consequence of the equation *session* = *interpreter* is that the scope of all global variables (*i.e.* Lua's *global table*) and chunk-wide local variables is the lifetime of the session from connect to disconnect. The following simple module defining an iterative Fibonacci function illustrates this behaviour:

```
local _fibn1 = 0
local _fibn2 = 1

function fibreset()
  _fibn1 = 0
  _fibn2 = 1
end

function fib()
  local f = _fibn1 + _fibn2
  _fibn1, _fibn2 = _fibn2, f
  return nowdb.makeresult(nowdb.UINT, _fibn1)
end
```

The module defines two local (but module-wide) variables *_fibn1* and *_fibn2*, which are initialised to 0 and 1 respectively. The function *fibreset*, which takes no parameters, resets the variables to these initial values.

The function *fib*, which takes no parameters either, computes the next Fibonacci number as *_fibn1* + *_fibn2* and "shifts" the module-wide variables one step up in the Fibonacci sequence, namely to the values *_fibn2*, which is now the new value for *_fibn1*, and *f*, which is the new value for *_fibn2*. Finally, it returns the row result created by the convenience function *nowdb.makeresult*, which creates a row with a single value.

When we create the interfaces for these procedures, we can call *fib* like this:

```
nowclient -d retail -Q "exec fib()"
```

which returns 1.

Repeated execution of the command above will always return the same, *e.g.*:

```
nowclient -q -d retail -Q "exec fib()"
1
nowclient -q -d retail -Q "exec fib()"
1
nowclient -q -d retail -Q "exec fib()"
1
```

This is, because each call creates a new session, which initialises the variables to their starting values 0 and 1. To show more of the Fibonacci sequence we can do:

```
echo "exec fib(); exec fib(); exec fib(); \
      exec fib(); exec fib(); exec fib();" |\
nowclient -q -d retail
```

Now we see:
1
1
2
3
5
8

and to vary a bit:

```
echo "exec fib(); exec fib(); exec fib(); \
      exec fib(); exec fib(); exec fib(); \
      exec fibreset(); \
      exec fib(); exec fib(); exec fib();" |\
nowclient -q -d retail
```

for which we get:
1
1
2
3
5
8
1
1
2.

The NoWDB module provides many more functions, such as time services, more error handling and safe execution and convenience functions for cursors and rows. For more details, please refer to chapter 10. There are also support modules providing more advanced functionality such as IPC (inter-session communication), unique identifiers, result caching, virtual cursors, *NumPy*-like number processing, basic statistics, statistical sampling, feature extraction and so on. The support modules are described in another document.

2.6 What's Next?

This chapter was only a brief introduction to some important features and the general flavour of NoWDB. The remainder of this manual will discuss the main features more deeply.

The next chapter discusses data modelling with the *graph + timeseries* approach.

The following chapter specifies the NoWDB SQL dialect.

Chapters 5 and 6 present the command line tools `nowdbd` and `nowclient` respectively.

Chapters 7 and 8 discuss the native clients in C and Python. Other clients are available, for instance C++, Lua and Go; but those are described in separate documentation sets.

The next chapter, 10 presents the server-side language bindings for Lua at more depth. Server-side Python is discussed in another document.

Chapter 11 provides detailed information on installation of server and client on different platforms.

The next chapters present more features, namely the loader (12) and server-side techniques such as publish and subscribe and filters (13).

The following chapters 14 and 15 discuss technical insight for application designers and DBAs.

Finally, the appendix 16 lists server-side error codes.

3 Data Modelling

stress differences between graph and relational
what is good design / best practice for both: timeseries and graph
examples: wmo (pure timeseries), retail (timeseries + graph)

4 SQL

4.1 Outline

SQL is a language to store, manipulate and query data in a database; traditionally SQL is used with relational databases. In recent years, however, people have started to use SQL also in other contexts, such as *graph* and *timeseries* databases and new patterns are evolving in the language to better address those data models.

SQL consists of statements that, in their turn, consist of clauses. A statement is a piece of SQL code that by itself constitutes a meaningful action. Statements are distinguished in

- DDL: Statements that create, drop or alter entities in the database that hold or define data like storages, types, edges, indices, procedures, *etc.*
- DML: Statements that manipulate data, *e.g. insert*, *update* and *delete*.
- DLL: Statements that load large volumes of data into the database or retrieve large volumes of data from the database.
- DQL: Statements that read data from the database.
- Miscellaneous: Statements that do not fall into any of those categories, in particular *use* and *exec*.

Clauses are parts of statements; a DQL statement, for instance, typically has a *select* clause and a *from* clause and may have additional clauses (*where*, *order by*, *group by* and so on).

Some clauses can appear in more than one type of statement. *update* and *delete* statements, typically, have a *where* clause, but no *select* clause.

Clauses can be seen as logical building blocks of SQL. But they cannot live alone. It is not possible to execute an isolated *where* clause or an isolated *from* clause. The smallest executable unit is therefore the statement.

Clauses are made of keywords, identifiers, numbers, text, symbols (such as $,\doteq()*+-$) and whitespace, *i.e.* ASCII 10 (line break), 13 (carriage return), 9 (horizontal tab) and 32 (space).

Keywords and identifiers are mutually exclusive, that is, if k is a keyword, k cannot be an identifier at the same time. This rule is relaxed in most SQL dialects – and that is a great relief for users, because SQL has an extraordinary large number of keywords which sometimes makes the choice of meaningful identifiers a painful. At the time of writing, the NOWDB parser does not yet relax this rule, but it will do so in the future. Keywords are defined by the SQL specification and represent syntactic elements for chosing actions over entities in the database; identifiers are chosen by the user and refer to constant values or entities in the database, such as edges, types, indices, *etc.*

In this specification, keywords are typeset in boldface (*e.g.* **select**); identifiers are typeset in italics (like 'mytable' in "**create table** *mytable*").

SQL is a textual interface. All statements that are passed to the database have a textual form. The results produced by the database, however, are not. They are binary data which may or may not contain textual elements.

In NOWDB SQL statements are strings of UTF-8 characters. Keywords, identifiers and numbers, however, must contain only characters from the ASCII subset. Identifiers are further restricted: They must start with an ASCII Latin alphabetic ($a \ldots z$ or $A \ldots Z$) and must contain only alphanumerics or the underscore (_). Text, by contrast, may contain any UTF-8 character.

Keywords and identifiers are case-insensitive. There is no difference between 'SELECT', 'select' or 'Select' and so on. Text, however, is case-sensitive; 'hello world' and 'hello World' are not the same thing!

SQL is a *guest* language that needs some kind of framework to support it. One way to provide this framework is the NOWDB client, which provides two "channels" to execute SQL statements in the database, *i.e.* by means of the `-Q` parameter and by means of standard input.

Another way is a host language that provides means to pass SQL statements to the database and means to receive and interpret the results produced by such statements. For NOWDB, Python, C, Go and Lua are available as host languages.

The protocol that defines how data are exchanged between the database and the host system is not part of this specification. Currently, native client and server libraries exist that implement this protocol without exposing it to the user. To support open standards, such as ODBC and JDBC, parts of this protocol will probably be documented and published in the future

4.2 Types

NoWDB SQL has a very simple type system, which is static and safe. This type system is used to design the database and to word SQL statements. We usually refer to it as the SQL Static Types.

However, since SQL is executed in a host environment, there is a second type system to describe the *results* of SQL statements. This other type system is even simpler – it, in fact, consists of only one type. This other type system, however, is dynamic. It is therefore called SQL Dynamic Types.

In the following, we first present the Static System and then the Dynamic System.

4.2.1 Static Types

The static types constitute the NoWDB SQL type system in the strict sense. The static types can be used in SQL statements. Each type is equipped with a declaration form and type constructors that create instances of this type.

The declaration form is used in DDL statements to define types, edges, procedure and functions. In DML, DLL and DQL statements, instances of the types are used, *i.e.* types are not explicitly declared, but used implicitly by means of their constructors, which are sufficient to determine the type uniquely.

In the case of numeric types (integers, unsigned integers and floats), NoWDB silently corrects type mismatches where possible. An unsigned integer inserted into a field where a signed integer is expected, is implicitly converted to an integer; correspondingly a signed integer is converted to an unsigned integer if possible. If the unsigned integer is out of range or the signed integer is negative, the statement is rejected with a type error.

Likewise, signed or unsigned integers are converted to floats if necessary (and possible) and a float might be converted to an integer (or unsigned integer) if it actually represents an integer.

The static types are

Integer
Declaration: *int*, *integer*
Values: $-2^{63} \dots 2^{63} - 1$
Constructors: $\pm n$, where n is an unsigned integer.
Examples: $-1, +0, +1$

Unsigned Integer
Declaration: *uint*, *uinteger*
Values: $0 \ldots 2^{64} - 1$
Constructors: One digit in the range $0 \ldots 9$ or one digit in range $1 \ldots 9$ followed by a sequence of digits $(0 \ldots 9)$.
Examples: $0, 1, 2, 1024$, but not: 01.

Float
Declaration: *float*
Represents a *binary64* IEEE-754 floating point number. For possible values, please refer to the standard or to the table in `https://en.wikipedia.org/wiki/IEEE_754#Basic_and_interchange_formats`.
Constructors: any integer followed by a dot and a sequence of digits, optionally followed by e followed by an integer. The exponential form is not yet available.
Examples: $-1.0, 0.0, 1.0, 3.14159, 1.797693e308$
A specific value of the float type is NaN (not-a-number). Currently there is no constructor for NaN, but it will be necessary.

Time
Declaration: *time*, *date*
Values: UTC 1677-09-21T00:12:44 – UTC 2262-04-11T23:47:16
Precision: nanosecond.
Note, however, that range and precision depend on server configuration. With less precision, a greater range can be reached. Please refer to the database configuration guide.
Timezone: UTC
Constructor: any integer or any string following ISO-8601 or any string following a locally defined time format.
Examples:
1535284617906179393,
'1940-12-21',
'1904-06-16T11:43:10',
'2011-11-11T11:11:11.123456789'

Bool
Declaration: *bool*
Values: *true*, *false* Currently, Bool cannot be stored in the database. The future solution won't be to store single Booleans, anyway, but bit patterns.

Text
Declaration: *text*
Values: UTF-8 string with up to 255 bytes
Constructor: string enclosed by '
Examples:
'hello world',
'州慧照序。'
An important detail is not yet handled: text that *contains* the character '. This is important for recursive SQL, *e.g.*
*exec metaquery('select * from myedge where a = \'some text\'')*

Longtext
Declaration: text
Values: UTF-8 string with up to 4096 bytes
Longtext is not yet available.

Blob
Blob is nice to have, but there are currently no concrete plans to add such a datatype.

NULL
NULL is a special value available for all types. It signals the absence of a value for the requested field. For instance, when a vertex was inserted with a subset of its attributes, a query on that vertex including those attributes in the *select* clause will produce NULL as value in the columns related to these attributes. Note that it is not possible to compare any value to NULL using $=$ *or* $\neq$. The special operator **is** must be used for that purpose, *e.g.* **where** *name* **is not null**.

4.2.2 Dynamic Types

Dynamic types are not used in SQL statements, but rather describe the return values of SQL statements. As such, they live in the context of a host language (C, Python, Lua, *etc.*) and the concrete implementation depends on that language. For more information on concrete implementations of dynamic types, please refer to the host language API specifications.

Status
Values: OK, NOK
The status should additionally provide the values

- error code
- a detailed error message (only in case of error)

Error codes together with a brief description of their meaning can be found in 16.

Report
Reports consist of up to three values:

- number of affected rows (returned by all DML and DLL statements)
- number of errors (returned by all DLL and some DML statements)
- running time (returned by most DML and DLL statements)

Row
A row is the result of a projection; it consists of an array of values with type information called *fields*. In the host language, one would access a field typically by an expression of the form: $row.field(i)$; which would return a tuple $(value, type)$.
A row result consists of one or many rows. The host language shall provide means to iterate over collection of rows.

Cursor
A cursor is an iterable collection of rows. The iteration directive is *fetch*. Each fetch may return one or many rows. A cursor is a server-side resource and shall be closed using the directive *close*.

4.3 Expressions

Some SQL clauses, like *select*, *where* or *insert*, allow the use of *expressions*. Which expressions are valid in a specific context varies between clauses.

The simplest expression, which is available in all clauses (that allow expressions), is a constant value, *e.g.* $true, +4, 3.14159$, 'hello world', *etc.* In some clauses, such as *select*, field names are also valid expressions.

Expressions are evaluated when the SQL statement is processed. The result of the evaluation is a value with a static type. In the case of a constant value, the result is just that constant value. In the case of a field name, the result is the value stored in the corresponding field in the row that is currently processed.

Expressions may be combined with operators or functions to form more complex expressions. If $\circ$ is a binary operator and a and b are valid expressions, then $a \circ b$ is also a valid expression. In consequence, $a \circ b$ can again be used with an operator to form an even more complex expression: $a \circ b \circ c$. Examples of binary operators are $+, -, /$ and $\times$.

To enforce a specific order of evaluation, parentheses may be used. So, if a is a valid expression, (a) is too; for instance $(a + b)/2$ is a valid expression.

Besides binary operators, there are also unary operators, *i.e.* operators that take only one operand. Those operators are usually prefixes. An example of a unary operator is *not*: if a is a valid expression, so is *not* a.

Expressions may also contain functions. Functions are very similar to operators, but differ in syntax. Functions have the form

function(*argument*, *argument*, ...),

where *function* is the name of the function and the *arguments* are again expressions. If Φ is a function that takes one argument and a is a valid expression, then $\Phi(a)$ is a valid expression too. An example of a function is *log*, which computes the natural logarithm for a number a. Valid expressions, hence, are $log(a)$, $log(a)/log(2)$ and $log(2.718281828)$.

The validity of an expression depends also on types. Operators and functions expect arguments of certain types. Some operators and functions are limited to a specific type or type family, *e.g.* to numeric types, to Booleans or to text; when applied to values of types they do not expect, the expression evaluates to a type error and the related SQL statement fails.

In some cases, arguments are *promoted* to other types. Binary operators, for instance, usually expect both operands to be of the same type. If one differs from the other, one of them may be promoted.

Promotion for numerical types follows the precedence: *float* > *int* > *uint*. In the following example, we try to add a *float* and an *int*: $3.14159 + 1$. In this case, the *int* is promoted to a *float* and the result is a *float*, namely 4.14159.

In general, if at least one of the operands or arguments is a *float*, all values are promoted to *float*. Likewise, if at least one of the operands is an *int* (but none of them is a *float*), then all others are promoted to *int*. Non-numerical types (like *bool* and *text*), however, are never promoted to another type.

4.3.1 Operators and Functions

Arithmetic

Arithmetic operators and functions accept only numerical values. If not mentioned otherwise, promotion follows precedence as described above. The basic arithmetic operators are

```
+, -, *, /, %
```

The first three of them have the obvious meaning. There is an annoying ambiguity in the tokeniser affecting + and −. The parser confuses `+b` in `a+b` with the pattern for integers. Workaround is to either introduce a blank before `b`: `a+ b` or to put parentheses around it: `a+(b)`.

