TickDB Server specifications

David Bellot

09/12/16

1 Introduction

The current implementation of the Tick Database relies on a set of Python, R and bash scripts and a simple C++ library. Their operations are scheduled with cron on a daily basis. The current version of this database is wobbly and prone to errors because these scripts are only weakly coupled and the querying is not aligned to data production and update. Therefore it is for the time being impossible to query the database and be sure the data are really up to date.

The current state of the Tick DB is given in `http://confluence.liquid-capital.liquidcap.com/display/DEV/TickDB+2.0` with details on its implementation and functionality. It has two parts: one made of a collection of scripts to update and maintain the database, and the other one is a C++ library, currently embedded in the Alpha Framework version 1.x.

The next step is to group all the data maintenance, update and querying into a single application, written in C++, which will take care a updating and delivering data on demand. New functionalities are needed too, like network access, sampling data, signaling clients when specific products are updated, logging and monitoring of the state of the database. More functionalities will be added in the future, using this new server, like a GUI or a web access.

Therefore what we want to achieve in the new version is:

1. reliability of the update mechanism with synchronization of operations, resiliency to crash and network failures and logging,
2. speed-up the delivery of data and provide an easy access to C++, R and Python (this 3 languages are mandatory),
3. enhanced queries and sampling of the data

2 Main functions of the server

We need to implement the main three functions in the server. The details are given in the following sections. The main functions are:

1. Data Update
   (a) monitor incoming files coming from all over the world at irregular intervals,
   (b) update the tick data storage with the new files,

    (c) update in memory and on disk the list of available instruments,
    (d) update the instruments database by merging data from the reference files sent by the exchange, Reactor and possible Bloomberg,
    (e) provide statistics and logs regarding all the above operations,

2. Data Query
    (a) query tick data on one or many instruments and make the data available to another process as fast as possible, knowing the volume of data can exceed several gigabytes,
        i. tick data from the same exchange,
        ii. tick data from multiple exchange,
        iii. reduced tick data for a specific range of strikes for options
    (b) query sampled data and create a cache of the sampling if the data don't exist yet,
    (c) query instruments
        i. basic queries about the content of the instruments database
        ii. statistics about instruments

3. Data Analysis
    (a) validate each data file according to a set of rules (run a R script) and store data for inspection,
    (b) compute daily statistics and generate daily reports,

The three main functions need software to be developed (or adapted from the existing code). They have to be implemented in the order of this list.

3 Use cases and users

We distinguish two types of users and both will have different needs:

1. Simulation, alpha generation:
    (a) queries are on one exchange only, ticks from each instrument need to be interleaved by receive time stamps to replay the data,
    (b) the volume of data is huge and needs to be transferred as fast as possible to the client application,

2. Low frequency, market analysis:
    (a) queries can be on several exchanges, data may need to be synchronized on the same time zone,
    (b) the volume of data is low but the queries are likely to be done remotely

4 Functional Specifications

In this section we describe the main functionalities of the application and give directions on how to implement them and what are the expected results. The list of functions is the one briefly described in section 2.

All along this section, we will assume the application always have access to a single JSON configuration file. An example is given in the appendix. The JSON configuration file format can be modified if needed to suit the needs of the new version of this database.

Because each section corresponds to a different and independent task, it seems possible to implement each of them into a separate thread to allow the database to update at any time and the users to query at any time. Even if it seems a bit drastic, one has to bear in mind that most of the users will be program for automated trading strategies and they can query at any rate and any time of the day.

