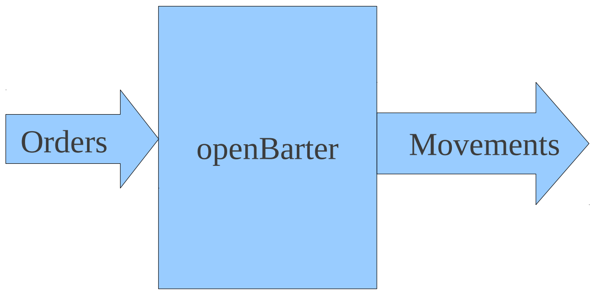
openBarter

openBarter implements a barter market place organized as a *central limit order book* allowing exchange of fungible values between two or more partners in a single transaction. It records orders of owners, and provides movements changing ownership of values according to orders and market rules.



1 The market

An order is expressed by the owner as:

I want to provide a value in exchange of an other value

The values provided and required are fungible values defined by a couple (quality,quantity) where the quality is a name and the quantity is an integer. A ratio between quantity offered and quantity required is used by the market instead of a price to compare competing orders.

The market does not require a central value standard to express prices.

The value provided is owned by the author of the order, and the quantity of the value required is the minimum expected in exchange[1]. Two orders match when the quality provided by the first equals the one required by the other.

The market contains the order book - a set of pending orders - scanned for each new order inserted looking for possible matchings. The following process occurs when an order O is submitted:

- The quantity available of O is set to the quantity provided,
- O is inserted in the order book,
- (***a***) If O does not form any cycle with pending orders, Stop.
- If O forms cycles with some pending orders a competition occurs to select a cycle C,
- for each order A of C the quantity available of A is used to create a movement and this quantity available is decreased of the same amount. The set of movements produced by C forms an agreement.
- Repeat from (***a***).

This process is wrapped in a single transaction that can be rolled back in case of hardware failure. This atomicity insures that for a given quality of the order book, the sum of available quantities of orders providing this quality added to the sum of quantities of movements providing

1 Quality provided and quality required must be different.

this quality is unchanged by the transaction even if it fails.

Movements define an agreement satisfying partially or entirely wants of orders of the cycle. A single transaction can produce several agreements and an order can produce several transactions. When the quantity available of an order becomes null, the order is removed.

This process does not differ from that implemented on a regular market except that an agreement can contain more than two movements. The competition performed on cycles is the same as that of a regular market when these cycles are bilateral.

Exchange agreements are formed so that the maximum limit defined by the ratio (quantity provided/quantity required) is satisfied even if the quantity received is lower than that required by the matching order.

1.1 Installation

1.1.1 Build from sources

Following instructions has been tested on linux 32 bits and 64 bits architecture with version 9.2 of postgreSql.

Follow instructions of postgreSql manual to install the sources of the database.

In the contrib/ directory of the sources of postgreSql, install the sources of openbarter using the package you downloaded from github:

```
$ cd contrib
$ gunzip olivierch-openBarter-vx.y.z.tar.gz
$ tar xf olivierch-openBarter-vx.y.z.tar
```

the package is compiled with:

```
$ cd openBarter/src
$ make
$ make install
```

1.1.2 Tests

To run tests, cd to openBarter/src and:

```
$ make installcheck
…
============== running regression test queries        ==============
test testflow_1               ... ok
test testflow_2               ... ok
test testflow_3               ... ok
test testflow_4               ... ok
test testflow_5               ... ok
test testflow_6               ... ok
============== shutting down postmaster               ==============

=====================
 All 6 tests passed.
=====================
```

1.1.3 Install the model

When the postgreSql server is running, the model can be installed. It is defined by the file openBarter/src/sql/model.sql. You must connect with a superuser role that is never user for market operations. When you are in openBarter/src:

```
$ createdb market
$ psql market
psql (9.2.0)
Type "help" for help.

market=# \i sql/model.sql
…..
```

The model does not depend of any schema, and creates roles *client* and *admin* if they do not exist yet. You quit psql by typing ctr-D.

To start operation, just connect to the database as admin, and create some clients with the function *createuser* like this:

```
$ psql -Uadmin market
psql (9.2.0)
Type "help" for help.

market=> SELECT fcreateuser('client1');
fcreateuser
-------------

(1 row)
```

1.1.4 execute primitives

This paragraph explains how commands behave with some examples. Connect as client1

```
$ psql -Uclient1 market
psql (9.2.0)
Type "help" for help.
```

1.1.4.1 simple bilateral exchange

Insert an order of an owner 'own1' that provides a value ('q1',2000) in exchange of a value ('q2',1000):

```
market=> select * from finsertorder('own1','q1',2000,1000,'q2');
NOTICE:  owner own1 created
CONTEXT:  PL/pgSQL function fgetowner(text,boolean) line 10 at assignment
PL/pgSQL function finsertorder(text,text,bigint,bigint,text) line 35 at
assignment
 id | uuid | own | nr | qtt_requ | np | qtt_prov | qtt_in | qtt_out | flows
----+------+-----+----+----------+----+----------+--------+---------+-------
  1 | 1-1  |   1 |  2 |     1000 |  1 |     2000 |      0 |       0 | []
(1 row)
```

The order is inserted with id=1 and uuid='1-1'. A new owner 'own1' and qualities 'q1' and 'q2' are inserted with their respective id (1,1,2). qtt_in, qtt_out and flows are empty since the order book contains only this order.