Division (/), however, is different in that it has different meanings for different numerical types. For *float* values it performs floating point division. For integral values (type *int* and *uint*), however, it computes the quotient of the Euclidean division, *e.g.* 5/2 is 2. However, 5.0/2 is 2.5, because 2 is promoted to *float* according to the precedence rule.

The fifth operator (%) is the remainder of the Euclidean division. It operates only on integral values and results in a type error if applied to *floats*.

It should be noted that this is not the modulo operator (*mod*) with which the remainder is often confused. People often expect the remainder to always return a non-negative result. But that is not the case. The remainder is the unique number r for which

$a = qb + r, r < b,$

where q is the quotient a/b.

In consequence, if a is negative and b is positive, then the quotient and the remainder are negative, *e.g.* $-7 = -2 \times 3 - 1$ for $a = -7$ and $b = 3$.

If a is positive and b is negative, then the quotient is negative but the remainder is positive, *e.g.* $7 = -2 \times -3 + 1$.

If both, a and b, are negative, then the quotient is positive and the remainder is negative, *e.g.* $-7 = 2 \times -3 - 1$.

Power The power operator is ^. It always promotes all its arguments to *float* and always returns a *float*.

Please notice that there is no *root* operator. Instead, the root is computed by raising to the power of a fraction, *e.g.*:

$\sqrt[2]{n} = n\text{^}(1/2)$,
$\sqrt[3]{n} = n\text{^}(1/3)$,
$\sqrt[4]{n} = n\text{^}(1/4)$,
...

Log As mentioned, the function *log* implements the natural logarithm. The function promotes its argument to *float* and always returns a *float*.

The natural logarithm finds x in the equation $n = e^x$ for a given n, where e is the *Euler-Napier* constant, which is ≈ 2.718281828.

Logarithms to any base other than e can be easily found by the formula $\log(n)/\log(b)$, where b is the desired base. The binary logarithm of n, for instance, is $\log(n)/\log(2)$.

Absolute Value The function *abs* computes the absolute value of its argument, *e.g.*: $abs(-1)$ is 1. The function accepts any numerical type and returns a value of the same type as the argument.

Rounding The rounding functions are *round*, *ceil* and $floor$. All of them promote their argument to *float* and always return a *float* value.

$round(n)$ returns the integer closest to n, which may be n itself; if the next greater and next smaller integer are equally far away, *i.e.* if the digital part of n is .5, the result is the next greater integer.

$ceil(n)$ returns n if n is an integer or, otherwise, the next integer; *i.e.* it always rounds up, *e.g.*: $ceil(1.1)$ is 2.

$floor(n)$ returns either n if n is an integer or, otherwise, the previous integer, *i.e.* it always rounds down; *e.g.*: $floor(1.9)$ is 1.

Trigonometry

NOWDB provides the trigonometry functions *sin* (*sine*), *cos* (*cosine*) and *tan* (*tangent*) as well as their inverses *asin* (*arcsine*), *acos* (*arccosine*) and *atan* (*arctangent*) and the corresponding hyperbolic functions *sinh* (*hyperbolic sine*), *cosh* (*hyperbolic cosine*) and *tanh* (*hyperbolic tangent*) as well as their inverses *asinh* (*hyperbolic arcsine*), *acosh* (*hyperbolic arccosine*) and *atanh* (*hyperbolic arctangent*).

Example:

select $sin(1)$ **from** *mytable*

results in (approximately) 0.84147098.

All these functions expect a *float* argument and also return a *float*. If the argument is of a numeric type other than *float*, the argument is promoted to *float* if possible. The argument is expected to be in units of radians.

The return value of the trigonometric functions *sin* and *cos* is in the range $-1 \ldots 1$.

The tangent function has singularities at odd multiples of $\pi/2$. If the argument is too close to one of these singularities, the result is undefined.

The inverse trigonometric functions *asin* and *acos* are defined over the domain $-1 \ldots 1$. For arguments outside of that domain, *asin* and *acos* return NaN (*not a number*, see IEEE-754). The result of all inverse trigonometric functions is in the range $-\pi/2 \ldots \pi/2$.

The hyperbolic function *acosh* is valid only for values ≥ 1. For smaller values, *acosh* returns NaN.

The hyperbolic function *atanh* is valid only for values ≤ 1. For 1, it returns ∞. For greater values, it returns NaN.

Mathematical Constants

NOWDB defines the mathematical constants π and e (the Euler-Napier number) as functions *pi* and, respectively, *e*. Both expect no arguments and return a float value. Example:

select *pi*(), *e*() **from** *mytable*

returns (approximately) 3.1415926 and 2.7182818.

Conversions

NOWDB provides explicit conversion functions between numerical types. The conversion functions are $tofloat, toint$ and *touint*. If conversion is applied according to precedence, the respective function has the obvious result, *i.e.* converting upwards preserves the full value if possible.

$tofloat(1)$, for instance, is 1.0 and $toint(1)$ is 1. $toint(18446744073709551615)$, however, is -1. This is because 18446744073709551615 is $2^{64} - 1$ and, hence, out of range of *int*. Similar effects happen with applications of $tofloat$ on values out of range. Please refer to the IEEE-754 specification for details.

Converting against precedence, *i.e.* converting downwards, does not always preserve the value. When converting a *float* value to an integral type, the decimal part is lost, *e.g.* $toint(3.14159)$ is 3.

Converting negative numbers to *uint*, does not preserve the value, but the bit pattern. $touint(-1)$, for instance, is 18446744073709551615.

There is also a way to create a textual representation of a number, namely *totext*. This function always produces a textual representation of the constructor for that number expression, *e.g.* $totext(1.0)$ is '1.0', $totext(1)$ is '1' and $totext(+1)$ is '+1'. totext is not yet available. Further missing are the inverse functions parsefloat, parseint, parseuint, parsebool.

Boolean

Boolean operators and functions are those resulting in a Boolean value. They are, hence, not defined by the type of their arguments, but by their result type.

There are different categories of Boolean operators (or functions): boolean operators in the strict sense, comparisons and special operators.

Boolean Operators in the Strict Sense are **and**, **or** and **not**. What makes them strict Boolean operators is that their operands must be Boolean values. Boolean operators can be used in all clauses where expressions are allowed. But they are especially important for the *where* clause and will be discussed in more detail there (please refer to section 4.7.5).

Comparisons compare values and return a Boolean value according to the comparison. Some comparisons are restricted in terms of the types of their arguments; others are unlimited. All comparisons, however, always compare values of the same type. Comparing incompatible values leads to a type error. For numeric types, the same promotion rules as for arithmetic operators apply.

The main comparison operators are

```
= < > <= >= != <>
```

with the obvious meaning for the first five operators; The last two operators are synonym and both implement the mathematical operator $\neq$.

The operators $=$ and $\neq$ can be used with any type. $=$ evaluates to *true* if the values are equal, *i.e.* their type is equal (or compatible via promotion) and their value is equal. For instance, $1 = 1$ is *true*; $1 = 1.0$, after promotion to *float*, is also true; but $1 =$ '1' results in a type error.

Furthermore, 'hello' = 'hello' is *true*, while 'Hello' = 'hello' is $false$.

Finally, $true = true$ and $false = false$ are *true*, while $true = false$ and $false = true$ are $false$.

The operator $\neq$ is *true*, if the operands are not equal. $\neq$, hence, is the negation of $=$. The expression $a \neq b$ is equivalent to $not(a = b)$

The *inequality* operators $<, >, \leq$ and $\geq$ can only be applied to numerical types including time (and result in a type error otherwise).

All comparison operators return $false$ when applied on the special value NULL. That is NULL is neither equal nor not equal to any other value. It is also not greater or less than any other value. It is incomparable.

Special Operators are similar to comparison operators but with peculiar operands. Special operators are **is**, **between**, **in**, **having**, **exists**, **any** and **all**.

between, **having**, **exists**, **any** and **all** are not yet available.

The operator **is** must be used to compare with the special value NULL, *e.g.* 1 **is null** (which is $false$) or 1 **is not null** (which is *true*). The expression a **is not null** is equivalent to **not**(a **is null**).

The operator **between** tests whether a value is in a range. Its first operand is an expression, while its second operand is a range, for instance 1 **between** $[1, 2]$, which evaluates to *true*, since **between** in this form is *inclusive*.

between has a special syntax for the brackets to indicate ex- or inclusiveness: 1 **between** $]1, 2]$, which would evaluate to $false$ because, in this form, the first element is excluded from the range. Correspondingly, **between** $[1, 2[$ would exclude the last element from the range and **between** $]1, 2[$ would exclude both.

between is especially interesting with time periods. We could, for instance, select all data of the first three months of year 2010 with the expression: **stamp between** ['2010-01-01', '2010-04-01'[.

The operator **in** tests whether a value is in a set. Its first operand is an expression, while its second operand is a list of expressions enclosed by parentheses, *e.g.* 1 **in** $(1, 2, 3)$, which is *true*, or 1 **in** $(2, 3, 4)$, which is $false$.

In a *where* clause the second operator of the **in** operator may be a DQL statement, *e.g.* 1 **in** (**select** 1 **from** *sales*). DQL statements may also be used for the second operand of the the operators **exists**, **any** and **all**.

exists tests whether a set contains data (*i.e.* is not the empty set). The basic form is **exists** $(1, 2, 3)$, which is *true*. **exists** is especially useful with subqueries.

Conditionals

The Case Expression Conditionals can be constructed by means of the **case** expression. The general syntax is:

case
    when $condition_1$ **then** $value_1$
    when $condition_2$ **then** $value_2$
    ...
    else $value_n$
end

where the conditions are boolean expressions and the values are any expressions. For instance:

case
    when *price* $>$ 100.0 **then** 'expensive'
    when *price* $>$ 50.0 **then** 'not cheap'
    when *price* $>$ 20.0 **then** 'considerable'
    when *price* $>$ 5.0 **then** 'not for free'
    else 'acceptable'
end

Note that the whole construct enclosed by the keywords **case** and **end** is an expression. **when**, **then** and **else** alone do not form expressions; they can only be used within a **case**.

A **case** expression consists of at least one **when**/**then** construct. Without such a construct, the **case** is invalid.

Likewise, a **case** must contain at most one **else** and that **else** must be placed behind all **when** constructs. With more than one **else** or with an **else** that is followed by one or more **when** constructs, the **case** is invalid.

The **when** constructs are evaluated in the order in which they appear in the **case** expression. The **case**, thus, evaluates to the *value* branch of the first **when** whose condition branch evaluates to **true**.

If none of the **when** conditions evaluates to **true**, the **case** expression evaluates to the **else** *value*. If there is no **else**, the **case** expression evaluates to **null**.

The **when** and **else** values may be of different types. For instance:

case
    when *category* = 1 **then** 'true'
    when *category* = 2 **then** 1
    when *category* = 3 **then** true
    else null
end

However, care must be taken when using this feature. Often the context where the **case** expression appears requires a certain type, *e.g.*:

select 1 + **case**
        when *category* **is null then** 0
        else *price* + category * 0.1
    end

This context expects a numerical value. Note that, if the first **when** is **true**, the whole expression will produce a *uint*; otherwise, it will produce a *float*. Evaluation will fail with a non-numerical value like in the following case

select 1 + **case**
        when *category* **is null then false**
        else *price* + category * 0.1
    end

This is of special importance in the **where** clause. When a **case** expression is used in a **where** like this:

where case ... **end**

i.e. the whole **where** evaluates to the result of the **case** expression, the **case** must evaluate to a boolean. Otherwise the effect of the **where** clause is undefined.

Coalesce *Coalesce* is a function that accepts an unspecified number arguments. *Coalesce* evaluates to the first argument that itself does not evaluate to **null**. If none of the arguments is **not null**, *coalesce* evaluates to **null**. Example:

coalesce(*category*, 99)

If the field *category* is **null**, *coalesce* evaluates to 99; otherwise, it evaluates to the current value of *category*.

coalesce may be shortened to *coal*. So, the following statement is equivalent to the previous:

coal(*category*, 99)

Since *coalesce* accepts an undetermined number of arguments, the following expression is legal:

coalesce(*species*, *genus*, *family*, *order*, *class*, *phylum*, *kingdom*, *domain*, 'life')

The following expression evaluates to **null**, whenever *category* evaluates to **null** (and is as such pointless):

coalesce(*category*, **null**)

By contrast, this statement always evaluates to 99:

coalesce(**null**, 99)

The arguments may be of different types. But, again, care must be taken when using this feature.

Bitwise Operators

$\ll, \gg, \&, |, \sim$, xor, bit

Not yet available. Bitwise operators, however, are very interesting to store bools as bitmaps.

Time

Time Component Functions Time component functions return a part of the time, such as the year, the month, the day of the month, *etc.* as *int* according to the European calendar. Example:

select *year*(**stamp**) **from** *buys*

year returns the year, *e.g.* 2018.

month returns the month starting from January as 1.

mday returns the day of the month (1-31).

wday returns the day of the week with Monday = 1, Tuesday = 2, ..., Saturday = 6 and Sunday = 0.

yday returns the day of the year.

hour returns the hour of the day from 0 ... 23.

minute returns the minute of the hour from 0 ... 59.

second returns the second of the minute, which is almost always in the range 0 ... 59. In case of leap seconds, however, 60 may be returned.

milli returns the milliseconds within the second.

micro returns the microseconds within the second.

nano returns the nanoseconds within the second.

Points in Time The following functions return specific points in time. They take no argument and return a **time** value. Example:

select *now*()

now returns the current system time.

dawn, dusk return the earliest (*dawn*) and the latest (*dusk*) point in time that can be represented.

epoch returns the system's epoch (usually '1970-01-01T00:00:00').

Time Formatting

Not yet available

Geospatial

Not yet available. Planned are: geohashing, distance calculation, bounding boxes, box is in another box, boxes share area, boxes touch each other, etc.

Text

Not yet available

4.3.2 Aggregates

Aggregates are functions (and, hence, may be part of expressions) with very special behaviour and very special restrictions concerning their usage context. While all functions (and operators) we have looked at so far, are applied to *single* values, aggregates are applied to *sets* of values. They produce, as their name suggests, an aggregation of the values in the set like the number of elements in the set (*count*), the sum of all the values in the set (*sum*) or the average (*avg*) or the median (*median*). They, hence, behave like *map* and *reduce* operators.

The set of values to which an aggregate is applied depends on its usage. Aggregates may be applied to all data in the result set or to all data in a *group* (please refer to section 4.7.6 for details).

The aggregate arguments are expressions. There may be restrictions on the type of expression that can be used with a particular aggregate. In general, however, aggregates may be applied to any kind of expression. For instance, $sum(1)$ is equivalent to $count(*)$ and $avg(price\hat{}2)$ would compute the average of the squares of the values of field *price*.

Aggregates can also be part of expressions. The formula $sum(weight)$ / $count(*)$, for instance, would be equivalent to $avg(weight)$.

count

The function counts the values in a set. It takes one argument, which may be any expression, but few expressions really make sense with *count*.

The function is applied either to the row ($count(*)$) or to a part of it ($count(field)$) or to anything else ($count(1)$). However, these expressions are all equivalent.