4.1 Data Update

Monitor incoming files coming from all over the world at irregular intervals At any time during a 24 hours period, files can copied into the *raw data* repository (so far located on the quant server in /mnt/data/raw-data). One directory per day is created and it contains **at least** one file per instrument per 24 hours period. The file name are formatted as: `<Appliance name>-<symbol>-<type>-[<expiry month>-]<creation time>.csv.xz`

It is up to the writer to compress them so they will always arrive compressed with xz. The format of the first field is unfortunately not fixed, but it is possible to parse a file name backward and extract all the necessary information without ambiguity. A token decomposition should be sufficient, otherwise I advise to use `Boost::Spirit`. The format decomposes as follow:

- `<Appliance name>`: the name of the Reactor's appliance which wrote the file. The format is somehow free and
- <symbol>: the Reactor's symbol (which is usually a copy of the exchange symbol). It can be a string of numbers and capitalized letters as `[A-Z0-9]+`
- `<type>`: a single letter defining the product. It can be F (futures), O options, E (equity), S (strategy), C (currency).
- `[<expiry month>-]`: the expiry month if applicable (futures and options only). It uses the Reactor format with `<3 letters month><year>` where `<3 letters month>:=JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC`
- `<creation time>`: creation time of the file in format `<HH><MM><SS>`. This time is very important because for each instrument, **we can have several files per day and they need to be concatenated in order to obtain a proper TickDB file.**

At any time, using *inotify*, the Linux kernel can warn a listening applications to a change in the file system. The server will receive such notifications and will start monitoring each incoming files. As soon as such a file has been closed by the remote writer, the server will either keep it for further use or will start generating the final file into the database. Indeed, because each instrument can have several files, it is impossible to know in advance how many of those files will arrive. Moreover, the order of arrival is not known either. However, they arrive in batch. So whenever a new file arrive, I would wait 15 minutes to be sure the transfer is done for all the files and start processing them. A better solution can be designed here.

Update the tick data storage with the new files The current database sorts files by month. The sorting order can be different and we can organize them by day or by exchange if needed. The file name has a format similar to the original files: `<symbol>-<type>-[<expiry month>-]<creation time>.csv.<extension>`

The definition are the same as in the previous paragraph. The *extension* is bz2 (bzip2) so far, but we aim at adding more compression scheme if it can improve the speed. We can also remove compression for the same reason. The server will be responsible to deliver the data to all the applications, so the compression scheme is local to the server now (unlike the current implementation).

The server will take the list of *raw* files from the previous step and concatenate them in the order of their time stamp. Then it will write the file to its final destination.

Update in-memory and on disk the list of available instruments The database should keep a list of available instruments, identified by their symbol, expiry, etc... and linked to a unique file. I propose we use a SQLite database here to simplify the queries and update. Each time a file has been written and closed as described in the previous paragraph, this small database file shall be updated. If a query comes in, then the database will be able to serve it right away, even if other files are still being processed. The choice of SQLite is because it will allow us to design any type of instruments and products queries and still keep a simple and editable way of storing data.

Update the instruments database by merging data from the reference files sent by the exchange, Reactor and possible Bloomberg Everyday, the various data capture appliance will also send several *reference data* files. These are CSV files and they are highly redundant on a daily basis because the exchange keeps sending everyday the list of all existing contracts. Every time the Linux *inotify* will send notification of a new *reference data* file, the server will update its *SQLite* database with the new data. If possible, the server will query Reactor and Bloomberg to get extra information, like tick size, first notice day, etc...

Provide statistics and logs regarding all the above operations Logs must be provided at each level, with accurate time stamps. No specific format is required here. We recommend using `Boost::Log` (to avoid more dependencies) but any other library will work equally.

4.2 Data Query

The Data Query function will take several forms and will imply writing operations from time to time. We make the difference between tick data and sampled data. In the first case, the main problem will be to read and deliver huge amount of data as fast as possible to the client. In the second case, the problem will be to produce data on-demand and keep them in an extra data storage. The format of the sampled data database can be CVS files, SQLite or InfluxDB. The sampled data are shorter time series than the tick data. The sampling scheme is essentially time based so the interval between each time stamps is fixed, thus making the structure very regular.

The Data Query function is responsible of:

1. answering the user's queries,
2. converting time stamps to the time zone of the query. Timestamps are all in UTC, but the user can ask data in local time of the exchange,
3. delivering the data by the most appropriate channel, either a shared memory segment for tick data or using a socket for the sampled data.
4. answering product queries

The queries will be made through an API by sending json formatted queries to the Tick DB server. Answers will be formatted in JSON for products and sampled data, and as `Boost::uBLAS` matrices for shared memory segment.