You can have a view of the order book with the command:

```
market=> select id,owner,qua_requ,qtt_requ,qua_prov,qtt_prov,qtt from vorder;
 id | owner  | qua_requ  | qtt_requ  | qua_prov  | qtt_prov  | qtt
----+--------+-----------+-----------+-----------+-----------+------
  1 | own1   | q2        |      1000 | q1        |      2000 | 2000
(1 row)
```

and of tables of owners and qualities with:

```
market=> select * from towner;
market=> select * from tquality;
```

Insert a new order :

```
market=> select * from finsertorder('own2','q2',1000,2000,'q1');
NOTICE:  owner own2 created
CONTEXT:  PL/pgSQL function fgetowner(text,boolean) line 10 at assignment
PL/pgSQL function finsertorder(text,text,bigint,bigint,text) line 35 at
assignment
 id | uuid | own | nr | qtt_requ | np | qtt_prov | qtt_in | qtt_out | flows
----+------+-----+----+----------+----+----------+--------+---------+-------
  1 | 1-2  |   2 |  1 |     2000 |  2 |     1000 |   2000 |    1000 | [..]
(1 row)
```

This order produced a flows [..], a json object, that presented separately for clarity:

```
 [[
{"id":1, "own":1,"nr":2,"qtt_requ":1000,"np":1,"qtt_prov":2000,"qtt":2000,
"flowr":2000 },
{"id":2,"own":2,"nr":1,"qtt_requ":2000,"np":2,"qtt_prov":1000,"qtt":1000,
"flowr":1000 }]]
```

A single bilateral exchange is created, where own1 provides (q1,2000) to own2 and own2 provides (q2,1000). Each movement presented within {} contains *id*, a reference to the order that produced the movement, *own* the owner, *np* the quality provided, *flowr* the quantity provided, and *qtt* the quantity remaining offered before this exchange. The two movements presented between [] represent the exchange. This exchange is itself presented between [] because more than one exchange can be produced by a single order. *qtt_in* and *qtt_out* returned by finsertorder() are the sum of quantities received and provided for all these exchanges.

Movements are recorded in a table *tmvt*. they can be viewed by the command:

```
market=> select nb,grp,provider,quality,qtt,receiver from vmvt;
 nb | grp | provider  | quality  | qtt   | receiver
----+-----+-----------+----------+-------+----------
```

```
  2 | 1-1 | own2       | q2        | 1000 | own1
  2 | 1-1 | own1       | q1        | 2000 | own2
(2 rows)
```

The order book is now empty since all orders are satisfied.

1.1.4.2 bilateral exchange with competition

We consider a scenario where

- 'own1' offers ('q1',5) in exchange of ('q2',10)
- 'own2' offers ('q1',10) in exchange of ('q2',5)
- 'own3 offers ('q2',20) in exchange of ('q1',10)

The first two orders are competing, and the second is the best:

```
market=> select * from finsertorder('own1','q1',5,10,'q2');
...(no matching)
market=> select * from finsertorder('own2', 'q1',10,5,'q2');
...(no matching)
market=> select * from finsertorder('own3', 'q2',20,10,'q1');
 id | uuid | own | nr | qtt_requ | np | qtt_prov | qtt_in | qtt_out | flows
----+------+-----+----+----------+----+----------+--------+---------+-------
  7 | 1-7  |   3 |  1 |       10 |  2 |       20 |     15 |      20 | [..]
```

This order produced a flows [..], a json object:

```
[[
{"id":6,"own":2,"nr":2,"qtt_requ":5,"np":1,"qtt_prov":10,"qtt":10,
"flowr":10 },
{" id":7,"own":3,"nr":1,"qtt_requ":10,"np":2,"qtt_prov":20,"qtt":20,
"flowr": 10 }
],[
{"id":5, "own":1, "nr":2, "qtt_requ":10, "np":1, "qtt_prov":5, "qtt":5,
"flowr":5 },
{"id ":7, "own":3, "nr":1, "qtt_requ":10, "np":2, "qtt_prov":20, "qtt":10,
"flowr":10 }
]]
```

representing two exchanges. The last order(id=7) exchanges with the second (id=6) , then with the first (id=5) because the order(id=6) is better than the order(id=5). The attribute *qtt* of the order (id=7) is decreased of the quantity provided by the first exchange. It is the quantity remaining available before the exchange.