When applied to a field, *count* is usually combined with **distinct** (which is not yet available), otherwise the result would be the same as applying it to the row. Common examples are therefore:

select *count*$(*)$ **from** *buys*

and

select *count*(**distinct origin**) **from** *buys*

The return type is always **uint**.

sum

The function takes one argument and produces the sum of the values to which its argument evaluates over the result set, *e.g.* the values in a field.

The argument must be numeric. The return type depends on the type of the input field.

Example:

select *sum*(*quantity*) **from** *buys*

produces the sum of the values in the *quantity* field for the whole table *buys*.

max and min

The functions take one argument. Function *max* produces the greatest value in the result set and function *min* produces the smallest value in the result set.

The arguments must be numeric. The return type depends on the type of the input field. Example:

select *max*(*price*), *min*(*price*) **from** *buys*

spread

The function takes one argument. It produces the difference $max - min$, where max is the greatest value in the result set and min is the smallest value in the result set.

The argument must be numeric. The return type depends on the type of the input field.

Example:

select *spread*(*price*) **from** *buys*

avg

The Function takes one argument. It produces the arithmetic mean of the values of this field in the result set, *i.e.*

$$\left(\sum x\right)/n,$$

where x represents the field values in the result set and n is the number of rows (*i.e.* *count*$(*)$).

The argument must be numeric. The return type is **float**.

Example:

select *avg*(*price*) **from** *buys*

median

The Function takes one argument. It finds the central point (or, in case the number of rows is even, the average of the two central points), when the values are ordered.

The argument must be numeric. The return type is **float**.

Example:

select *median*(*price*) **from** *buys*

No precaution is currently taken for the case that the result set outgrows available memory. In that case the query will fail (and probably the queries in other sessions too).

mode

The Function takes one argument. It finds the most frequent value in the result set.

The argument may be of any type. The return type depends on the type of the input field.

select *mode*(*price*) **from** *buys*

Not yet available

stddev

The Function takes one argument. It finds the standard deviation according to the formula:

$$\sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}},$$

where N is the number of rows in the result set, x_i is the ith element and $\overline{x}$ is the arithmetic mean (*i.e. avg*).

The argument must be numeric. The return type is **float**.

select *stddev*(*weight*) **from** *sales*

Note that there is not much optimisation done in aggregates. For instance, *median* and *stddev* create a collection of all values, but each one creates its own collection. That is, the values are collected twice.

4.4 Data Definition

4.4.1 Schema

The keywords *schema*, *database* and *scope* are interchangeable.

CREATE

The *create schema* statement creates an empty database physically on disk. It has the following form:

create schema *mydb*

This would create all objects necessary to manage that database.

The following forms are equivalent:

create database *mydb*
create scope *mydb*

All **create** clauses can be combined with the clause **if not exists**, *e.g.*:

create schema *mydb* **if not exists**

The **if not exists**-clause suppresses the 'duplicate key' error in case the schema already exists. It is a convenient way to avoid that a SQL script is abandoned in such a situation.

DROP

The *drop schema* statement removes a database physically from disk. It has one of the following forms, which are all equivalent:

drop schema *mydb*
drop database *mydb*
drop scope *mydb*

The statement removes all objects and data belonging to the database 'mydb' from disk.

All **drop** clauses can be combined with the clause **if exists**, *e.g.*:

drop schema *mydb* **if exists**

The **if exists**-clause suppresses the 'key not found' error in case the schema does not exist. It is a convenient way to avoid that a SQL script is abandoned in such a situation.

4.4.2 Storage

CREATE

The *create storage* statement creates a new storage entity for edges and types physically on disk.

The simplest form is:

create storage *mystorage*

The keyword **storage** may be decorated with a sizing option:

create big storage *mystorage*

Valid sizing keywords are: **tiny, small, medium, big, large, huge**.

Sizing keywords affect the allocation unit for disk space. The concrete meaning is not part of this specification and may change in the future.

It is also possible to add options to a *create storage* statement. Options have the general form

set *option = value, option = value.*

Valid options and values are listed in the following table:

Option	Values	Meaning	Default
stress	**moderate**	Low ingestion volume with occasional peaks	X
	constant	Constant ingestion of high volume	
	insane	Constant ingestion of very high volume	
disk	**hdd**	Disk space is allocated in large chunks	X
	ssd	Disk space is allocated in small chunks	
	raid	Currently, no effect	
compression	'zstd'	zstd is used for compression	X
	'lz4'	lz4 is used for compression (not available)	
	"	Data in this table are not compressed at all	

The option **stress** affects the number of threads allocated to perform ingestion tasks like compression, sorting and indexing. How many threads are allocated is not part of this specification and may vary between platforms.

The user may decide on the compression algorithm to use. The standard compression algorithm is *zstd*, which is fast, but also has a very good compression ratio. It is recommended to use *zstd* in most cases. *lz4* (not yet available) is faster than *zstd*, in particular on decompression, but has a weaker compression ratio. Finally, no compression at all (empty string) makes sense on small tables that are known never to grow beyond some megabyte in size (or beyond some million edges or vertices).

An example of a *create storage* statement with options is

create storage *mystorage* **set stress** = **constant**, **compression** = 'zstd'

DROP

The *drop storage* statement removes an existing storage entity physically from disk. It has the form:

drop storage *mystorage*

SHOW, DESCRIBE

The command **show storages** returns a cursor with the names of all storages defined in the current database, one storage name per row.

The command **describe** *mystore* returns a cursor with all options set for the storage *mystore*, one option name and value per row.

4.4.3 Type

Types are user-defined vertex types. The syntax resembles very much the *create table* syntax in traditional SQL:

create type *product* (
    prod_key **uint primary key**,
    prod_desc **text**,
    prod_price **float**)

This creates a type called *product* with three attributes: *prod_key*, *prod_desc* and *prod_price*.

There is no limit on the number of attributes a type may have. Indeed, vertices with hundreds of attributes are not uncommon.

Attributes may have any static type with one exception: Each type needs a unique primary key and the field that is primary key must be either **uint** or **text**.

The order in which attributes are declared determines the *canonical order* for this type.

The **create type** statement has an optional **storage** clause of the form

storage = *mystorage*.

The **storage** clause comes after the attribute definition, *e.g.*

create type *client* (
    client_key **uint primary key**,
    client_name **text**
) **storage** = *client_store*

The storage clause defines the storage that manages this specific vertex type. If no storage clause is given, the vertex type is managed by the default storage for vertices.

DROP

The *drop type* statement removes a type and all its vertices from the database and physically from disk. It has the form:

drop type *product*

SHOW, DESCRIBE

The command **show types** returns a cursor with the names of all types defined in the current database, one type name per row.

describe *mytype* returns a cursor with the attributes and attribute types for the vertex type *mytype*, one pair of attribute name, type per row.

4.4.4 Edge

CREATE

The *create edge* statement defines the layout of a specific edge type in the database.

There are two variants of the **create edge** clause. The simple variant defines a simple edge, which must have precisely two fields: **origin** and **destin**, *e.g.*:

create edge *prod_category* (
    origin *product*,
    destin *category*)

The types of **origin** and **destin** are vertex types, since they refer to vertices which are connected by this specific edge. Stored in the database is the value of the primary key of the specific type (*e.g.* **uint** for product).

Edges can also be *stamped*. Stamped edges have additionally to the fields **origin** and **destin** a field **timestamp** of type *time*. This field does not need to and must not be defined. It is always the third field of a stamped edge.

A stamped edge can additionally have up to 99 user-defined fields of any static type. Here is an example:

create stamped edge *buys* (
    origin *client*,
    destin *product*,
    quantity **uint**,
    paid **float**)

This edge has five fields (in canonical order): origin, destin, stamp, quantity and paid.

It is possible to rename the fields origin and destin according to their real purpose in a concrete application (*e.g.* naming *origin client*). This is done by adding an **as** clause to the attribute definition, *e.g.*:

origin *client* **as** *client*

The corresponding field can then be referenced as either **origin** or *client*. Not yet available!

The **create edge** and **create stamped edge** statements have an optional **storage** clause of the form

storage = *mystorage*.

The **storage** clause comes after the attribute definitions.

It defines the storage that manages this specific edge type. If no storage clause is given, the edge is managed by the default storage for edges.

DROP

The *drop edge* statement removes the edge (definition and data) from the database. It has the form (for both simple and stamped edges):

drop edge *buys*

SHOW, DESCRIBE

The command **show edges** returns a cursor with the names of all edges (simple and stamped) defined in the current database, one edge name per row.

describe *myedge* returns a cursor with the attributes and attribute types for the edge *myedge*, one pair of attribute name, type per row.

4.4.5 Index

CREATE

The *create index* statement creates an index physically on disk. It has the form:

create index *myidx* **on** *mytable* (*field1*, *field2*)

where *mytable* may be a vertex type or an edge. The fields ("field1", "field2", *etc.*) are user-defined fields or edge fields. Any combination of fields can be used in index definitions. It is not recommended, however, to create indices on fields that are defined as **float** or **time** (*e.g.* **stamp**).

There will be the possibility to define indices over ranges of **float** fields and periods of **time** fields. But that is not yet available.

NoWDB internally, creates some standard indices, namely on the primary key of vertices and on origin and destin of edges.

The **index** keyword can be decorated with a sizing indication, *e.g.*:

create tiny index *myidx* **on** *mytable* (*field1*, *field2*)

The default sizing is **small**.

It rarely makes sense to create **big**, **large** or **huge** indices. It actually makes sense, when the index has many data points per key. More details on this can be found in 15.

DROP

The *drop index* statement removes an index physically from disk. Example:

drop index *myidx*

4.4.6 Procedure

CREATE

The *create procedure* statement creates a procedure interface in the database. It has the form:

create procedure *mymodule.myfun*(*param1* **uint**, *param2* **text**) **language** *lua*

A procedure may have no parameters. The definition then simplifies to

create procedure *mymodule.myfun*() **language** *lua*

Any number of parameters is allowed and parameters may have any static type.

Known languages are *lua* and *python*.

DROP

The *drop procedure* statement drops a procedure interface from the database. It has the form:

drop procedure *myfun*

SHOW, DESCRIBE

The command **show procedures** returns a cursor with the names of all procedures defined in the current database, one procedure name per row.

describe *myproc* returns a cursor with the attributes and attribute types for the procedure *myproc*, one pair of attribute name, type per row.

4.4.7 Function

Not yet available

4.4.8 Lock

CREATE

The *create lock* statement creates a read-write lock in the current database. It has the form:

create lock *mylock*.

DROP

The *drop lock* statement drops a lock from the current database. It has the form:

drop lock *mylock*

SHOW

The command **show locks** returns a cursor with the names of all lock defined in the current database, one lock name per row.

4.4.9 Event

Not yet available

4.4.10 Queue

Not yet available

4.4.11 Period

Period is not a *first-class citizen* like types, edges, procedures, *etc.* In particular, periods cannot be created or altered. They evolve as an effect of inserting edges into the database. However, periods can be identified and they can be dropped.

Dropping data according to timestamps is an important feature in timeseries databases. Without this feature, databases would grow to an extent that would make efficient queries difficult or even impossible. Deleting data by means of the *delete* statement, however, is not efficient for large amounts of data and, in NoWDB (but also in other databases), deleting the data would not solve the problem anyway, because *delete* does not physically remove the data, but just makes them invisible.

Dropping a period, by contrast, removes all data files that contain only data belonging to that period and removing files is a very efficient operation on most platforms.

The syntax is:

drop period on *mytable*
where stamp between ['2018-01-01', '2018-04-01'[

This would drop all data files of table 'mytable' that contain only data between Jan, 1, 2018 (inclusive) and April, 1, 2018 (exclusive).

Two remarks are in place. First, *drop period* must have a *where* clause and this clause must contain exactly one condition, namely **between** related to the timestamp.

The rationale for this restriction is to avoid accidentally dropping too many data using too complex or incomplete *drop* statements.

To illustrate that, the following example is legal and it drops all data before a given date (which is very common for timeseries databases, that often represent gliding time windows):

drop period on *mytable*
where stamp between [*dawn*(), '2018-04-01'[

Second, *drop period* does not guarantee to drop all data that lie in the period in question – in fact, it does not even guarantee to drop any data at all. It guarantees, however, to remove all files that contain *only* data that lie within the period. In other words, the behaviour is conservative and prefers dropping fewer data than possible over dropping too many data. On the long run, however, with a consistent dropping policy old data will be removed and the database won't grow (except when the periods themselves grow).

4.5 Data Manipulation

4.5.1 Insert

The *insert* statement inserts one or more rows into a given table. The basic form is

insert into *mytable* (*myfield1*, *myfield2*) **values** (value1, value2)

where 'myfield1' *etc.* are fields of entity *mytable* and 'value1' and so on are expressions.

A more concrete example is

insert into *product* (*prod_key*, *prod_desc*, *prod_price*) **values** (100001, 'Spinach', 1.99)

For a stamped edge, this would be:

insert into *buys* (**origin**, **destin**, **stamp**, *quantity*, *price*) (
9000001, 100001, '1929-01-22T08:53:22', 3, $3 * 1.99$)

When the list of values is complete, *i.e.* covers all fields in the table, and respects the canonical order, a shorthand form can be used, *e.g.*:

insert into *product* **values** (100001, 'Spinach', 1.99)

It is also possible (not yet!) to insert data from a query, *e.g.*:

insert into *buys* (
origin, **destin**, **stamp**, *quantity*, *price*) (
select origin, **destin**, **stamp**, *quantity*, *price*
from *another_table*
where ...)

Insert (just as data loading) respects data integrity on vertices. That is, the primary key of the vertex must be unique for that type. Otherwise, *insert* fails with the error *duplicate key.*

Edges, on the other hand, have no primary key. Several edges that all look the same can be inserted without limits. For people coming from the relational world, this may sound

strange. However, edges are timeseries data and in the timeseries world it is completely acceptable and even common to insert the same event for the same point in time. It is not so clear what "same time" shall mean in the first place. Typically a point in time in a timeseries application is not an exact spot (*e.g.* that millisecond or that minute). More often than not events happen in time frames; how to identify and correlate single events is not so much database methodology, but data science.

NOWDB does also not enforce the relation between vertex type and edge. That is, one can insert edges to which no corresponding vertex exists. The reason is that the life cycles of timeseries data and master data are often not in sync. For one, timeseries data may arrive, before the respective vertices have been inserted into the database and, even more typical, timeseries data might outlive their vertices in the database. A client, for instance, who does not renew his or her customer card, may be removed from the database; the timeseries data in the database, however, are still there.

4.5.2 Update

The *update* statement changes the values of fields in rows in tables or types. Its general form is

update *mytable*
    set field = value,
    set field = value
 where ...

For instance:

update *product*
    set *prod_price* = 1.89
 where *prod_key* = 100001

For details on the *where* clause, please refer to the DQL section.

update has still many unsolved issues. For instance, when changing primary keys and other indexed fields, we need to delete that particular row from the index and add it again with the new value. That, however, is quite expensive. It therefore may take still some time to make update available :-(

4.5.3 Upsert

4.5.4 Delete

The *delete* statement eliminates single data points from tables. Its general form is

delete from *mytable* **where** ...

A more concrete example:

delete from *product* **where** *prod_key = 100001*

For details on the *where* clause, please refer to the DQL section.