Query tick data on one or many instruments and make the data available to another process as fast as possible The user asks for a sequence of days on a set of instruments. The query can be done by instrument or by product and in the second case, for futures and options, the contracts will be rolled based on the user's specification. Roll can be done on the expiry date or any other predefined date. An offset can be subtracted from the date. Roll can also be done on the volume.

If the data comes from different exchange, a latency model will be applied to replay the time series as accurately as possible. If the user asks for options, the data set can be reduced to a range of strikes to avoid sending too big a data set.

The result is a linked list of objects, each one containing the times series of order books of all the instruments, but an additional time series mixing the data. The code exists already and is in production. The developer will be able to re-use this code.

When the results is ready, it is made available to the client through a segment of shared memory.

query sampled data and create a cache of the sampling if the data don't exist yet This is a complex task and will need to be done carefully. A user can ask for a time series of sampled data. The tick data are therefore extracted, and sampled at regular interval of time. The interval is left to the user. Usually, the user will ask for a very long period of time, over several months or years. This is a very CPU intensive task and a very I/O intensive task, especially for options. In certain cases, it will be required to load dozens of gigabytes of data, read and parse them. Therefore, each query's result shall be stored in a new data storage, and every time a query is done, the server will first check if the data have been sampled already and will use this data first, before reading the tick data storage to produce the new data. The more this database will be used, the faster it will become.

A preliminary study should be done to know which database engine should be used to store the sampled data. We propose either flat CSV files, SQLite again or Influx DB. Depending on the load and the user's appetite for such a database, the last option could be interesting. Given the structured form of the data, the result can be delivered in a CSV file (or a CSV string over a socket).

Query instruments database The instrument database is in SQL format so any SQL query compatible with the SQLite engine can be transmitted to the server. Moreover, the Tick DB server will run daily statistics and validation and the user can ask for such statistics on an instrument or product basis. A preliminary version of the product already exists (and is called the *dashboard*).

Recurrent queries can also be stored in the database engine for a client to know when new data on a specific product are available. This will be used by strategies which need to regularly update their parameters.

4.3 Data Analysis

Validate each data file according to a set of rules (run a R script) and store data for inspection The Tick DB server is responsible to run on every new file inserted in the Tick DB a set of R scripts which will analyze the data and validate them or detect errors. Each script will return a result to the server and the server will maintain a list of valid files that it can use for answering the data queries.

Compute daily statistics and generate daily reports On top of the required data validation, the Tick DB Server will also run a set of R scripts which compute statistics and various analysis about volume, latency, volatility, liquidity, etc... The result will be stored in a flexible way, so that one can add more statistics if needed. The server will answer queries to this statistics and deliver the results by the usual way (socket, JSON).

5 Queries

5.1 Tick data queries

The following queries apply to instrument. For futures contracts, an instrument is defined by its underlying product and expiry month. It is unique from both the exchange and the TickDB point of view. For options contracts, an instrument is defined by its underlying product and expiry month. It is unique from the TickDB point of view but not from the exchange point of view because the exchange makes the difference between each strike, puts and calls.

1. all the ticks from date d_1 to date d_2 for N instruments, $N \geq 1$
2. all the ticks from date d_1 over x days for N instruments, $N \geq 1$

3. all the ticks from date d_1 over x business days$N$ instruments, $N \geq 1$. Business days can be computed with QuantLib
4. all the previous queries for options restricted to a strike range $[S_1, S_2]$, to a strikes list $\{S_1, \dots, S_K\}$and to *call* or *put*

The following queries applies equivalently to products, that is we take into account the rolling of the contracts at defined dates.