The view vmvt gives a reference to the exchange, (here grp='1-5' and grp='1-7') and the number of partners of the exchange, (here nb=2) . oruuid is a reference to the original order:

```
market=> select nb,oruuid,grp,provider,quality,qtt,receiver from vmvt;
 nb | oruuid  | grp | provider  | quality  | qtt  | receiver
----+---------+-----+-----------+----------+------+----------
  2 | 1-2     | 1-1 | own2      | q2       | 1000 | own1
  2 | 1-1     | 1-1 | own1      | q1       | 2000 | own2
  2 | 1-4     | 1-3 | own2      | q2       | 1100 | own1
  2 | 1-3     | 1-3 | own1      | q1       | 1100 | own2
  2 | 1-7     | 1-5 | own3      | q2       |   10 | own2
  2 | 1-6     | 1-5 | own2      | q1       |   10 | own3
  2 | 1-7     | 1-7 | own3      | q2       |   10 | own1
  2 | 1-5     | 1-7 | own1      | q1       |    5 | own3
(8 rows)
```

1.1.4.3 non bilateral exchange with competition

We consider a scenario where

- 'own1' offers ('q3',320) in exchange of ('q1',80)
- 'own2' offers ('q2',20) in exchange of ('q1',20)
- 'own3' offers ('q3',540) in exchange of ('q2',20)
- 'own4' offers ('q1',100) in exchange of ('q3',100)

The first two orders are competing, and the second is the best:

```
market=> select * from finsertorder('own1','q3',320,80,'q1');
...(no matching)
market=> select * from finsertorder('own2', 'q2',20,20,'q1');
...(no matching)
market=> select * from finsertorder('own3', 'q3',540,20,'q2');
...(no matching)
market=> select * from finsertorder('own4', 'q1',100,100,'q3');
 id | uuid | own | nr | qtt_requ | np | qtt_prov | qtt_in | qtt_out | flows
----+------+-----+----+----------+----+----------+--------+---------+-------
 11 | 1-11 |   4 |  3 |      100 |  1 |      100 |    260 |     100 | [..]
```

This order produced a flows [..], a json object:

```
[[
{"id":9, "own":2, "nr":1, "qtt_requ":20, "np":2, "qtt_prov":20, "qtt":20,
"flowr":20 },
{"id":10, "own":3, "nr":2, "qtt_requ":20, "np":3, "qtt_prov":540, "qtt":540,
"flowr":180 },
{"id":11, "own":4, "nr ":3, "qtt_requ":100, "np":1, "qtt_prov":100, "qtt":100,
"flowr":60 }
],[
{"id":8, "own":1, "nr":1, "qtt_requ":80, "np":3, "qtt_prov":320, "qtt":320,
"flowr":80 },
{"id":11, "own":4, "nr":3, "qtt_requ":100, "np":1, "qtt_prov":100, "qtt":40,
"flowr":40 }
]]
```

representing two exchanges. The cycle with 3 partners is preferred to the bilateral cycle.

```
market=> select nb,oruuid,grp,provider,quality,qtt,receiver from vmvt order by
id;
 nb | oruuid | grp  | provider | quality | qtt  | receiver
----+--------+------+----------+---------+------+----------
  2 | 1-2    | 1-1  | own2     | q2      | 1000 | own1
  2 | 1-1    | 1-1  | own1     | q1      | 2000 | own2
  2 | 1-4    | 1-3  | own2     | q2      | 1100 | own1
  2 | 1-3    | 1-3  | own1     | q1      | 1100 | own2
  2 | 1-7    | 1-5  | own3     | q2      |   10 | own2
  2 | 1-6    | 1-5  | own2     | q1      |   10 | own3
  2 | 1-7    | 1-7  | own3     | q2      |   10 | own1
  2 | 1-5    | 1-7  | own1     | q1      |    5 | own3
  3 | 1-11   | 1-9  | own4     | q1      |   60 | own2
  3 | 1-9    | 1-9  | own2     | q2      |   20 | own3
  3 | 1-10   | 1-9  | own3     | q3      |  180 | own4
  2 | 1-11   | 1-12 | own4     | q1      |   40 | own1
  2 | 1-8    | 1-12 | own1     | q3      |   80 | own4
(13 rows)
```

The view *vmvt* shows that all partners received more than expected compared to what they expected, and that the barter is fair: the ratio *qtt_provided/qtt_required* has been increased in the same proportion for partners of an exchange.

2 The model

Let ω be the a ratio (quantity provided/quantity required) defined by an order. It measures the pain to give an amount of the quality provided compared to the pleasure to receive a unit of the quality required. The dimension of this measurement is (quality provided/quality required).

For a cycle of orders, let be Ω the product of their ω. This product is non dimensional.

When Ω equals to 1, an agreement can be formed where each partner provides some value to an other.