It is worth mentioning that delete does not physically remove data from disk. It marks the corresponding rows as deleted.

4.6 Data Loading

4.6.1 Create

The *create loader* statement creates a user-defined loader.

Not yet avalailable

4.6.2 Drop

The *drop loader* statement removes a user-defined loader form the database.

Not yet avalailable

4.6.3 Load

The *load* statement loads data from an external data source into the database. Its general form is

load '/path/to/datafile' **into** *mytype_or_edge*

The *load* statement has optional *use* and *ignore* clauses

use loader *myformat*

With this clause, a user-defined loader is applied. Without this clause the default loader (CSV) is applied. That is equivalent to

use loader *csv*

In the case of a CSV loader, the **header** option can be used:

use header

or:

use csv, header

which means that the data source has a header and that the loader will use this header to determine how to load the columns in the data source. If the data file has a header, but the user decides not to use it, the *ignore* clause can be used:

ignore header

This will ignore the first line of the CSV.

Here is a complete example for the default loader:

load '/opt/import/client.csv' **into** *client* **use header**

The *load* statement returns a report on success that indicates how many rows have been loaded, how many rows have failed and how long it took. Notice that *load* does not stop on errors. Instead, errors are written to a file with the row number where the error occurred. This way, the faulty lines can be corrected and reimported later.

Here is how to indicate an error file:

load '/opt/import/transactions.csv' **into** *buys*
  set errors = '/opt/import/transactions.err'

If no error file is indicated, the errors are written to standard error. This rarely makes sense, since different sessions can load data concurrently. Standard error would then contain a mix of errors of all sessions that tried to load data.

4.6.4 Dump

Not yet available

4.7 Data Querying

There is only one type of DQL statement. However, this statement is much more complex than the statements we have seen so far.

A DQL statement consists of at least a *select* clause (also called *projection* clause) and, in most cases, a *from* clause, which in itself may contain a *join* clause. The DQL may additionally contain a *where* clause, a *group* clause and an *order* clause. having, limit, sample

4.7.1 Select Clause

The basic form of a *select* clause is

select *expression*, *expression*, ...

If the statement has a *from* clause, any expression is allowed. If it consists only of the *select* clause, fields are not allowed.

If the expression involves fields, it must be fields of the entity (edge or type) referenced in the *from* clause. If all fields of that entity are selected, the statement can be simplified to

select $*$

It is also possible to refer explicitly to the data source (which is identified in the *from* clause), *e.g.*:

select *product.prod_price*

In the *from* clause, we can define aliases for data source and then refer to the data source by this alias, *e.g.*:

select *p.prod_price*

where *p* is an alias defined in the *from* clause. This is especially useful with joins.

Expressions are, of course, not necessarily fields. They also may be functions, constants or even complex expressions composed of fields, constants, functions and operators. The following clauses are all valid:

select *true*
select 'X'
select 3.14159
select *sum*(*price*) / *count*($*$)
select *count*($*$), *sum*(*price*)

The first three clauses may appear pointless. Why select constant values from the database? There are, however, very common use cases for selecting constant values, in particular in combination with the **exists** operator, but also for DQL used within an *insert* statement.

The *select* clause may also contain aggregates, which are applied to partitions of the result set. On which partition aggregates are applied, depends on the *group* clause. If no *group* clause is present, the partition consists of all rows produced by the DQL statement.

The following are valid *select* clauses:

select *count*($*$)
select *sum*(*price*)
select *sum*(*log*(*price*))
select *avg*((*price* + *price*) / 2)
select *stddev*(*price*)

The freedom of the *select* clause is restricted by other clauses, in particular the *from* clause and the *group* clause. In the *select* clause, only those fields may appear that are actually part of the entity chosen in the *from* clause. The interdependencies with the *group* clause are more subtle and will be discussed later.

A meaningful example of a DQL statement that consists only of a *select* clause is:

select *now*()

which obtains the current time. In a *select* clause without a *from* clause fields, obviously, have no meaning. Thus, only constants, functions and operators are allowed.

There is a subtle difference between *select*-only statements and "complete" DQL statements. Complete DQL statements always result in a cursor, while *select*-only statements result in a single row. This has no impact on queries sent from a client, since rows are always wrapped into a cursor before they are sent to the client. In server-side programming, however, this actually makes a difference (see chapter 10).

It is also possible to define an alias for fields in the *select* clause by means of the **as** keyword, *e.g.*:

select *count*(∗) **as** *cnt*

The effect is that some client APIs (see for example chapter 9) can refer to columns in a cursor by the alias, *e.g.*

```
print(mycursor['cnt'])
```

4.7.2 From Clause

The *from* clause determines the data source of the DQL statement. The simplest form of a *from* clause is

from *mytable*

The identifier must refer to an edge or a type. Valid *from* clauses are for instance:

from *buys*
from *product*
from *client*

Here is a first example of a complete DQL statement:

select origin, **destin**, **stamp**, *quantity*, *price* **from** *buys*

It is possible to define an alias for the data source:

from *buys* **as** *b*

The table *buys* can now be referred to as "b" in all other clauses. This technique is especially interesting in combination with joins.

In most SQL dialects, it is common to select data from different data sources at once, *e.g.*:

from *product*, *client*

This form is not supported by NOWDB. Whenever more than one data source is addressed, a join has to be used. However, there are paths which are more handy than using joins in many cases.

4.7.3 Join Clause

A join combines vertices and edges and, through edges, vertices with each other. The basic form is:

join *mytype* **on edgefield**

To make this more concrete:

from *buys* **join** *client* **on origin**

This would produce an *inner join* between *buys* and *client*. Joins in NOWDB are in fact always inner joins. In consequence, there is no difference between *left* and *right*; one could say, joins in NOWDB are *abelian*.

A way to emulate outer joins is using *paths* (see section 4.7.4 for details).

Since the primary key of a type is known and there is always exactly one field which is primary key, the foreign join key (that of *client*) needs no explicit mentioning. It is implicitly clear that **join** *client* **on origin** joins on **origin** = *client_key*.

Every edge connects two vertices. A join, therefore, consists of at most two sub-joins. The syntax is straightforward:

from *buys* **join** *client* **on origin**
join *product* **on destin**

With this join, all attributes from *buys*, *client* and *product* are available in all clauses.

It may happen that the joined entities have fields with the same name. We could have named the primary key in both, *client* and *product*, *key*, instead of *client_key* and *prod_key*. To distinguish the fields, one has to use the entity name together with the field name in the *select* clause and (as we will see) in all other clauses that refer to fields. Example:

select *client.key*

Here, using aliases comes in handy, *e.g.*:

from *buys* **as** *b* **join** *client* **as** *c* **on origin**
join *product* **as** *p* **on destin**

In the *select* clause, we can now refer to fields with the alias instead of the full entity name, *e.g.*:

select *c.key*, *p.key*

Joins are not yet available.

4.7.4 Paths

tbd

4.7.5 Where Clause

The *where* clause adds criteria for the selection of specific rows. *where* clauses can be very complex. Indeed, a *where* clause is one complex *Boolean* expression.

A very simple *where* clause would be:

where *prod_key* = 100001

Which evaluates to true for all rows that have 100001 as *prod_key*.

Since *where* clauses are Boolean expressions, it is possible to use the Boolean operators **and**, **or** and **not**. The first two operators are binary, *i.e.* they expect two operands (which again are Boolean expressions), while **not** is unary and expects one operand (which again is a Boolean expression).

Example:

where *destin* = 100001 **and** *stamp* $\leq$ '2018-04-01'

Before we look at more interesting cases, let's assume the following set of data:

```
|---------+-------------|
| destin  | timestamp   |
|---------+-------------|
| 100001  | '2018-03-15' |
| 100002  | '2018-03-15' |
| 100001  | '2018-04-15' |
| 100002  | '2018-04-15' |
| 100001  | '2018-05-15' |
| 100002  | '2018-05-15' |
| 100001  | '2018-06-15' |
| 100002  | '2018-06-15' |
|---------+-------------|
```

Here is a tricky **where** clause:

where *destin* = 100001 **and** *stamp* < '2018-04-01' **or** *stamp* ≥ '2018-05-01'

or has precedence over **and**. Both, **or** and **and**, bind to the left, *i.e.* the first operand is the one in front of the operator.

That means, here, that **or** is at the top-level: The clause selects all rows that have **stamp** ≥ May, 1, and those that have **destination** 100001 and **stamp** less than April, 1. In other words, what we see is

```
|---------+-------------|
| destin  | timestamp   |
|---------+-------------|
| 100001  | '2018-03-15' |
| 100001  | '2018-05-15' |
| 100002  | '2018-05-15' |
| 100001  | '2018-06-15' |
| 100002  | '2018-06-15' |
|---------+-------------|
```

This might be surprising at the first glance. But, indeed, most SQL dialects follow this convention.

To force another binding, parentheses can be used:

where destin = 100001
  and (**stamp** < '2018-04-01' **or stamp** ≥ '2018-05-01')

This clause would now select all rows that have **destination** 100001 and a **timestamp** that is not in April, like this:

```
|---------+-------------|
| destin  | timestamp   |
|---------+-------------|
| 100001  | '2018-03-15' |
| 100001  | '2018-05-15' |
| 100001  | '2018-06-15' |
|---------+-------------|
```

Equivalent to this second clause is

where *destin* = 100001
  and not (**stamp** $\geq$ '2018-04-01' **and stamp** $<$ '2018-05-01')

or and **and** have precedence over **not**; **not** binds to the right, *i.e.* **not** is a *prefix*.

Would we again leave out the parentheses, like this:

where *destin* = 100001
  and not stamp $\geq$ '2018-04-01' **and stamp** $<$ '2018-05-01'

we would select all rows that have **destination** 100001 and not a **stamp** greater April and that are before May, hence:

```
|---------+-------------|
| destin  | timestamp   |
|---------+-------------|
| 100001  | '2018-03-15' |
|---------+-------------|
```

4.7.6 Group Clause

Grouping partitions the result set according to a set of *keys*. The result set will then be presented according to these partitions. The keys used to partition the set are defined in the *group* clause. A simple *group* clause could be:

group by destin

This clause would partition the result according to **destin**.

In the simplest form a statement with group clause could look like this:

select destin from *buys* **group by destin**

Let's consider an edge with the user-defined field *category* and the following set of data:

```
|----------+---------+-------------|
| category | destin  | timestamp   |
|----------+---------+-------------|
| 'buys'   | 100001  | '2018-03-15' |
| 'buys'   | 100002  | '2018-03-15' |
| 'buys'   | 100001  | '2018-04-15' |
| 'buys'   | 100002  | '2018-04-15' |
| 'buys'   | 100001  | '2018-05-15' |
| 'buys'   | 100002  | '2018-05-15' |
| 'buys'   | 100001  | '2018-06-15' |
| 'steals' | 100002  | '2018-06-15' |
|----------+---------+-------------|
```

We apply the following grouping to this data set

select *category*, **destin from** *buys* **group by** *category*, **destin**

This would produce the following output

```
|----------+---------|
| category | destin  |
|----------+---------|
| 'buys'   | 100001  |
| 'buys'   | 100002  |
| 'steals' | 100002  |
|----------+---------|
```

It, hence, produces a set of distinct key values. One could say that grouping *abstracts* or *reduces* the data set according to the keys. This, in itself, is often a useful feature, *viz.* when we want to apply a certain processing only once per key. As an aside it may be mentioned that this kind of grouping is extremely fast in NOWDB; for performance considerations in general, please refer to chapter 14.

The real power of grouping becomes evident when we let the database do the processing. This is the main purpose of aggregate functions (please refer to section 4.3.2 for details).

The aggregate functions go to the *select* clause behind the grouping keys:

select *category*, **destin**, *count*(∗) **from** *buys* **group by** *category*, **destin**

This would now produce:

```
|----------+---------+-------|
| category | destin  | count |
|----------+---------|-------|
| 'buys'   | 100001  |  4    |
| 'buys'   | 100002  |  3    |
| 'steals' | 100002  |  1    |
|----------+---------+-------|
```

The *group* clause has strong interdependencies with the *select* clause. In particular, the *select* clause must contain those fields mentioned in the *group* clause and the order of the fields must be identical. The only things allowed in the *select* clause besides the grouping keys are aggregates. Some of these requirements will be relaxed in the future.

The following statements are therefore wrong:

select *category*, *count*(∗) **from** *buys* **group by category, destin**

select destin, *category*, *count*(∗) **from** *buys* **group by** *category*, **destin**

select *category*, **destin**, 'hello world' **from** *buys* **group by** *category*, **destin**

Notice that aggregates without grouping are applied to the whole result set. For instance, to count all rows in *buys*, one could say:

select *count*(∗) **from** *buys*

The partition to which *count* is applied is here the whole result set.

Syntactically, even this is OK:

select *category*, **destin**, *count*(∗) **from** *buys*

The values shown for *category* and **destin**, however, have no relation whatsoever to the *count* result. They correspond to any of the rows in the result set, the first, the last or any other. That is unspecified.

To be discussed: grouping and having.

Currently, grouping is only possible for combinations of keys for which an index exists. That is to say: the index must be defined over the same set of keys and the order of the keys must be the same. Otherwise, when there is no matching index, the query fails with error.

4.7.7 Order Clause

The *order* clause sorts the result set according to a set of sorting criteria (the *order keys*). Its simplest form is just:

order by *category*, **destin**

which would present the result set sorted by *category* and **destin**.

The statement
select *category*, **destin from** *buys* **order by** *category*, **destin**

applied to the data set already used in section 4.7.6 would produce the following output:

```
|----------+---------+-------------|
| category | destin  | timestamp   |
|----------+---------+-------------|
| 'buys'   | 100001  | '2018-03-15' |
| 'buys'   | 100001  | '2018-04-15' |
| 'buys'   | 100001  | '2018-05-15' |
| 'buys'   | 100001  | '2018-06-15' |
| 'buys'   | 100002  | '2018-03-15' |
| 'buys'   | 100002  | '2018-04-15' |
| 'buys'   | 100002  | '2018-05-15' |
| 'steals' | 100002  | '2018-06-15' |
|----------+---------+-------------|
```

Similar to grouping, ordering works only with an index that has the same keys in the same order.

4.7.8 Limit and Sample Clause

not yet available

4.8 Miscellaneous

4.8.1 Use

The *use* statement sets the database for the current session. All following statements until the next *use* statement will be applied to this database. Example:

use *retail*

The statement returns a *Status* result.

4.8.2 Exec

The *exec* statement executes a stored procedure whose interface was previously defined. The simplest form is

exec *myprocedure*()

In this case *myprocedure* has no parameters. A concrete example with parameters may be

exec *revenue*(9000001, '2018-05-01')

The result of the statement depends on the procedure.

4.8.3 Lock

The *lock* statement acquires a lock for reading or writing. The simplest form is

lock *mylock*

This would acquire *mylock* for writing. This can also be done explicitly:

lock *mylock* **for writing**

or, for reading:

lock *mylock* **for reading**

When locking for reading, the lock is acquired if no session is currently holding the lock for writing. That is, locking for reading is not exclusive. Many sessions may hold the same lock for reading at the same time. Locking for writing, however, is exclusive. A lock may only be acquired for writing, when it is not held by another session (for reading or writing) and, when a lock is held for writing, no other session can acquire this lock for neither reading nor writing.