1. all the ticks from date d_1 to date d_2 for N products, $N \geq 1$
2. all the ticks from date d_1 over x days for N products, $N \geq 1$
3. all the ticks from date d_1 over x business days$N$ products, $N \geq 1$. Business days can be computed with QuantLib
4. all the previous queries for options restricted to a strike range $[S_1, S_2]$, to a strikes list $\{S_1, \dots, S_K\}$and to *call* or *put*

5.2 Rolling out a contract

A contract can be rolled out at a specific date given by a rule. Most of the exchanges will send the expiry date of each instrument but in many cases, we are interested in other types of date, like the first notice date for commodities. The queries for rolling out contracts are as follows:

1. $roll\,date = expiry\,date - x\,days$
2. $roll\,date = expiry\,date - x\,business\,days$, where business days can be obtained with QuantLib
3. every other date available in the instrument database (like first notice day), can be used to roll out a contract

Because each tick data is associated with a UID, the result will automatically switch to a new contract in the result. The TickDB will therefore only need to make sure that the right contract is used on the right day.

5.3 Instruments queries

The SQL structure of the instruments database is to be made public, or at least a view on it. Therefore, any SQL query is acceptable. It is up to the user to define its own query and send an SQL string to the Tick DB Server.

5.4 Sampled data queries

Queries can be made on instruments or products with the same format and restrictions as the tick data queries. The rolling of the contracts will be done exactly the same way. However, the user can specify an sampling interval in the unit they want, be it in the set of seconds, minutes, hours or days, or any combination of them. `Boost::DateTime` shall be used to solve interval problems.

The results can either be presented in the same data structure as the tick data or sent in a CSV format (by file or socket).

6 Extension of the file format

In its current form, the file format and the capture assume we capture data locally to each exchange. We can capture data in one exchange from another exchange, thus taking into account the latency between the 2 exchanges. Moreover, the timestamps are in UTC non daylight-saving. We want to use atomic clock so a extra fields are needed to store time. The current format is as follows:

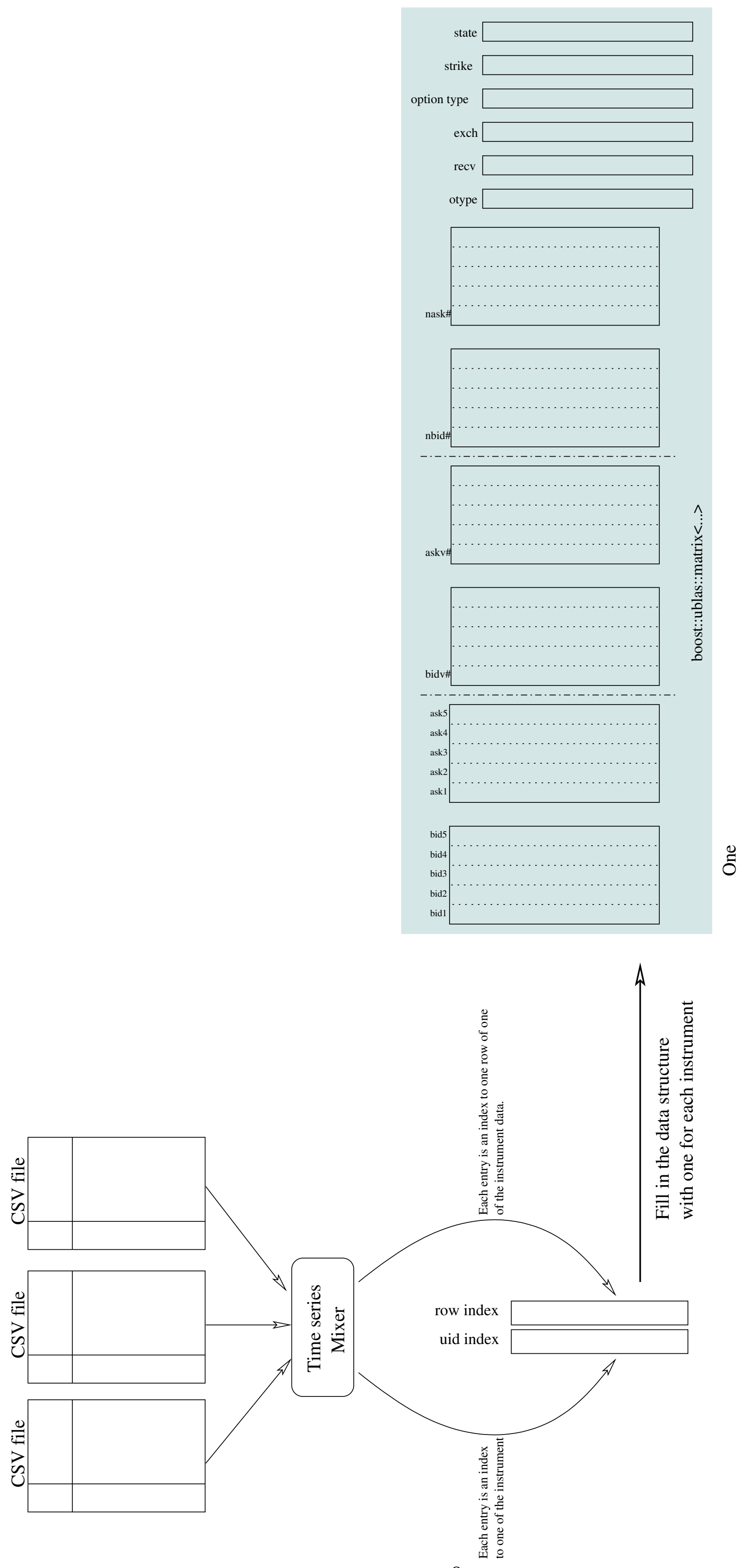
Name	**type**	**description**
otype	*character*	type of the tick data
recv	64 bits unsigned integer	receive time stamp as recorded by our clock
exch	64 bits unsigned integer	exchange time stamp given by the exchange
bid_i	floating-point	5 levels of bid prices with $i = 1, \ldots, 5$
$bidv_i$	integer	5 levels of bid volumes with $i = 1, \ldots, 5$
$nbid_i$	integer	5 levels of number of orders on the bid with $i = 1, \ldots, 5$
ask_i	floating-point	5 levels of ask prices with $i = 1, \ldots, 5$
$askv_i$	integer	5 levels of ask volumes with $i = 1, \ldots, 5$
$nask_i$	integer	5 levels of number of orders on the ask with $i = 1, \ldots, 5$
product	string	product name with strike and type for options
state	integer	state of the product (open, close, etc...)