When $\Omega \neq 1$, ω are divided by the geometric mean of ω of the cycle. This division converts ω to ω' in such a way that the product of ω' equals to 1 . This adjustment is a bartering. It is fair when all partners are distinct. When it is not the case, the fairness is maintained by sharing it first between partners, then for each partner between it's orders.

To satisfy the minimum quantity required by the order, we must have $\omega > \omega'$, that is $\Omega > 1$. Otherwise, the cycle is ignored.

When an order forms several cycles, a competition is performed between them by choosing the one having the maximum Ω. This rule applied to cycles formed by two orders is equivalent to the best price rule of a regular market.

Computations produce numbers that need to be rounded to be stored and later presented as integers. These roundings are performed by minimizing a distance defined on the cycle in order to reduce the consequence of round-off errors on the fairness of the agreement.

3 Implementation

The market is seen as a directed graph where orders define nodes, and relations between orders define arrows. This graph is used to transform orders into exchanges when cycles appear on this graph. This can occur each time an order is added. A competition also occurs between possible cycles when more than one cycle is found. Quantities exchanged reduce quantities available of orders, and produce movements between owners.

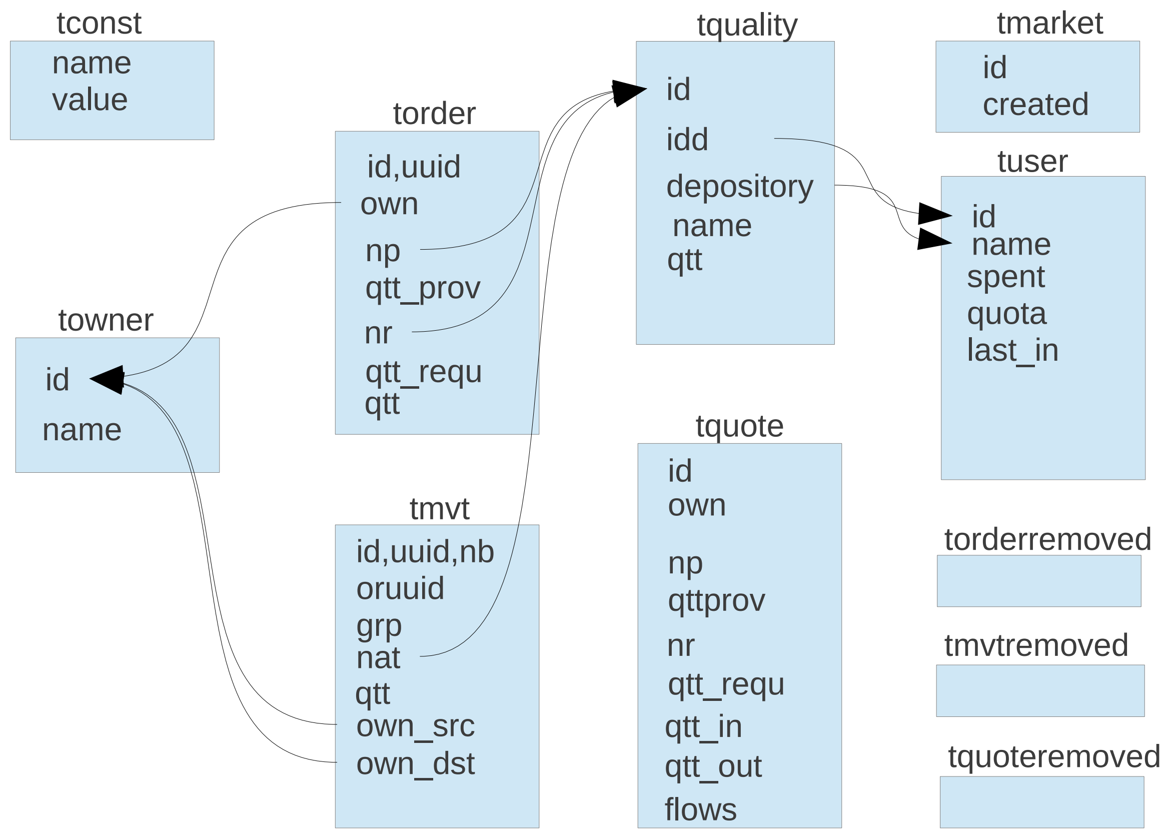
Stored procedures act on a model representing qualities, owners, orders and movements.

3.1 Database model

The database model is described by src/sql/model.sql. It consists in related tables and stored procedures. A type *yflow* representing a draft agreement and a type *yorder* representing an order are used to perform fast calculations in C language. They are defined by the file src/flow-1.0.sql.

Users that interact with the database have roles that can be *client* or *admin*.

Except on special circumstances, objects represented by rows are inserted or moved from a table to an other, but never deleted. Instead, they are moved in a table with the same name suffixed by *removed*. This way, periodic archiving of the database also contains all objects.