If a lock is already held, so that the current lock request cannot acquire it, the session blocks until the lock is released. There is no guarantee of any particular order in which locks are acquired. If two sessions are waiting for the same lock, any of the two will acquire the lock when it is released.

The *lock* statement allows for specifying a timeout by means of an option clause, *e.g.*

lock *mylock* **for writing set timeout** = 100

The timeout is specified in milliseconds. If the timeout expires, the statement fails with the error "timeout expired". If the timeout value is 0, the session will not block in case the lock is already held by another session, but return immediately with *timeout expired*. If no option is given, the session blocks until it has successfully acquired the lock.

It is not allowed to acquire a lock already held by the caller. Attempts to do so are detected and the error "selflock" is generated. Other kinds of deadlocks are currently not detected.

4.8.4 Unlock

A lock is released by the *unlock* statement, *e.g.*:

unlock *mylock*

If the calling session is not currently holding the lock, the statement fails with "does not hold the lock". Otherwise, the lock is released and immediately afterwards acquirable.

5 The NoWDB daemon

5.1 Outline

NoWDB implements a classic client/server approach. The NoWDB daemon acts as the server side, that is, the daemon process implements the server; the terms *daemon* and *server* are therefore interchangeable. One server may run one or more databases, which reside in the same base directory on the server machine. Several servers may run on the same machine, but need to use different database directories and, of course, different ports.

Clients connect to the server through TCP/IP. A TCP/IP connection constitutes a *session*. It is always possible to execute SQL statements within a session and it may be possible to execute server-side Lua or Python scripts, depending on the command line options passed to the process on start-up.

For Lua and Python, the session also defines the scope and lifetime of global and module-wide variables. This makes it possible to implement stateful server modules that maintain state for as long as the session exists. This feature allows implementing a wide range of patterns, such as complex queries using internally one or more cursors or partial computation results, but also long running publish & subscribe services.

The NoWDB daemon is a POSIX-compliant command line program optimised and tested for 64bit machines running under Linux. Clients are available for a wide range of languages (C, Python, Lua, Go, *etc.*) and platforms (Linux, Apple, Windows). In the future, the NoWDB daemon may address other platforms as well. But, currently, this is not a priority.

The behaviour of the daemon process is influenced by a number of command line parameters and environment variables. Configuration files are (currently) not necessary. In the following two sections, we will discuss available command line parameters and environment variables. The last section will present the NoWDB docker. The docker eases the installation especially of the server-side components and the maintenance of NoWDB servers.

5.2 Command Line Parameters

The program accepts the following command line options:

- `-b`: The base directory, where databases are stored. (default is the current working directory).
- `-s`: the binding domain, default: any. If set to a host or a domain, the server will accept only connections from that host or domain. Example: `-s localhost` does only accept connections from the server. The host or domain can be given as name (*localhost*) or as IP address (`127.0.0.1`);
- `-p`: the port to which the server will listen; default is 55505, but any other (free) port may be used.
- `-c`: number of connections accepted at the same time. If the argument is 0, indefinitely many simultaneous connections are accepted (until the server is not able to serve more requests); otherwise, for `-c n`, n being a positive integer, the server accepts up to n connections at the same time and will refuse to accept more if this number is reached. The default value for this parameter is 128;

  It should be noted that, up to a certain threshold (which is not configurable) sessions are reused, that is, sessions enter a connection pool from which they are fetched, when new connection requests arrive.
- `-q`: runs in quiet mode (*i.e.* no debug messages are printed to standard error);
- `-n`: does not print the starting banner;
- `-y`: activates server-side Python support;
- `-l`: activates server-side Lua support;
- `-V`: prints version information to standard output;
- `-?` or `-h`: prints a help message to standard error.

5.3 Environment Variables

The following environment variables have influence on the behaviour of the program:

`LD_LIBRARY_PATH`
This is the Linux search path for shared libraries. If libraries needed by the daemon are not in the standard path `/usr/lib`, the path where those libraries were actually installed must be added.

Non-standard libraries used by the daemon are:

- The *tsalgo* library that provides fundamental algorithms and datastructures;
- The *beet* library that provides on-disk B$^+$Trees needed for indices;
- The *zstd* and *lz4* libraries both implementing compression algorithms;
- The *libcsv* library that implements efficient CSV handling.
- avron
- The *icu* library providing UTF-8 handling for C;
- The *embed*TLS library which equips NoWDB with secure communication;
- The *python* 3.5 library;
- The *lua* 5.3 library.

`PYTHONPATH`
This is the search path used by the Python interpreter to find imported modules and packages.

`LUA_PATH`
This is the search path used by the Lua interpreter to find imported modules. The syntax for this variable is special. For details, please refer to the Lua 5.3 documentation.

`LUA_CPATH`
This is the search path used by the Lua interpreter to find C libraries that are used in Lua modules. The syntax for this variable is special. For details, please refer to the Lua 5.3 documentation.

`NOWDB_LUA_PATH`
This is the search path used by NoWDB to find top-level user modules. The syntax for this variable is special and is discussed in detail in section 10.

`NOWDB_HOME`
This variable shall hold the path to NoWDB resources, such as the Lua and Python NoWDB modules. Where this path is located on your system depends on the installation. Please refer to 11 for details.

5.4 The nowdb Docker

The NOWDB docker provides a ready-made environment for the server to run. All libraries are installed, relevant environment variables are set and the daemon is started with a consistent parameter setting. All decisions can be reviewed by the user and alternative settings can be chosen.

Within the docker, NOWDB is started by means of a shell script. One possible way to start the docker is:

```
docker run --rm -p 55505:55505 \
           -v /opt/dbs:/dbs -v /var/log:/log \
           -d nowdbdocker /bin/bash -c "/nowstart.sh"
```

This command creates the docker container and starts it. The parameters are

- `--rm` instructs the docker daemon to remove the container immediately after it will have stopped.
- `-p 55505:55505` binds the host port *55505* to the same docker port.
- `-v` maps the host path `/opt/dbs` to the docker path `/dbs` and the host path `/var/log` to the docker path `/log`.
- `-d` means the docker runs in the background (*detached*).
- `nowdbdocker` is the name of the docker image.
- `/bin/bash` is the command to be executed within the docker; `-c` passes a command to be executed to `bash`, namely `/nowstart.sh`.

For more information on how to start, stop and manage dockers, please refer to the docker documentation at `https://docs.docker.com/`.

The script `nowstart.sh`, contains the instructions to start the NOWDB daemon:

```
export LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib
export NOWDB_HOME=/usr/local
export LUA_PATH=/lua/?.lua
export LUA_CPATH=/lua/?.so
export NOWDB_LUA_PATH=*:/lua
nowdbd -b /dbs -l 2>/log/nowdbd.log
```

The script sets all relevant environment variables, namely the `LD_LIBRARY_PATH`, `NOWDB_HOME`, `LUA_PATH` and `LUA_CPATH` as well as the `NOWDB_LUA_PATH`. Note that the `PYTHONPATH` is not set. This is because Lua is considered the default server-side language.

Then, the NoWDB daemon itself is started. The script passes two options to the daemon: the base directory where all databases managed by this particular daemon live (`-b /dbs`) and the `-l` switch, which activates server-side Lua support.

The docker also contains the NoWDB command line client and the client libraries for C, Python, Lua and Go. This way the docker can be used to work with the server interactively without the need to additionally install programs on the server or the client; the docker can also be used to prototype a complete solution including server and client applications.

6 The Client Tool

6.1 Outline

This chapter presents the *simple client*, *nowclient*, which comes with basic NoWDB installation and is available in the NoWDB docker. The advantage of this client is that it is extremely simple, it has a very small code base, it is fast and very easy to use. On the other hand, it does not implement advanced features like an interactive user interface or pretty printing query results. There is a more complete client implemented in Go (or Rust?) that can be downloaded here.

The simple client is a program to send a single statement or a sequence of statements to the NoWDB server. The result, in case of DQL statements, is presented in a CSV-like format on *stdout* (with semicolon instead of comma).

For a single SQL statement, the program provides the command line parameter `-Q` (see next section for details). Without this option, *nowclient* expects the statement(s) to be sent to the server on *stdin*. The program could, hence, be called in the form:

```
"use retail; select count(*) from buys;" | nowclient
```

This line would send two statements to the server, a *use* statement and a *select*. Note that both statements are terminated with a semicolon. This is mandatory, when sending queries through *stdin*.

In this example, no command line options are used. The program will therefore fall back to its default behaviour. The following section discusses the command line options.

6.2 Command Line Parameters

Options known to *nowclient* are:

- `-s`: The server address or name, *e.g.* `myserver.mydomain.org` or `127.0.0.1`. Default is `127.0.0.1`;
- `-p`: the port to which the database is listening. Default: 55505;
- `-d`: the database to which we want to connect. Default: no database at all, which means that we cannot send queries without naming a database. Below we will look at alternatives to using this parameter;
- `-Q`: the query we want the database to process; the query is added as a string, *e.g.*:

  ```
  nowclient -d retail -Q "select count(*) from buys"
  ```

  It is not necessary to end the statement with semicolon. This is only necessary when statements are sent through *stdin*;
- `-t`: print some (server-side) timing information to standard error;
- `-q`: quiet mode: don't print processing information to standard error;
- `-V`: prints version information to standard output;
- `-?` or `-h`: prints a help message to standard error.

7 The Low-Level C Client

7.1 Outline

The low-level C API is not intended for application development. For this purpose high-level C and a C++ APIs exist (please refer to document xxx). This API aims instead to ease the development of client language bindings. The Python, Lua and Go client APIs are built on it.

The API consists of the `libnowdbclient` library, which is installed in a canonical library directory such as `/usr/local/lib`, and the `nowclient.h` interface, which is intalled in a canonical header directory such as `/usr/local/include`.

The API provides services for time conversion, connection management, result handling and error handling.

7.2 Time

The API defines the NOWDB time type, `nowdb_time_t` as `uint64_t`. Values of this type represent a UNIX timestamp with nanosecond precision. The smallest and greatest possible values are defined as

- NOWDB_TIME_DAWN ('1677-09-21T00:12:44') and
- NOWDB_TIME_DUSK ('2262-04-11T23:47:16') respectively.

The standard formats are defined as

- NOWDB_TIME_FORMAT `"%Y-%m-%dT%H:%M:%S"` and
- NOWDB_DATE_FORMAT `"%Y-%m-%d"`.

The following time conversion functions are available:

nowdb_time_fromUnix

The function receives a POSIX `struct timespec` and returns a `nowdb_time_t`. The conversion never fails.

nowdb_time_toUnix

The function receives a `nowdb_time_t` and a pointer to a POSIX `struct timespec`, which must not be NULL. The function returns an `int` representing an error code (see chapter 16). The conversion fails if the pointer to the *timespec* structure is NULL or when the `nowdb_time_t` value is out of range (which can happen only with custom time configurations).

nowdb_time_parse

The function receives

- a time string (which must not be NULL),
- a format string (which must not be NULL) and
- a pointer to a `nowdb_time_t` (which must not be NULL).

The function returns an `int` representing an error code (see chapter 16).

The function attempts to parse the time string according to the format string and, if successful, stores the result at the address passed in as third parameter.

nowdb_time_show

The function receives

- a `nowdb_time_t`
- a format string (which must not be NULL),
- a `char` buffer (which must not be NULL) and
- a `size_t` indicating the size of the buffer. (which must not be NULL).

The function returns an `int` representing an error code (see chapter 16).

The function writes a string representation of the `nowdb_time_t` value according to the format string into the `char` buffer. The function fails (among other reasons), if the buffer is not big enough to hold the result.

nowdb_time_get

The function expects no argument and returns a `nowdb_time_t` representing the current time. On failure, the function returns NOWDB_TIME_DAWN (which is certainly not *now*).

7.3 Connection

The type `nowdb_con_t` represents a TCP/IP connection to the database. The type is defined as pointer to an *anonymous struct* and cannot be allocated by the user.

Note that connections are not *threadsafe*. Threads must not access connection objects concurrently.

For connection management the following services are available:

nowdb_connect

The function receives six arguments:

- a pointer to a `nowdb_con_t` (which must not be NULL),
- a string (which must not be NULL) defining the *node* on which the database is running and which may be an IP `v4` or `v6` address or a hostname.
- a string (which must not be NULL) defining the service which is usually the port to which the database is listening,
- a string (which may be NULL) defining the user (currently not used),
- a string (which may be NULL) defining the password of that user (currently not used) and
- an `int` representing connection flags.

The function returns an `int` representing a client error code (see section 7.6).

The function attempts to connect to the database server. On success (when the return value is OK), the `nowdb_con_t` pointer is guaranteed to point to a valid connection object. Otherwise, no memory is allocated.

The flags represent connection options. Currently, the following options are available:

- NOWDB_FLAGS_TEXT: results for this session will be sent in a textual (CSV) format. Not available
- NOWDB_FLAGS_LE: results for this session will be sent in binary *little endian* format.
- NOWDB_FLAGS_BE: results for this session will be sent in binary *big endian* format. Not available

nowdb_connection_close

The function receives a `nowdb_con_t` and returns an `int` representing a client error code (see section 7.6).

The function terminates the connection. On success, it is guaranteed that all resources have been freed. Otherwise, when an error code is returned, resources are still allocated and must be freed using `nowdb_connection_destroy`.

nowdb_connection_destroy

The function receives a `nowdb_con_t`. The function frees all resources allocated to this connection. The function never fails and is declared as `void`.

7.4 Result

The API implements the four NOWDB dynamic types:

- NOWDB_RESULT_STATUS
- NOWDB_RESULT_REPORT
- NOWDB_RESULT_CURSOR
- NOWDB_RESULT_ROW

Three anonymous structs are defined to represent these types:

- `nowdb_result_status` represents
  - NOWDB_RESULT_STATUS and
  - NOWDB_RESULT_REPORT,

  i.e. the services defined for *status* also accept an object of type *report* and *vice versa*;
- `nowdb_result_cursor` represents NOWDB_RESULT_CURSOR;
- `nowdb_result_row` represents NOWDB_RESULT_ROW

7.4.1 Execution

nowdb_exec_statement

The function receives

- a connection
- a string representing an SQL statement
- a pointer to a result object

None of those must be NULL.

The function returns an `int` representing a client error code.

The function sends the SQL statement to the database and waits for the result. On success, the `nowdb_result_t` pointer is guaranteed to point to a valid result object. Otherwise, no additional memory is allocated.

nowdb_exec_statementZC

This function is a *zerocopy* variant of `nowdb_exec_statement`. `nowdb_exec_statement` copies all data received from the database to a private buffer that belongs to the result object. This allows user programs to process interleaving queries, *i.e.* statements can be executed, while others still have pending results.

In some cases, this is not necessary, in particular when the result is just a status that is checked once immediately after `exec` has returned. In such cases `nowdb_exec_statementZC` might be more efficient, because it does not copy the result data into a private buffer, but leaves them in the connection object. The next query, hence, will overwrite those data.