In the future file format, we will need extra fields for:

- the atomic clock (to be specified)
- the receive timestamp in the colocation and the remote receive timestamps (if applicable)

Moreove, the file names can be extended to include both the exchange name and recording location. For example: CME-EUREX-CL-F-DEC2016-20161212.csv.bz2 for CME crude oil captured in EUREX (Frankfurt).

7 Data structure for tick data



state
strike
option type
exch
recv
otype
nask#
nbid#
askv#
bidv#
ask5
ask4
ask3
ask2
ask1
bid5
bid4
bid3
bid2
bid1
boost::ublas::matrix<...>
One
CSV file
CSV file
CSV file
Time series Mixer
row index
uid index
Each entry is an index to one row of one of the instrument data.
Each entry is an index to one of the instrument
Fill in the data structure with one for each instrument

Each CSV file is read and its content stored in a data structure similar to the one in the blue box. Each box (with dashed lines) is a matrix (as in boost::ublas::matrix<>) and has an inner type set according to the type of data it stores (`price_type` for prices, `volume_type` for volumes, etc...). After reading the data we have as many *blue boxes* as we have instruments, which corresponds to the number of CSV files which have been read too. The row index and uid index will therefore build an history of all the data from all the *blue boxes* into one single history, thus making a replay of the market possible.

8 Deliveries

The implementation will be done in the following order:

1. TickDB Server (1 day)
   (a) choice of the data format for each type of stored data
   (b) choice of communication means (shared memory, socket, pipe, etc...)
   (c) main architecture of the server taking into account the threads, logs, configuration files, etc...
2. Data format for the query and answer (0.5 day)
   (a) data structure and C++ classes for the tick data
   (b) json format for the sampled data
   (c) json format and CSV files for the instruments and statistics
3. Data Update (1 week)
   (a) Architecture of the Data Update
   (b) C++ Implementation of the Data Update
   (c) Regression tests against the current version of the database
4. Data Query (1 week)
   (a) Determination of the scope of possible queries for tick data
   (b) Architecture of the Data Query. It is advised to re-use part of the current library
   (c) C++ Implementation within the server
   (d) C++ Implementation of the API for clients
   (e) Implementation of a test client in C++
5. Data Analysis (3 days)
   (a) Determination of the scope of possible queries
   (b) Architecture
   (c) Implementation in C++
   (d) C++ Implementation of the API for clients