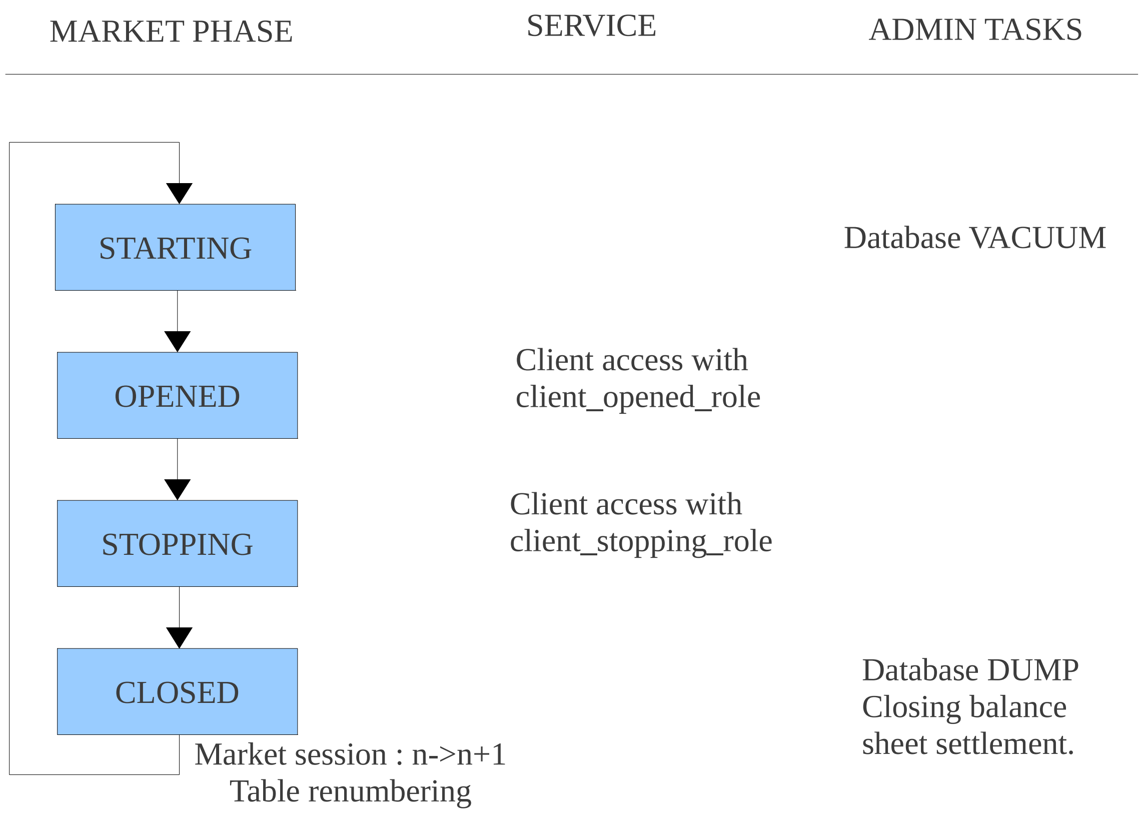
3.1.1 Order book

A table torder represents the order book. When an order is inserted in this table and a cycle is found with $\Omega \geq 1$ movements are created in a table tmvt, decreasing the quantities available in the order book.

When an order is empty (the provided quantity is 0) it is moved out from the book to an other table torderremoved. This keeps the order book as small as possible for performance. The garbage collector (§3.1.5) also removes orders in special circumstances.

3.1.2 Market opening and closing

The life cycle of the market is represented as follows:


MARKET PHASE
SERVICE
ADMIN TASKS
STARTING
Database VACUUM
OPENED
Client access with client_opened_role
STOPPING
Client access with client_stopping_role
CLOSED
Database DUMP Closing balance sheet settlement.
Market session : n->n+1 Table renumbering

The market session is defined as an integer incremented at each transition CLOSED->STARTING.

A client can use the market during the OPENED phase. At the transition CLOSED->STARTING tables tquality, torder and tmvt are renumbered and market session incremented. Accounting and technical administration tasks such as cold backup or closing balance sheet settlement can be performed during the CLOSED phase because the database represents the final and stable state of the ending market session. After this event VACUUM of tables must be performed by the admin user to optimize the database before a new cycle starts.

The history of status[2] is given by the view vmarkethistory:

```
SELECT * from vmarkethistory;
```

Each row gives a market session number, the market phase and the starting time of this phase.

The current state of the market, the session number, and starting time of this state are given by:

```
 SELECT * FROM  vmarkethistory ORDER BY ID DESC LIMIT 1;
```

2 The market status and market phase have the same meaning.

A view vmarket gives the same result:

```
 SELECT * from vmarket;
```

The command:

```
SELECT * from fchangestatemarket(false);
```

Gives informations on the transition to the next state. This transition is performed by:

```
 SELECT fchangestatemarket(true);
```

3.1.3 Users

By user, we mean the actor connecting to the PostgreSQL database. The database implements an extensive set of security mechanisms including authentication and access rules. openBarter uses those mechanisms to allow write access to objects only through predefined functions. A single user “admin” is allowed to perform administration tasks. A role “client” is that of regular users. Clients can get a quote and set an order only during the OPENED phase. They can remove movements and orders only during OPENED and STOPPING phase.