Note that `nowdb_exec_statementZC` is not allowed when the result may be a cursor.

7.4.2 Status and Report

nowdb_result_errcode

The function receives a `nowdb_result_t` and returns its error code which is a server-side error code (or 0 for success).

nowdb_result_eof

The function receives a `nowdb_result_t` and returns an `int` which is $\neq 0$ if the error code is *end-of-file* and 0 otherwise.

nowdb_result_details

The function receives a `nowdb_result_t` and returns a constant string providing details on the error represented by the status. If no error has occurred, the result string is `"OK"`.

nowdb_result_report

The function is declared as `void`. The function receives a `nowdb_result_t` and three more parameters:

- a pointer to a `uint64_t` which must not be NULL and to which the number of affected rows is written;
- a pointer to a `uint64_t` which must not be NULL and to which the number of errors is written;
- a pointer to a `uint64_t` which must not be NULL and to which the running time in microseconds is written.

nowdb_result_destroy

The function is declared as `void`. The function receives a `nowdb_result_t` and frees all resources assigned to it.

7.4.3 Cursor

nowdb_cursor_open

The function receives a `nowdb_result_t` and a pointer to a `nowdb_cursor_t` (which must not be NULL). It returns an `int` representing a client error code. The function *casts* the result passed in to the pointer address. The function fails if the result does not represent a valid result or if that result was created with *zerocopy* option. Note that `nowdb_exec_statementZC` is not allowed when the result may be a cursor.

nowdb_cursor_errcode

The function receives a `nowdb_cursor_t` and returns its error code.

nowdb_cursor_details

The function receives a `nowdb_cursor_t` and returns a constant string providing error details. If no error has occurred, the result string is `"OK"`.

nowdb_cursor_eof

The function receives a `nowdb_cursor_t` and returns an `int` which is $\neq 0$ if the error code is *end-of-file* and 0 otherwise.

nowdb_cursor_ok

The function receives a `nowdb_cursor_t` and returns an `int` which is $\neq 0$ if the error code is OK and 0 otherwise.

nowdb_cursor_id

The function receives a `nowdb_cursor_t` and returns a `uint64_t` which represents the identifier under which this cursor is known in the server.

nowdb_cursor_fetch

The function receives a `nowdb_cursor_t` and returns an `int` which represents a client error code. On success, the cursor fetches the next bulk of rows from the server. If there are no more rows in the server, the error code of the cursor passes to *end-of-file*.

nowdb_cursor_row

The function receives a `nowdb_cursor_t` and returns a `nowdb_row_t`. Note that the result is not a copy of the rows in the cursor, but a reference to those. It is therefore not necessary to destroy the returned rows. On the other hand, the rows are lost when either `fetch` or `close` is called on the cursor.

nowdb_cursor_close

The function receives a `nowdb_cursor_t` and returns an `int` which represents an error code. It sends a close request for this cursor to the database and, on success, frees all resources assigned to this cursor. Otherwise, if the close request fails, the cursor must be destroyed using `nowdb_result_destroy` and casting the cursor to a `nowdb_result_t`

7.4.4 Row

nowdb_row_next

The function receives a `nowdb_row_t` and returns an `int` that represents a client error code. It advances to the next row. If the current row was already the last one, *end-of-file* is returned.

nowdb_row_rewind

The function is declared as `void`. The function receives a `nowdb_row_t`. It resets the row struct to the first row.

nowdb_row_field

The function receives a `nowdb_row_t` and two more parameters, namely

- an `int`, n, indicating that we want to obtain the nth field starting to count from 0 for the first field in the row;
- a pointer to an `int` which must not be NULL and is set to the type of the field.

The function returns the address of the first byte of the nth field or NULL on error.

Valid types are

- NOWDB_TYP_UINT
- NOWDB_TYP_INT
- NOWDB_TYP_FLOAT
- NOWDB_TYP_TIME
- NOWDB_TYP_TEXT
- NOWDB_TYP_BOOL

nowdb_row_copy

The function receives and returns a `nowdb_row_t` and copies the row passed in allocating new memory. Rows created with copy must be destroyed using `nowdb_result_destroy` casting the row to the generic result type.

nowdb_row_write

The function receives a FILE pointer and a `nowdb_row_t`. It returns an `int` representing a client error code. The function writes a textual representation of the row(s) passed in to the file.

7.5 Error Handling

nowdb_err_explain

The function receives a (client or server) error code and returns a constant string explaining the error.

7.6 Error Codes

Error Name	Numerical Code	Meaning
out of memory	-1	
no connection	-2	
socket error	-3	
error on address	-4	
cannot create result	-5	
invalid parameter	-6	
error on read operation	-101	
error on write operation	-102	
error on open operation	-103	
error on close operation	-104	
use statement failed	-105	
protocol error	-106	
statement or requested resource too big	-107	
operating system error (check errno)	-108	
time or date format error	-109	
cursor with zerocopy requested	-110	
cannot close cursor	-111	

8 Simple Python Client

8.1 Outline

The Python client implements a simple yet powerful and quite fast API to interact with the the NoWDB server. It lives in the module *now.py* and provides the *Connection* and the *Result* class as well as a number of handy support functions.

The module uses the package *dateutil* which must be installed on the system. (Please refer to chapter 11 for details.)

8.2 Connections

The *Connection* class represents a TCP/IP connection to the database. The constructor takes four arguments, which are all strings:

- Address: used to determine the address of the server. The parameter is passed to the system service `getaddrinfo` and allows: an IPv4 address, an IPv6 address or a hostname. Examples: "127.0.0.1", "localhost", "myserver.mydomain.org".
- Port: used to determine the port of the server. The parameter is passed to the system service `getaddrinfo` and allows: a port number or a known service name.
- User: currently not used
- Password: currently not used

The Connection class provides the following methods:

execute(statement)

The method sends the string *statement* to the database and waits for the response. It returns an instance of the *Result* class or, on internal errors, raises one of the exceptions *ClientError* or *DBError*.

Variants

There are two variants of the *execute* method, namely *rexecute* and *rexecute_*. *rexecute* suppresses error results. If the result does not represent an error, it returns the result just like *execute* does. Otherwise, it releases the result and raises an exception instead (the leading 'r' stands for 'raise').

*rexecute_* is intended for situations where the result as such is not interesting, for instance, when issuing a DDL or DML statement. The function always releases the result; if it is an error, *rexecute_* would raise an exception, just like *rexecute*; otherwise, it terminates properly, but does not return anything.

Single Row Results

Many SQL statements return per definition only one row. This is true for statements of the form

```
select count(*), sum(quantity), sum(price) from buys
```

or

```
select now()
```

The *execute* statement (or one of its variants), however, would force the programmer to write some boilerplate code to obtain this single row, *e.g.*:

```
for row in c.rexecute("select count(*), sum(quantity) from buys"):
    cnt = row.field(0) # see below for details
    sm  = row.field(1)
# use cnt and sm
```

For this special use case, there is the method *oneRow*: it receives an SQL statement, executes it and handles the result as a cursor (see below). If it is not a cursor (or when an error occurs), the function will raise an exception. Otherwise, it obtains the first row of this cursor and appends all the fields in this row to a list. It releases the cursor and then returns the list. The above code can thus be simplified to

```
row = c.oneRow("select count(*), sum(quantity) from buys")
# use row[0] and row[1]
```

To go even further: a subset of queries that return only one row return only one single value. For such such cases, there is yet another convenience interface, namely *oneValue*, which is implemented as

```
return self.oneRow[0]
```

close()

The method closes the connection and releases the C objects allocated by the constructor. If the connection cannot be closed for any reason, the exception *ClientError* is raised.

Resource Manager

Connection is a *resource manager*. It is therefore rarely necessary to call *close* directly in user code. Instead, *Connection* can be used with the *with* statement, *i.e.*:

```
with Connection('localhost', '55505', 'myusr', 'mypwd') as conn:
    # here goes your code
    # refer to the connection as 'conn'
```

8.3 Results

The *Result* class represents the dynamic types described in chapter 4. Instances of *Result* are created and returned by the *Connection.execute()* method.

The *Result* class has the following methods:

rType()

The method returns the result type, either STATUS, REPORT, ROW or CURSOR.

ok()

The method returns *True* if the instance does not represent an error and *False* otherwise.

release()

The method releases the C resources allocated together with the *Result* object. It is not strictly necessary to call this method in user code, since the Python GC would invoke it anyway. However, it is a good policy to release resources as soon as they are not needed anymore. Yet again, *Result* is a resource manager and can be used in *with* statements, which call release internally when leaving the block, *e.g.*:

```
with conn.execute("select count(*) from buys") as cur:
    # here goes your code
    # 'conn' is a previously created connection
    # 'cur' is the result and,
    # if no error has occurred, cur is a cursor
```

8.3.1 Status

If *Result* is a *Status*, two more methods are available:

code()

The method returns the NOWDB error code. The error code may be

- 0: Success, no error has occurred;
- positive: An error in the database occurred; chapter 16 provides a list of server-side error codes.
- negative: An error in the client library occurred; chapter 7 provides a list of client-side error codes.

details()

The method returns detailed information on the error (or *None* if no error has occurred).

8.3.2 Cursors

If the result is a cursor, four more methods are available:

fetch()

The method fetches the next bulk of rows from the database. After successful completion, the cursor contains this bulk of rows, which can be obtained by means of the method *row()*. Note that the first bulk of rows is available immediately after the Cursor has been created.

row()

The method obtains the current bulk of rows from the cursor. It returns a *Row* result.

close()

The method closes the cursor; internally, this method is called wherever, for other results, *release* would be called, that is, by the GC and when leaving the *with* statement for which the cursor was created.

eof()

The method returns *True* if the error state of the cursor is *end-of-file* and *False* otherwise.

Iterator

Cursor is an iterator. Usually, *fetch()*, *row()* and *eof()* do not need to be called explicitly. Instead a *for*-loop can be used, *e.g.*:

```
with conn.execute("select * from buys") as cur:
    for row in cur:
        # process row
```

When using a cursor as an iterator it is automatically released immediately before the iteration ends. The above code can therefore be simplified to:

```
for row in conn.execute("select * from buys"):
    # process row
```

8.3.3 Rows

If the result is a row, two more methods are available:

field(n)

The method returns the value of the *n*th field (starting to count from 0 for the first field). The value is an SQL base type converted to Python. Conversion takes place in the obvious way (integer and unsigned integer to int, float to float, text to string, *etc.*). An exception are *time* fields. One could expect *field* to return a *datetime*, but that is not the case. Time is returned as a (signed) integer representing a UNIX timestamp with nanosecond precision. It can be converted to *datetime* with *now2dt* (please refer to section 8.5 for details).

typedField(n)

The method is similar to *field*, but returns additionally the NoWDB type of the *nth* field, *e.g.*:

```
for row in conn.execute("select * from buys"):
    (t,v) = row.typedField(0)
    # t is the type, v is the value
```

count()

The method returns the number of fields in the row, *e.g.*:

```
for row in conn.execute("select * from buys"):
    for i in range(row.count()):
        (t,v) = row.typedField(i)
        print("field %d of type %d: %s" % (i, t, v))
```

nextRow()

The method advances to the next row. If there was one more row, the method returns *True* and the next call to *field(n)* will return the *n*th field of that row. Otherwise, if no more rows are available, the method returns *False*.

Note that, working with a cursor as iterator, it is not necessary to use this method. The iterator already produces single rows.

8.3.4 Reports

If the result is a report, three more methods are available:

affected()

The method returns the number of affected rows.

errors()

The method returns the number of errors. Note that DML statements will return a *Status* result if an error occurred. In practice, only DLL statements provide this information.

runTime()

The method returns the running time of the DML or DLL statement.

A usage example is

```
with conn.execute("load '/opt/import/client.csv into client") as rep:
    if not rep.ok():
        print("ERROR: %s " % rep.details())
        return
    print("affected: %d, errors: %d, running time: %dus" %
            (rep.affected(), rep.errors(), rep.runTime()))
```

8.4 Exceptions and Errors

ClientError

Is raised when an error in the client library occurred (*e.g.* a socket error).

DBError

Is raised when an error in the database occurred.

WrongType

Is raised in case of type mismatch, *i.e.* trying to call a method not available for this specific return type.

explain(err)

The function *explain* returns a description of the error code passed in.

8.5 Support Functions

dt2now(dt)

The function expects a *datetime* object and converts it to a NoWDB timestamp.

now2dt(tp)

The function expects a NoWDB timestamp and converts it to a *datetime* object.

utc

This is a global variable defined in *now.py*. It is the UTC timezone object needed for many datetime constructors and conversion functions. It is provided for convenience.

TIMEFORMAT and DATEFORMAT

These are global constants providing the standard NoWDB time and date format strings that can be used, for instance, in the *datetime strftime* and *strptime* methods, *e.g.*:

```
stmt = "select count(*) from buys \
          where timestamp = '" + tp.strftime(TIMEFORMAT) + "'"
```

9 Python DB API Client

9.1 Outline

The NoWDB DB API implements PEP 249, the Python DB API. Its structure is similar to the simple API, but there are some differences. It is in some cases more convenient to use than the simple API, but, more importantly, it can be used with Python packages that rely on PEP 249, in particular *Pandas*. The main disadvantage of this API is that it is significantly slower than the simple API when it comes to queries with large result sets.

The API lives in the module *nowapi.py* and provides a Connection class, a Cursor class (which differs conceptionally from cursors in the simple API), a rich exception hierarchy and type constructors to address database-specific types, in particular timestamps.

9.2 Connections

Connection is conceptionally equivalent to the Connection class in the simple API and can be used in exactly the same way (*e.g.* as a resource manager). There is a difference in the constructor *connect*, which expects one more argument: a database name. If this argument is given, connect, on success, issues a *use* statement. Example:

```
with connect('localhost', '55505', 'usr', 'pwd', 'retail') as c:
    # here goes your code
    # refer to the connection as 'c'
```

When execution leaves the *with* block, the connection's *close* method is invoked internally, which works exactly as for the simple API.

A peculiarity of the Connection class is the *cursor*() method, which creates and returns a cursor object. The cursor concept is quite different in this API. A cursor is a dedicated means to execute SQL statements. PEP 249 does not foresee an *execute* method as part of the Connection class like the simple API; instead statements are executed through cursors. However, we provide such a method for convenience. Internally, it creates a cursor, executes the statement through this cursor and finally returns it to the caller or, on error, raises an exception.

The *execute* method accepts up to three arguments: the statement (which is mandatory), a list of parameters for this statement (which is optional) and the row format (which, too, is optional).

The parameters argument is expected to be a list of parameters, which will be added to the statement according to usual string formatting in Python. We will need server-side parameter binding in the future. A simple example is:

```
with connect('localhost', '55505', 'usr', 'pwd', 'retail') as c:
    with c.execute("select count(*) from buys \
                    where stamp >= '%s'", ['2018-10-01']) as cur:
      # process cursor here
```

The row format is relevant only for *select* statements and can be either:

- *dictrow*: the rows in the result set will be presented as dictionaries with the *name* of the fields as keys. This is the default row format. The field names are derived from the statement as follows:
  - for a projection clause of the form **select** *, the names of the fields are obtained from the database via the *describe* statement;
  - fields that have an alias (**select** x **as** y) are named after the alias;
  - fields that have no alias, are named as they are selected. For instance, the expression **count**(*) is literally called "count(*)";
- *tuplerow*: the rows are presented as tuples and can be referenced by their index starting, as usual, from 0.
- *listrow*: the rows are presented as lists and can be referenced by their index.