   (e) Implementation of a test client in C++

6. Data Query for sampled data (1.5 week)
   (a) Format of the stored data
   (b) Determination of the scope of queries
   (c) Implementation of a Sampling Library in C++ based on the queries
   (d) Implementation of the caching library
   (e) Integration into the server
   (f) Implementation of a C++ test client

9 Dependencies

- Tools:
  - C++
  - CMake
  - R
  - InfluxDB (maybe)
- Libraries:
  - Boost C++
  - SQLite
  - json C++ library
  - QuantLib

Appendix

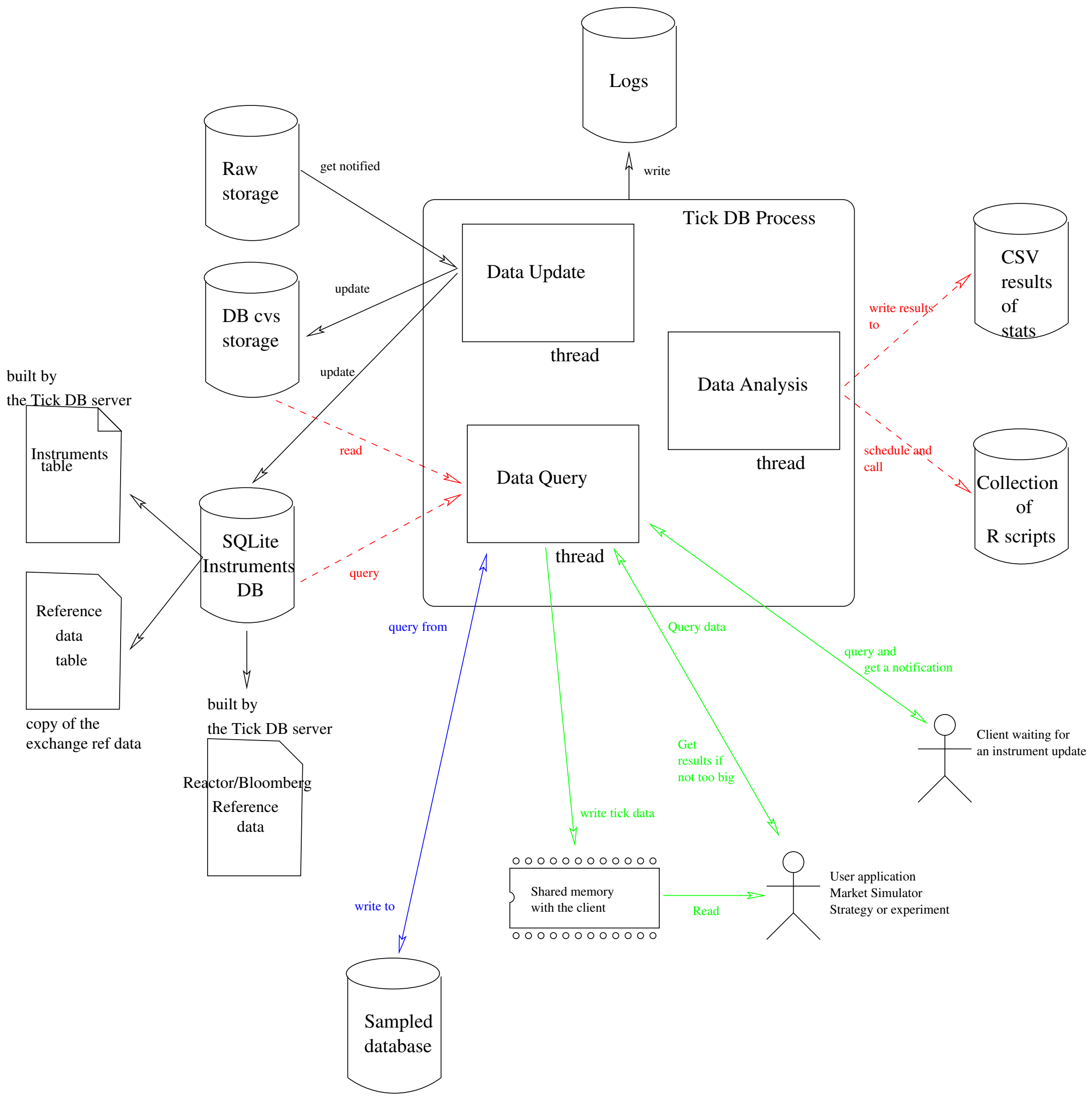


Figure 1: Overview of the system by functions and data repositories

JSON configuration file example

```
{
        "rootdir": "/mnt/data",
        "tmpdir" : "/tmp",
        "parjobfile" : "/tmp/gnupar_job_file.sh",
        "level": 5,
        "nbcores": 24,
        "shinyappdir" : "/home/dbellot/recherche/tickdatabase/database_builder/
        "decoder":
        {
                "BSXCBTMA1" : [{ "from": 0, "to":201312, "code": "not_valid"},
                                                { "from": 201401,"to":0, "code"
                "BSXCMEMA1" : [{ "from": 0, "to":201312, "code": "not_valid"},
                                                { "from": 201401,"to":0, "code"
                "BSXNYMMA1" : [{ "from": 0, "to":201409, "code": "not_valid"},
                                                { "from": 201410,"to":0, "code"
                "BSXCBT3MA1": [{ "from": 0, "to":0, "code": "qtg_mdp3"}],
                "BSXEURMA1" : [{ "from": 0, "to":0, "code": "qtg_eobi"}],
                "BSXCME3MA1": [{ "from": 0, "to":0, "code": "qtg_mdp3"}],
                "BSBUSKMA1" : [{ "from": 0, "to":0, "code": "qtg_kospi"}],
                "BSXASXMA1" : [{ "from": 0, "to":0, "code": "qtg_asx"}]
        },

        "liquid_capture":
        {
                "src_dir" : "/mnt/data/rawdata",