A role *client_opened_role* is granted to the *client* role only at the OPENED phase, while a role *client_closing_role* is granted to the *client* role only at the CLOSING phase.

The admin can register a new client, change the state of the market, but cannot participate to the market. He can register a new client by the command:

```
SELECT fcreateuser(<user_name>);
```

The super user that creates the database is distinct from *admin* and *client*. This super user must be used only for this creation.

3.1.4 Objects

All objects are stored in tables.

Table	Description
tmvt	History of movements where the ownership of values are moved between owners. It is the output flow of the market.
tmvtremoved	Movement removed
torder	Order, value provided, value required and owner
torderremoved	Order removed
tquote	Quote, value provided, value required, owner, and flows of value produced
tquoteremoved	Quote removed
towner	Owner
tquality	Quality
tuser	Description of client
tmarket	History of market
tconst	Constants of the market

3.1.4.1 Owner

They are owners of values provided by orders and the authors of orders, while clients connect to the database and act on the market on the behalf of owners. An owner is defined by a name. It is recorded when the first order of this owner is recorded in the current session of the market.

3.1.4.2 Quality

A quality is a string. It's form depends on a constant CHECK_QUALITY_OWNERSHIP in the table *tconst*.

When CHECK_QUALITY_OWNERSHIP=0, the name of the quality can be any non empty string.

When CHECK_QUALITY_OWNERSHIP=1, the quality belongs to a single client whose name is *client_name*. The following rules are implemented:

- The quality provided by an order must belong to the client that insert it.
- A client can only remove movements whose quality belongs to him.

The form of the quality is *<client_name>/<quality_name>*.

A value belongs to an owner while a quality belongs to a client.

A quality is inserted into *tquality* the first time an order use it in a market session.

After market creation, This constant should not be changed after the market is created. It can be changed only by a super user.

3.1.4.3 Movement and Order

An exchange is a set of movements representing a cycle where each partner provides a value he owns to an other partner. An exchange is simply a set of record in the table *tmvt*, where each defines the value, the provider, the receiver (see §3.2.9 and §3.2.11 for details).

3.1.4.4 Quote

Represents a quote made by an owner.

Column	Type	Meaning
id	int	Internal id of the quote that is used to reference the quote in order to execute the corresponding order.
own	int	Internal reference to the owner.
nr	int	Internal reference to the quality required.
qtt_requ	int8	Quantity required by the quote.
np	int	Internal reference to the quality provided.
qtt_in	int8	Sum of quantities received by flows produced by the quote. The couple (nr,qt_in) define the value received.
qtt_out	int8	Sum of quantities provided by flows produced by the quote. The couple (np,qtt_out) define the value provided.
flows	yflow[]	An array of flows representing the list of agreements

		produced by the quote.
created	timestamp	Time when the quote is created.
removed	timestamp	Time when the quote is moved to the table tquoteremoved.

The id is a unique key referencing this quote. (nr,qtt_requ) is the value received while (np,qtt_prov) is the value provided if the quote was executed.

The type *yflow* represents an exchange. An order is a tuple (*id*,*own*,*nr*,*qtt_requ*,*np*,*qtt_prov*,*qtt*) where *id* is its unique key, *own* the owner of the value provided, (*nr*,*qtt_requ*) the value required, (*np*,*qtt_prov*) the value initially provided, and *qtt<=qtt_prov* the quantity remaining available for exchange. *yflow* is a list of such orders, where the quality provided by one equals the quality required by the next one. A given *yflow* defines a flow of quantities provided by each partner (owner) of *yflow*. The field *flows* is the set of exchanges that would be created if the quote was executed.

3.1.5 Garbage collector of orders

Orders that are frequently included in refused cycles tend to slow the speed of matchings. Orders that belong to potential cycles but are not elected to form exchanges are removed from the searching table when they consume too much time. The time consumed by an order is accumulated until a limit is reached. The limit is a constant tconst.MAXTRY expressed in microseconds. Rejected orders are moved to torderremoved. The quantity available creates a single movement from the owner to the same owner for accounting consistency reason.

3.1.6 Limits on potential exchanges

Due to the number of possible combinations, scanning required to form cycles can be huge. The traversal of the order book is limited by the number of partners of cycles explored. This limit is defined by a constant tconst.MAXCYCLE.

For the same reason, the exploration stops when the number of potential cycles fetched reaches a limit tconst.MAXPATHFETCHED. Since all cycles having a given number n of partners are explorated before those having n+1 partners; this mechanism limits the number of partners of cycles for high workload.