A basic usage example is

```
import nowapi as na
with na.connect('localhost', '55505', 'usr', 'pwd', 'retail') as c:
  # cursor is an iterator
  for row in c.execute("select client_key as k,\
                               client_name as n \
                        from client"):
      print("%d: %s" % (row['k'], row['n']))
```

And an example involving Pandas:

```
import nowapi as na
import pandas as pd
with na.connect('localhost', '55505', 'usr', 'pwd', 'retail') as c:
  sales = pd.read_sql("select origin, stamp, quanity, price \
                       from buys", c)
  # do something with dataframe 'sales'
```

9.3 Cursors

A cursor is a device to send SQL statements to the database. The *execute* method of the Connection class is just a convenience interface that internally creates and uses a cursor to execute a statement. The explicit way to handle statements is to create a cursor by means of the method *cursor*, *e.g.*:

```
import nowapi as na
with na.connect('localhost', '55505', 'usr', 'pwd', 'retail') as c:
  cur = c.cursor()
  # do something with 'cur'
```

Cursors provide the *close* method, which will finally be called by the GC, but, since cursors bind resources on the server, they should be closed as soon as possible. However, cursors are resource managers. They can, thus, be used in a *with* statement which will close the cursor when leaving the block. As we have seen, they are also iterators. When used as iterator, the cursor is closed after the final iteration. There is, hence, rarely the need to call *close* explicitly.

The Cursor class provides methods to execute statements and fetch rows from result sets. The *execute* method takes up to two arguments: the SQL statement and a list of parameters. It does not accept the row format argument like the Connection method. Instead the row format must be set explicitly *before execute* is called. For this purpose Cursor provides the method *setRowFormat* which accepts one argument, namely the row format. If the row format is not set, it defaults to *dictrow*.

The rows can be fetched from the cursor by either *fetchone*, *fetchmany* or *fetchall*. The method *fetchone* returns the next row in the cursor according to the defined row format. *fetchmany* returns a list of the next n rows, where n is passed in as parameter. If *fetchmany* is called without parameter, n defaults to 1. Finally, *fetchall* fetches all rows at once.

As a rationale to provide these methods, the PEP 249 specification refers to performance reasons. It should be noted that, in the NOWDB implementation of PEP 249, *fetchmany* has no performance advantage over *fetchone*. The method does not send a *fetch* to the database for each row. This is already handled in the underlying C library, which always receives a bunch of rows from which single rows are then fetched locally in the client.

Furthermore, since Cursor is an iterator, there is no need to use these methods at all; the common *for* loop implements an optimal row fetching mechanism.

In general, processing cursors with this API is slower than with the simple one. The reason is that PEP 249 forces database clients to transform all rows into Python data types, *viz.* dictionaries or tuples, which is very expensive and, in most cases, unnecessary. Using the values directly to compute the final result is often much more efficient. This leads to the somewhat paradoxical situation that we create expensive Python objects in order to create Pandas structures (like series and dataframes), which we want to use because they are much faster than native Python objects in the first place.

Cursors have two attributes that provide information about the data. The most important is the *description*, a list of tuples of the form (field name, type code). *field name* is the chosen name of the *nth* field and the type code the NoWDB type of the *nth* field where n is the *nth* element in the list. The description is available, when a DQL statement has been executed and *None* otherwise.

The second attribute is the *rowcount*, which is -1 if no statements has been executed. For a DML statement, *rowcount* reflects the number of affected rows; for a DQL statement, *rowcount* shall, according to PEP 249, reflect the number of rows in the result set. Here, the NoWDB implementation deviates from the specification. The *rowcount* is not written immediately after executing the statement. At this time the *rowcount* is not yet known. Instead, the value is incremented as the user goes along fetching rows from the cursor.

9.4 Exceptions

In the DB API, errors are always communicated as exceptions. The specification foresees a rich hierarchy of exceptions, which is completely implemented in the NoWDB client, but not all are actually raised. This section discusses the exceptions that are currently used.

9.4.1 Error

This is the base class for other exceptions and is not raised itself.

9.4.2 InterfaceError

This exception is raised for errors on the level of the API, *e.g.* parameter errors, missing connection, fetching from a cursor that has not yet been executed, *etc.*

9.4.3 DatabaseError

This exception is raised for errors in the database server. The errors may be related to SQL parsing errors, data inconsistency (*e.g.* duplicated key) or non-existing entities (*e.g.* inserting into a non-existing table).

9.4.4 InternalError

This is a "panic" error and indicates a software bug in NOWDB.

9.4.5 NotSupportedError

The requested feature is not available.

9.5 Type Constructors

The PEP 249 specification foresees a number of type conversions to deal with types that are specific for the database in question. The NOWDB client implements the constructors *Date* and *Timestamp*. Both create a UTC *datetime* object that can be used as parameters in the *execute* variants.

Date expects three integer arguments: year, month and day of month; *Timestamp* expects six arguments: year, month, day, hour, minute and second, for instance:

```
import nowapi as na
with na.connect('localhost', '55505', 'usr', 'pwd', 'retail') as c:
  for row in c.execute("select * from buys \
                        where stamp = '%s'", \
                       [Date(2018, 1, 15)]):
    # process rows here
```

10 Embedded Lua

10.1 Outline

The main server interface besides SQL is Lua (version 5.3 or higher). While SQL is intentionally kept simple in NoWDB, it is in fact a mere data access language without control structures and variables, Lua is a full-fledged (but still simple) programming language that can be used to implement complex processing logic. The fact that the Lua environment is hosted on the server implies that the overhead for communicating with the storage engine and getting data in and out is very small compared to the client/server round-trip. Typical use cases for server-side Lua programming are therefore:

- implementing complex queries that require repeated data access or complex joins;
- building algorithms that are beyond pure SQL queries like path finding or recommendation engines;
- data preparation for data science tasks like grouping, statistical sampling or feature extraction;
- background tasks to pre-compute statistics or perform housekeeping and backups and other DBA jobs;
- active server tasks like publish and subscribe or alerts;
- tasks that depend on in-memory state.

The server-side API is similar to its client-side cousins. It provides an interface to execute SQL statements with a number of convenience interfaces for common use cases; a polymorphic result type to communicate SQL results from the database to user code and from user code to the client; services for time manipulation and error handling routines.

The NoWDB Lua environment itself lives in the session. When a client connects (and the Lua support is activated in the NoWDB daemon), the Lua interpreter for this session is initialised and the Lua standard modules as well as the main Lua module are loaded into the session. Further modules, in particular the toplevel modules containing the user code for stored procedures, are loaded when needed, that is, on the first execution of a specific stored procedure.

The NoWDB Lua module creates the Lua table *nowdb* which contains all functions and constants defined by the NoWDB Lua environment. This table is a global variable, so that this functionality is immediately available when the session starts. It is not necessary to import this module explicitly in user code.

In order to find the user-defined Lua code, NoWDB inspects the environment variable `NOWDB_LUA_PATH`. NoWDB expects this variable to contain semicolon-separated definitions of the form

```
db:path
```

where *db* is the name of a database and *path* is an regular POSIX path. NoWDB will then search for a specific module in the path given for the database currently in use (defined by the most recent SQL **use** statement).

For instance, if a procedure in database *mydb* was created as

create procedure *mymodule.myprocedure*() **language** *lua*

NoWDB will search for the key *mydb* in `NOWDB_LUA_PATH` and search in the *path* indicated after the colon for a file called *mymodule.lua*.

It is possible to define a wildcard that matches all databases. The wildcard is $*$, *e.g.*:

```
*:/path/to/lua
```

If such an entry exists in `NOWDB_LUA_PATH`, but not an entry for the current database, then modules will be searched in this path.

The rationale for providing a wildcard is that new databases may be created on the fly in an already running NoWDB daemon with an environment variable that does not yet contain the name of this new database. Toplevel modules for such new databases should then be stored in the wildcard path.

An issue with this approach is that names of toplevel modules in the wildcard path must be unqiue; this is not true for modules of different databases living in different paths. When the session switches from one database to the other, by issuing a new **use** statement, the modules of the old database will be unloaded.

In general, database-specific paths are always searched before the wildcard path and the first module found will be loaded.

10.2 Execute

Most interactions with the NOWDB server are performed by one of the varieties of the *execute* function. The most fundamental is *nowdb.execute*, which receives a string containing an SQL statement, returns a result or raises an error (*i.e.* calls the Lua *error* function), *e.g.*:

```
local cur = nowdb.execute([[select * from product]])
```

In some situations, it might be inconvenient that a function raises an error. For this cases, there is a protected variant called *nowdb.pexecute*. It also expects an SQL statement as input, but returns two return values: an error code and a result or string. If the error code is *nowdb.*OK, the second return value is a polymorphic result type (a status, report, cursor or row). Otherwise, it is a string describing the error with more details, *e.g.*:

```
local ok, cur = nowdb.pexecute([[select * from product]])
if ok ~= nowdb.OK then
   -- error handling
end
-- handle cursor
```

Another variant of *execute* is the function *nowdb.execute_*. This is a convenience interface for situations where the programmer does not want to deal with a result at all. The function returns no result, but raises an error if something goes wrong. typical use cases may be:

```
nowdb.execute_([[create type product(
                  key uint primary key,
                  desc       text,
                  base_price float)]])

nowdb.execute_([[insert into product (key, desc, base_price)
                           values  (1010, 'cool product', 1.49)]])
```

Often queries can only have one result row or even just one value. In such cases, the functions above are not convenient, because they force the programmer to write all the boilerplate code for cursors. There are two functions to handle this kind of situations: *nowdb.onerow* and *nowdb.onevalue*.

nowdb.onerow accepts an SQL statement and returns a Lua array representing one result row, *e.g.*:

```
local r = nowdb.onerow([[select * from product where key = 12345]])
```

where *r* is a Lua array. If no data are available for the specific query, the array is empty.

nowdb.onevalue behaves like *onerow*, but returns only one single value, *e.g.*:

```
local d = nowdb.onevalue([[select desc from product where key = 12345]])
```

where *d* is a single Lua value.

Both functions raise an error when something goes wrong.

One might be tempted to write code like this

```
local p = nowdb.onevalue([[select pi()]])
```

This, however, would fail, since queries that have only a projection clause (*i.e.* no *from* close) do not produce a cursor, but a row (see chapter 4). *onerow* and *onevalue*, however, expect statements that produce a cursor and fail otherwise. The correct way to write the above code is

```
local r = nowdb.execute([[select pi()]])
local p = r.field(0)
r.release()
```

Since this is a lot of boilerplate, there is a function to handle such cases, namely *nowdb.eval* which evaluates a single SQL expression. It expects a string that contains only the expression, *e.g. pi*(), $1 + 2$, *now*(), *etc.* The above code can thus be written as

```
local p = nowdb.eval('pi()')
```

10.3 Results

The *execute* functions return a polymorphic result type (see dynamic types in chapter 4). A result always has the following methods:

- *resulttype*() returns a numeric code indicating the type of this specific result. Result types are:
  - *nowdb.NOTHING*
  - *nowdb.STATUS*
  - *nowdb.REPORT*
  - *nowdb.ROW*
  - *nowdb.CURSOR*
- *ok*() returns a boolean indicating whether the result represents an error condition; if the return value is *true*, the result does not represent an error, otherwise, it does. The programmer can then inspect the error code and the error details method discussed in the following subsection.
- *release*() releases the C resources associated with the result. It is not strictly necessary to release these resources explicitly, since the Lua GC takes care of that. However, it is a good policy to release results as soon as possible; the GC is not aware of the size of the C resources and may decide to postpone the next cycle although there is already a lot of unused memory on the C side.

10.3.1 Status and Error Handling

The simplest result type is the status which represents an error condition. It provides two methods, namely *errcode*() and *errdetails*(). The former returns a numeric error condition (see appendix 16), the latter a string describing the details of the error. Important error codes that are often used to decide on how to proceed in the code are

- *nowdb*.OK: successful completion;
- *nowdb*.EOF: no more data available;
- *nowdb*.NOMEM: the server ran out of memory;
- *nowdb*.TOOBIG: the requested resource is too big;
- *nowdb*.KEYNOF: key not found;
- *nowdb*.DUPKEY: duplicated key;
- *nowdb*.TIMEOUT: a timeout occurred;
- *nowdb*.NOTACUR: a cursor method was invoked on a result, that is not a cursor;
- *nowdb*.NOTAROW a row method was invoked on a result, that is not a row;
- *nowdb*.USRERR: error in user code;
- *nowdb*.SELFLOCK: attempt to acquire a lock that the caller is currently holding;
- *nowdb*.DEADLOCK: deadlock detected;
- *nowdb*.NOTMYLOCK: attempt to release a lock that the caller is currently not holding;

The programmer may decide to return an error result from her function. That is not strictly necessary: if the user-defined function does not return anything, NoWDB will send *nowdb*.OK back to the client. If user code raises an error, NoWDB will convert this error into an error message which is sent back to the client with error code *nowdb*.USRERR.

For explicit error handling the environment provides functions to create status codes, namely:

- *nowdb.success*() creates a status with error code *nowdb*.OK;
- *nowdb.error*(rc, msg) creates a status with error code *rc* and error message *msg*.

The programmer can use the function *nowdb.raise*(rc, msg) to immediately terminate execution. The client will then receive an error with the error code USRERR and a message string that contains the error code *rc* and the message *msg*. In some situations, it may be preferrable to explicitly return an error status, since in that case the calling client receives the intended numerical error code instead of USRERR.

To ease program flow in face of exceptions, there are the function *nowdb.bracket* and *nowdb.pbracket*. The functions take three arguments, which are all functions themselves, called *before*, *after* and *body*. Ignoring details of parameter passing (which in fact is bit more complicated), the implementation of *pbracket* looks like the code snippet:

```
function nowdb.pbracket(before, after, body)
  local r = before()
  local ok, t = pcall(body,r)
  after(r)
  return ok, t
end
```

Typically *before* is a function that obtains a resource and *after* closes this resource. *body* is a function that runs between *before* and *after* using the resource. *bracket* guarantees that *after* is called even when *body* fails. A use case may be:

```
local function before()
  nowdb.execute_("lock mylock")
end
local function after()
  nowdb.execute_("unlock mylock")
end
local k = nowdb.bracket(before, after, function()
  local x = nowdb.onevalue([[select max(key) from mytable]])
  nowdb.execute_(string.format(
        [[insert into mytable (key) values (%d)]], x+1))
  return x
end)
```

In this code fragment, *bracket* is used to obtain and increment a unique key from a table. The resource in this case is a lock held in the database and just represented by the state *nil*, *i.e. before* does not return a value and *after* and *body* do not expect values. The point is, however, that the lock is released independent of *body* raising an exception or not.