                "dbdir" : "/mnt/data/database/liquid_capture",
                "instdb": "/mnt/data/database/liquid_capture_inst_db.csv",
                "daily_db": "/mnt/data/database/liquid_capture_daily_db.csv",
                "prefix" : [
                        "CME_Agri_price-",
                        "CME_Index_price-",
                        "CME_Kospi_price-",
                        "CME_Metals_price-",
                        "CME_Treasuries_price-",
                        "CME_Crude_price-",
                        "CME_mdrec-",
                        "CNX_ETF_price-",
                        "Eurex_mdrec-",
                        "HKFE_Shim-",
                        "ICE_crude_price-",
                        "ICE_MFT_price-",
                        "ICE_brent_future_price-",
                        "ICE_fin_price-",
                        "ICE_sugar_future_price-",
                        "IDEM_Price-",
                        "MEFF_Price-",
                        "OSE-QTG-Shim-",
                        "SGX-TNG-Price-Server-Appliance-LCF60-",
                        "SIX_Price-",
                        "TNG-HKFE-QTG-Shim-",
                        "TNG-JPX-QTG-Shim-",
                        "UTP_aex_price-",
                        "UTP_cacind_price-",
                        "UTP_grains_price-",
                        "QH_ARCA-",
                        "QH_PROD-"
                        ],
                "owner" : "dbellot",
        "group" : "dev"
        },

        "qtg":
        {
                "dbdir" : "/mnt/data/database/qtg",
                "dbprocessed" : "/mnt/data/database/qtg_processed_files.db",
                "src_dir" : "/mnt/data/qtg",
                "unwanted" : "/mnt/data/database/unwanted_files.db",
                "instdb" : "/mnt/data/qtg/instRefdataCoh.csv",
                "daily_db": "/mnt/data/database/qtg_daily_db.csv"
        }
}
```