3.1.7 Quotas

The time spent to execute long primitives (*finsertorder*,*fgetquote*,*fgetprequote*,*fexecquote*) is cumulated for each client. When this time spent reaches a limit defined by the field *tuser.quota*, these functions become forbidden for this client.

The time spent is cleared when the market session is opened.

The quota allocated to a client can be disabled by setting the quota of the user to 0. When set to a non null integer, it limits the total number of microseconds allocated to this client. It can be set individually for each client.

3.2 Application programming interface

The *client* role acts through stored procedures wrapped in read-commited transactions that is the default mode of transactions of postgreSQL.

The following table lists functions and views of the model.

Function and views	action	Market phase	Roles allowed
finsertorder	Inserts an order	OPENED	client
fgetprequote	Gets a prequote	OPENED	client
fgetquote	Gets a quote	OPENED	client
fexecquote	Executes a quote	OPENED	client
fremoveorder	Removes an order	OPENED	client
fgetagr	Describes an agreement		
fremoveagreement	Removes movements	OPENED, CLOSING	client
fgetstats	Produces statistics		admin
fgeterrs	List of errors		admin
fchangemarketstate	Change the state of the market		admin
fcreateuser	Creates a user		admin
vorder	List of pending orders		
vorderremoved	List of removed orders		
vorderverif	List of active and removed orders		
vmvt	List of pending movements		
vmvtremoved	List of removed movements		
vmvtverif	List of removed movements		
vmarket	Market state		
vmarkethistory	Market history		

In case of error, an exception is raised depending on it's type and current transaction is rolled back.

Error codes	Type
YA001	Quantity of a given quality overflows
YA002	accounting error
YA003	internal error
YU001	abort dues to incorrect use of a primitive

In the following, int is used for 32 bit integer, and int8 for 64 bits integer.

3.2.1 finsertorder

```
SELECT finsertorder(
_owner text,
_qualityprovided text,
_qttprovided int8,
_qttrequired int8,
_qualityrequired text);
```

conditions :

- ***_qttprovided*** > 0
- ***_qttrequired*** > 0

the function inserts an order made by *_owner* providing the value (*_qttprovided,_qualityprovided*) in exchange of a value having the quality *_qualityrequired* and for a minimum quantity of *_qttrequired*.

Possible cycles are found and converted into movements. The remaining quantity provided that is not used by theses exchanges is stored in the order object.

The record returned is a y*resorder* representing details of exchanges produced:

Column	Type	Meaning
id	int	Internal reference to the order
uuid	text	The reference of the order (session number - id)
own	int	Internal reference to the owner
nr	int	Internal reference to the quality required
qtt_requ	int8	Quantity required
np	int	Internal reference to the quality provided
qtt_prov	int8	Quantity provided
qtt_in	int8	Sum on flows of quantities received
qtt_out	int8	Sum on flows of quantities provided (qtt_out <= qtt_prov)
flows	json	The list of exchange produced

The ratio *qtt_out/qtt_in* is lower than *qtt_prov/qtt_requ* and *qtt_out <= qtt_prov*.

The json *flows* represents an array of exchanges produced by the order. Each order is an array of movements described by a dictionary with the following keys:

Key	Type	Value

id	int	Internal reference to the order
own	int	Internal reference to the owner
nr	int	Internal reference to the quality required
qtt_requ	int8	Quantity required by the order
np	int	Internal reference to the quality provided
qtt_prov	int8	Quantity provided by the order
qtt	int8	Quantity of the order remaining available before the exchange
flowr	int8	Quantity exchanged

3.2.2 fgetquote

```
SELECT fgetquote(_owner text,_qualityprovided text,_qttprovided
int8,_qttrequired int8,_qualityrequired text);
```

conditions :

- ***_quantityprovided*** >0,
- ***_quantityrequired*** >0,

It provides the results that would be obtained if *finsertorder* was executed with these arguments. It returns a record *tquote* describing produced agreements. The *id* field can be used to reference this quote for execution of *finsertorder* with the same arguments.

3.2.3 fexecquote

```
SELECT fexecquote(_owner text,_id int);
```

conditions :

- ***the quote has been submitted with the same owner***,

It executes a *finsertorder* with the arguments of the referenced quote. Agreements provided by the quote are the same as those provided by the quote is the market is unchanged between the quote and it's execution. The quote is removed after execution. It returns a record *tresorder* (described in §3.2.1).

An error is returned when the quote does'nt exist or was not created with the same owner.

3.2.4 fgetprequote

```
SELECT fgetprequote(_owner text,_qualityprovided text,_qttprovided
int8,_qualityrequired text);
```

conditions :

- ***_quantityprovided*** >0,

Gets a quote without defining the quantity required. This prequote is used to have an idea of the quantity required by the market in order to make a quote using *fgetquote*.