The difference between *bracket* and *pbracket* is that *pbracket* returns all return values from *pcall* including the leading Boolean (*ok*); *bracket*, by contrast, removes the Boolean and reraises the error in case the Boolean was *false*

10.3.2 Reports

tbc

10.3.3 Rows

Row results represent single lines of a result set. They appear naturally with cursors which provide methods to iterate through the rows of a result set and this is basically the only way where they appear on client side. On server-side there are two other occasions where rows appear: 1) a stored procedure may return a row (or bunch of rows). On client side, this is not the case, because NoWDB transforms rows into cursors before they are sent to the client. On the server, however, rows are returned in their "raw" form. 2) Projections without a **from** clause.

A row contains one or more typed values. There are two methods that provide access to these values: *field* and *typedfield*.

Both take one argument, namely the index of the field starting to count from 0. the difference is that *field* returns only the value at that position in the row, while *typedfield* returns a NoWDB type code together with the field. The NoWDB type codes are:

- *nowdb*.NOTHING
- *nowdb*.TEXT
- *nowdb*.DATE
- *nowdb*.TIME
- *nowdb*.FLOAT
- *nowdb*.INT
- *nowdb*.UINT
- *nowdb*.BOOL

Here is a simple usage example:

```
local row = nowdb.execute("select pi(), e()")
print(string.format(
  [[pi is             : %.8f
    Euler's number is: %.8f]],
  row.field(0), row.field(1)))
row.release()
```

The method *countfields*() returns the number of fields in a row. Here is a usage example that also illustrates the use of *typedfield*:

```
for i in 0, row.countfields() do
  local t, v = row.typedfield(i)
  print(string.format(
    "field %d is of type %d", i, t))
end
```

There are two helper functions to convert type codes into type strings and vice versa, namely *nowdb.nowtypename*, which expects a type code and returns a string (*e.g. nowdb.* FLOAT is converted to 'float') and *nowdb.nowtypebyname*, which expects a string and, if this string is recognised as the name of a type, returns the corresponding type code; valid type strings are 'text', 'date', 'time', 'uint', 'uinteger', 'int', 'integer', 'float', 'bool', 'boolean', 'null' and 'nil'.

Rows are also an important means to return results back to the calling client. A row is created with the function *nowdb.makerow*, which takes no arguments and returns an empty row.

The method *add2row* can be used to add a value to the row. The method takes two arguments: the first is the NoWDB type code of the value and the second is the value itself. A row is closed (*i.e.* an *end-of-row* marker is inserted) with the method *closerow*, which takes no arguments and returns nothing.

The following code snippet is an example:

```
function getconstants()
  local row = nowdb.makerow()
  row.add2row(nowdb.FLOAT, 2.718281828)
  row.add2row(nowdb.FLOAT, 3.141592653)
  row.closerow()
  return row
end
```

A very common use case is to create a row from an array. There is a special function to do this: *nowdb.array2row*, which takes two arguments and returns a row.

The first argument is an array of NoWDB types, the second argument is an array of values. The two arrays shall have the same number of elements, otherwise, an error is raised. The function creates a row and adds the values with the matching type in the first array, *e.g.*:

```
function getconstants()
  local r = {2.718281828, 3.141592653}
  local t = {nowdb.FLOAT, nowdb.FLOAT}
  return nowdb.array2row(t, r)
end
```

The inverse of *array2row* is the row method *row2array*, which receives a row and returns two arrays: the first containing the values, the second containing the types. The following code example (which is somewhat pointless) illustrates this behaviour:

```
local row = getconstants() -- constants in row
local vs, ts = row.row2array() -- row to arrays
return nowdb.array2row(ts, vs) -- arrays back to row
```

Another convenience interface is *nowdb.makeresult*, which creates a row from a single value. The function takes two arguments, the type and the value itself and returns a row containing this value as its only field. A very common use case for this function is to communicate a single value back to the calling client. Here is a, perhaps, somewhat artificial example:

```
function integral(ts, fld, k, t0, t1)
  local stmt = string.format(
    [[select stamp/%d, %s from %s
       where origin = %d
         and stamp >= %d and stamp < %d
       order by stamp
    ]], nowdb.hour, fld, ts, k, t0, t1)
  local cur = nowdb.execute(stmt)
  local x_1 = 0
  local x_2 = 0
  local x = 0
  local first = true
  for row in cur.rows() do
      if not first then x_1 = x_2 end
      x_2 = row.field(0)
      if not first then
         local d = x_2 - x_1
         local y = row.field(1)
         x = x + d*y
      else
         first = false
      end
  end
  cur.release()
  return nowdb.makeresult(nowdb.FLOAT, x)
end
```

The function computes an approximation to the area under the curve formed by the stamped edge 'ts' with origin k and timestamp as the x-value and field 'fld' as the y-value.

10.3.4 Cursors

Cursors are iterators over data in the database, either a vertex type or an edge. A successful execution of a **select** statement almost always returns a cursor; the only exception being projections without a **from** statement.

The main functionality provided by cursors is the *rows* method, an iterator factory for the rows in the result set of the statement. The basic pattern is

```
local cur = nowdb.execute(stmt)
for row in cur.rows()
    -- do something
end
cur.release()
```

A more elaborate example is the *integral* function in the previous section.

Internally, *rows* uses the methods *open*, which starts the iteration, *fetch*, which obtains the next bunch of rows from the storage engine and *nextrow*, which advances to the next row within the current bunch of rows. For user code, it is rarely necessary to use these methods directly, but it may be useful in some situations.

The simple pattern above can be built from these building blocks in the following way:

```
local cur = nowdb.execute(stmt)
local rc, msg = cur.open()
if rc ~= nowdb.OK then nowdb.raise(rc, msg) end
rc, msg = cur.fetch()
if rc ~= nowdb.OK then
   nowdb.raise(rc, msg)
end
while true do
  -- do something
  -- use row methods to access data
  local rc = cur.nextrow()
  if rc ~= nowdb.OK then
     local rc, msg = cur.fetch()
     if rc == nowdb.EOF then break end
     if rc ~= nowdb.OK then
        nowdb.raise(rc, msg)
     end
  end
end
cur.release()
```

open and *fetch* return an error code and, if this error code is not *nowdb*.OK, an error message. The methods have effect on the internal state of the cursor, but do not return the state directly. In fact, the *row* methods like *field* and *typedfield* can be called directly on the cursor.

Data are immediately available, when the cursor has been opened and *fetch* has been called on it. The internal offset which points to the current row within the bunch of rows now held by the cursor is incremented with the method *nextrow*. The method returns either *nowdb*.EOF, *nowdb*.NOTACUR or *nowdb*.OK. Details are not available for this method.

The logic, hence, is:

1. open
2. fetch
3. process the data
4. nextrow until EOF
5. if EOF then fetch
6. if EOF again, terminate the process
7. otherwise, go to step 3

Contrary to rows, cursors cannot be created "out of thin air"; cursors can only be created as results of a **select** statement. But they can be returned to the calling client. Here is a simple difference engine that illustrates this technique:

```
function differences(ts, fld, k, t0, t1)
  local stmt = string.format(
    [[select stamp/%d, %s from %s
       where origin = %d
         and stamp >= %d and stamp < %d
       order by stamp
       ]], nowdb.hour, fld, ts, k, t0, t1)
  local y_1 = 0
  local y_2 = 0
  local first = true
  local cur = nowdb.execute(stmt)
  for row in cur.rows() do
      if not first then
         y_1 = y_2
      end
      y_2 = row.field(1)
      if not first then
         local d = y_2 - y_1
         nowdb.execute_(string.format(
           [[insert into babbage (origin, destin, stamp, diff)
                          values (%d, 0, %d, %f)
           ]], k, row.field(0), d))
      else
         first = false
      end
  end
  return nowdb.execute(string.format(
         [[select * from babbage
            where origin = %d]], k))
end
```

It may not be satisfactory in many cases that we have to store the results in another table, before we can pass them back to the client. Instead, we would like to emulate a cursor, so that the client code can iterate over the results generated by the server as if it was a single cursor.

In such cases, we use the original cursor to maintain state on server-side, while each fetch from the client would trigger additional computations over the data produced by the cursor. This behaviour is implemented in the NoWDB Lua support package (see section 10.5). But it can also be built directly from the NoWDB module. The main building block is coroutines and, indeed, there is a convenience interface for *execute* that we have not yet mentioned, namely *nowdb.cocursor*. The function accepts an SQL statement and returns a coroutine that yields a single row on each *resume*.

We could iterate over all rows like this:

```
local co = nowdb.cocursor(stmt)
while true do
  if coroutine.status(co) == 'dead' then break end
  ok, row = coroutine.resume(co)
  if not ok then nowdb.raise(nowdb.USRERR, row) end
  -- do something with row
end
```

There is a function that does exactly this: *nowdb.corows*. This function returns an iterator, so that we can simplify to:

```
local co = nowdb.cocursor(stmt)
for row in nowdb.corows(co) do
  -- do something with row
end
```

Assuming that the producer as well as some of the other variables are held in module-wide variables, we could implement a *fetch* for the difference engine that works roughly like this:

```
function fetchbabbage()
  local y_1 = 0
  local res = nowdb.makerow()
  for i = 1,100 do
    if coroutine.status(co) == 'dead' then break end
    ok, row = coroutine.resume(co)
    if not ok then nowdb.raise(nowdb.USRERR, row) end
    if not first then
      y_1 = y_2
    end
    y_2 = row.field(1)
    if not first then
      local d = y_2 - y_1
      res.add2row(nowdb.UINT, k)
      res.add2row(nowdb.TIME, row.field(0))
      res.add2row(nowdb.FLOAT, d)
      res.closerow()
    end
  end
  return res
end
```

The function creates a bunch of up to hundred rows each time it is called and sends them to the client. The rows are computed in the same way as in the previous implementation of the difference engine; the variables y_2, $first$ and the coroutine *co* are stored in a module-wide variable. Of course, in production code, that state would be stored in a Lua table with some kind of key to identify the query currently processed. But that is not essential for the point here: that we can emulate cursors using global or module-wide state.

10.4 Time

The NoWDB Lua API provides services to convert data from NoWDB format to the Lua time format (which is also used in the *os* module) and back to the NoWDB format. The conversion functions are called *nowdb.from* and *nowdb.to*;

nowdb.from expects a NoWDB timestamp, *i.e.* an integer, and returns a Lua table with the fields

- year: the year as 4-digit number;
- month: the month (1–12);
- day: the day of th month (1–31);
- wday: the day of the week starting at sunday with 1;
- yday: the day of the year starting with January, 1, as 1;
- hour: the hour of the day (0–23);
- min: the minute of the hour (0–59);
- sec: the second of the minute (0–60);
- nano: the nanosecond of the second (0–999999999).

The opposite conversion, performed by *nowdb.to*, expects such a table and returns a NoWDB timestamp. Note that the table must have all fields (except *wday*, *yday* and *nano*), otherwise, an error will be raised.

Lua time tables can be created on the fly by *nowdb.time* or *nowdb.date*. *nowdb.time* expects up to seven parameters: *year*, *month*, *day*, *hour*, *minute*, *second* and *nano*, but only the first three are mandatory. The *nowdb.date* function is just a wrapper around *nowdb.time* and equivalent to *nowdb.time* with only the first three parameters.

Lua time tables can also be transformed into time or date strings using one of the functions *nowdb.timeformat* or *nowdb.dateformat*. The first function converts the timestamp into a canonical time string and the latter converts it into a canonical date string. *time-*

format expects all fields in the table to be present (with the same exception as *nowdb.to*); for *dateformat* only *year*, *month* and *day* are mandatory.

The function *nowdb.getnow*() returns the current system time in UTC as NOWDB timestamp.

NOWDB timestamps can be rounded by means of *nowdb.round*, which expects a NOWDB timestamp and a unit (which is an unsigned integer) and returns a new NOWDB timestamp. Time units are *nowdb.second*, *nowdb.minute*, *nowdb.hour*, *nowdb.day* and *nowdb.year*. Care should be taken with larger units; This function is not aware of leap years and seconds and other time subtleties. If precision is needed, SQL should be used to perform time arithmetic.

A special function is *nowdb.sleep*. This is not a time function in the strict sense; it is more about a thread execution function. It receives one parameter, a NOWDB time value, and suspends execution for *at least* the number of nanoseconds given in this parameter. For how much time exactly execution is suspended depends on the underlying system.

In the following example, we whould suspend execution for five seconds:

```
nowdb.sleep(5*nowdb.second)
```

The function does not return any result and calls *nowdb.raise* on error.

10.5 Lua Support Modules

The NoWDB Lua Support Package is a collection of useful modules to extend the server-side scripting capabilities and to ease common tasks. The collection contains currently:

- The IPC Module: a collection of inter-session communication means, in particular
  - locking
  - events
  - queues
- Unique Identifiers: services to obtain and manage unique identifiers;
- The Result Cache Library: services for result caching with various expiration strategies;
- Virtual Cursors: a module to ease stateful computation and, in particular, cursor-like processing (request-and-fetch). not yet...
- Numnow: Is a Lua implementation of NumPy based on the open source project Lunum. not yet...
- Stats not yet...
- The Sampling Library not yet...
- Feature Extraction not yet...

11 Detailed Installation Guide

tbd

12 Defining Data Loaders

tbd

13 Publish and Subscribe

tbd

14 Optimising Queries

14.1 Terminology

14.2 Fullscan

14.3 Searching

Index vs. no index, role of periods

14.4 Grouping and Ordering

Explain: FRANGE, KRANGE and CRANGE, grouping and ordering without indices

14.5 Joining

15 Storage Sizing

tbc

16 Error Codes

Error Name	Numerical Code	Meaning
out of memory	1	
invalid parameter or value	2	
no resource available	3	
resource is busy	4	
requested resource too big	5	
locking error	6	
unlock error	7	
end of file	8	
feature not supported	9	
bad path	10	
bad name	11	
error in map operation	12	
error in unmap operation	13	
error in read operation	14	
error in write operation	15	
error in open operation	16	
error in close operation	17	
error in remove operation	18	
error in seek operation	19	
internal error (panic)	20	
error reading data catalog	21	
error in time operation	22	
key not found	26	

Error Name	Numerical Code	Meaning
duplicated key	27	
duplicated name	28	
error in sync operation	30	
error in pthread operation	31	
error in sleep operation	32	
error in dequeue operation	33	
error in enqueue operation	34	
worker thread failed	35	
timeout	36	
bad block	38	
bad filesize	39	
cannot set max files	40	
error in move operation	41	
index-related error	42	
wrong or unknown version	43	
error in compression	44	
error in decompression	45	
error in compression dictonary	46	
error in data store	47	
error on table level	48	
error on database level	49	
error in stat operation	50	
error in create operation	51	
error in drop operation	52	
wrong or unknown magic number	53	
error in loader	54	
error in trunc operation	55	
error in flush operation	56	

Error Name	Numerical Code	Meaning
error in beet library operation	57	
error in aggregate function	58	
resource not found	59	
parser error	60	
error waiting on signal	61	
error in signal operation	62	
error setting signal set	63	
protocol error	64	
cannot create socket	65	
error in bind operation	66	
error in listen operation	67	
cannot accept	68	
server error	69	
cannot find or use address	70	
python interpreter error	71	
unknown symbol	72	
user error	73	
unknown error	9999	