The quote depends on the owner because fairness of the bartering depends on it (see §7). For each cycle the quantity provided is such as no barter is required.

It returns a record *yresprequote* with the following fields:

Column	Type	Meaning
own	int	Internal reference to the owner
nr	int	Internal reference to the quality required
qtt_prov	int8	Quantity provided
np	int	Internal reference to the quality provided
qtt_in_min	int8	Quantities received and provided by the agreement having the minimum ω
qtt_out_min	int8	
qtt_in_max	int8	Quantities received and provided by the agreement having the maximum ω
qtt_out_max	int8	
qtt_in_sum	int8	Sum of quantities received and provided by flows produced
qtt_out_sum	int8	
flows	json	The list of agreements produced

3.2.5 fremoveorder

```
SELECT fremoveorder(_uuid text)
```

conditions:

- an order with the label ***_uuid*** exists

The order is removed from the order book.

Returns a row representing the order just removed, as the view ***vorder*** does.

3.2.6 fremoveagreement

```
SELECT fremovemvt(_uuid text)
```

conditions :

- a movement ***_uuid*** exists

The function is called by a client when all movements of the exchange are red from the table of movements. It the movements _uuid into the table ***tmvtremoved*** if the movement belongs to this client to. The function returns an integer that is the number of movements removed.

3.2.7 fcreateuser

```
SELECT fcreateuser(_username text)
```

The function creates the user and provides access to he database with the role client. It can only be executed by admin.

3.2.8 fstats

```
SELECT fstats(_extra bool)
```

gives general informations about the database:

Column	Type
Number of qualities	int
Number of owners	int
Number of quotes	int
Number of orders	int
Number of movements	int
Number of quotes removed	int
Number of orders removed	int
Number of movements removed	int
Number of agreements	int
Number of orders rejected	int
For each agreement length, the numer of agreements	int

3.2.9 vorder

Gives a description of the order.

Column	Type	Meaning
id	int	Internal reference of the order
uuid	text	External reference of the order
owner	text	Name of the owner
qua_requ	text	Quality required
qtt_requ	int8	Quantity required
qua_requ	text	Quality required
qtt_requ	int8	Quantity required
qtt	int8	Quantity not yet exchanged for this order
created	timestamp	Time when the order was inserted
updated	timestamp	Time when the order was last updated

examples

```
SELECT * FROM vorder WHERE owner='jack';
```
List of orders owned by the owner '***jack***'.

3.2.10 vorderremoved

Same as vorder.

When the quantity left in order is 0, the order is appears in this view. When the order is removed by a client, it also appears here with a qtt >0.

3.2.11 vmvt

It is the list of movements.

Column	Type	Meaning
id	int	Internal id of the movement
uuid	text	External reference of the movement (session id - id).
nb	int	Number of movements of the agreement
oruuid	text	Reference to the order that produced it
grp	int	id of the agreement. It is the id of the first movement of this agreement.
provider	text	Owner providing the value
nat	text	Quality of the value moved
qtt	int8	Quantity of the value moved
receiver	text	Owner receiving the value
created	timestamp	Time when the agreement was formed

examples:

```
        SELECT * FROM vmvt WHERE quality='gold';
```
List of movements of the quality '***gold***'.

3.2.12 vmvtremoved

Same as vmvt but for the table tmvtremoved.

3.3 Releases

0.1.0

First release. Tests units are functional [Olivier Chaussavoine].

0.1.1

Berkeley-db is resides in memory instead of files in $PGDATA. This increases global performance of searches. [Olivier Chaussavoine]

0.1.2

rights of roles of the database model are defined globally using schemas instead of granted individually for each function. [Olivier Chaussavoine]

0.1.6

ported on postgres9.1.0

0.2.0

The use of berkeleydb is replaced by WITH .. SELECT of PostgreSQL. A new type “flow” is defined, containing low level calculations. Tests units are functional [Olivier Chaussavoine].

0.2.1

Memory allocation and code cleaned. Tests units are functional [Olivier Chaussavoine].

0.2.2

Core algorithms optimized. Tests units are functional [Olivier Chaussavoine].

ob_fget_omegas(np,nr) provides the list of all prices found, even those not requested. [Olivier Chaussavoine]

0.3.0

The constraint of acyclic graph is removed. Complete redesign. [Olivier Chaussavoine].

0.4.0

quote and prequote added. [Olivier Chaussavoine].

Order rejection mechanism added [Olivier Chaussavoine].

0.4.1

ported on postgreSql 9.2. [Olivier Chaussavoine].

Bug fixes [Olivier Chaussavoine].

0.4.2

Bug fixes [Olivier Chaussavoine].

0.5.0

fgeterrs() optimized, it can be run when the market is running,

index optimization in fcreate_temp()

increasing preformance of fgetprequote(),fgetquote(),fexecquote(),finsertorder()

X6 faster

MAXCYCLE was 8, it can now be up to 64 [Olivier Chaussavoine].

.