Real-Time database : Firebase
INFO-H-415 : Advanced database

Baudoux Nicolas et Bauwin Lucie

Univeristé Libre de Bruxelles
nbaudoux@ulb.ac.be, lbauwin@ulb.ac.be

Academic year 2017-2018

Abstract

This report will present our project concerning the model of real-time database systems (RTDBS). We will define the concept of real-time database, their features and their specificities. Then we will present some existing softwares to manage a RTDBS and give examples of applications of such systems. Finally, we will develop a Firebase real-time database for one of these examples and explain its functionalities.

1 Introduction

A classic database is a collection of information organized and which can be easily accessed, managed and updated. Generally, data are organized in tables. This is not always the case as we will see in our example of real-time database system. This kind of databases without predefined tables is called schema-free. A database can be of different types. The most common are SQL, NoSQL and cloud databases.

A real-time database system is a classic database system which is providing real-time constraints and ensure reliability on system's timing requirements. Timing constraints are not required to be extremely short but the database need to manage explicit time constraints in a predictable way using time-cognizant methods. This kind of database combines multiple features facilitating :

— Description of data;
— Maintenance of correctness and integrity of data;
— Efficient acces to the data;
— Correct execution of query and transaction execution in spite of concurrency and failures.

We will define in a more formal way these notions and explain differences between traditional database and real-time database.

2 Definitions

2.1 Real-time processing

Real-time processing or computing is a system which is subject to "real-time constraints". It means that this system is able to "control an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time"[1].

2.2 Real-time database

Intuitively, a real-time database may be seen as a classic database which is able to handle workloads whose states are permanently changing by using real-time processing.

If we cross the definition of real-time processing and the intuition presented above, we may re-write a more formal definition of real-time database : *A Real-Time database is a database which follows a number of time constraints. Those are the temporal characteristics of the data stored in the database, the timing of the queries and the performance goals.*

The real-time database mostly differs from a traditional database by its performance goals, time constraints which are in microseconds or even in nanoseconds and by its ability to evaluate the average of the missed transactions and the cost incurred in these missed transactions.

Furthermore, like conventional databases, real-time databases have to respect ACID properties which are :
— Atomicity : In the end, a transaction is never half-completed. It is completely done or not.
— Consistency : Transactions are run in a given specific order.
— Isolation : Transactions cannot see actions of another transaction while it is not yet commited.
— Durability : Once commited, a transaction cannot be modified.

3 Why real-time database ?

Real-time database are adequate when we try to have a good picture of the current state of an environment. It uses time constraints that are not supported in conventional databases. Unlike the conventional database, a RTDBS uses deadlines to determine the accuracy of the measured value. Traditional databases have poor responsiveness and a lack of predictability which are main features of a RTDBS. Finally, a data never becomes obsolete in a traditional database which is not the case in a real-time database system (static data vs. real-time data).

4 Is a RTDBS a temporal database ?

No. Even if the two systems support time-specific transactions, temporal databases associate to data a span of validity and real-time databases have a defined span of time in which a transaction has to be executed.

5 Model

A real-time database is composed of two main parts :

- Controlling system : Meeting point between the computer and its software.
  and
- Controlled system : System which is perceiving the state of the environment observed and saved in the database.

A high degree of accuracy must be maintained between the actual state of the environment and the state of the environment in the database. Therefore the environment's monitoring at fixed interval is essential.

5.1 Data consistency

To the timing constraints one must add the timing correctness requirements to respond to the need to make data available for the controlling system and its decision making activities. This leads to the notion of temporal consistency which has two constituents:

- Absolute consistency: data is only valid during a determined period of time. This helps to keep the database corresponding with the actual state of the environment.
- Relative consistency: it is required that data represents the state of the environment during a time interval so the delay to update the different data in the tables is not taken into account. The goal is to derive new data from existing information.

An item is temporally consistent if and only if it is absolutely and relatively consistent.

5.1.1 Example

Assume we want to know the electric efficiency of a train, we will need at least two tables:

- Electric consumption
- Covered distance

The train consumption is registered at time 100msec and has consumed 0.6W and the distance it traveled is 10meters. and is registered at time 110msec. Both information have an absolute consistency of 20msec.

$$d_{elec}(0.6W, 20msec, 100msec)$$

$$d_{distance}(10m, 20msec, 110msec)$$

With those data, we can observe that the two datas are contemporary for an interval of 10msec.

5.2 Transaction in real-time database system

A transaction can be informally defined as an operation where :

— the set of operation is *read*, *write* and *abort/commit*;
— if an abort is executed a non-commiting operation is launched
— if an operation t performs an abort or commit then all other operations have priority on operation t as defined in the ordering relation.
— if operations *read* and *write* are launched on the same data at the same time then the ordering relation defines the priority.

Transactions can be identified with three dimensions :

— The way transactions use the data
— The nature of time constraints
— The importance of processing a transaction before a given deadline

A RTDBS transaction has a specific set of attributes:

— Timing constraints: a series of constraints applied to the different tables of a database. It may be period, frequency, maximum delay between end points, or maximum net delay.
— Criticalness: determine how critical it is that a transaction is executed in time. In other words, criticalness defines how important the transaction is and the effect it may have if it misses its deadline.
— Ressource requirements: represents the required CPU time and the amount of I/O operations needed to execute the transaction.
— Expected execution time: measures with the worst case scenario but usually hard to define.
— Periodicity: only valid when a transaction is repeated periodically.
— Time of occurence of events: determines the time a transaction launch a request.
— Other semantics.

5.2.1 Timing constraints

A timing constraint can be :

— Hard : Transaction has to be completed before the deadline. This means that "the best effort" is not enough. The transaction has to be periodic. The resources requirements and the worst-case execution time must be known;
— Firm : If the transaction is not completed before the deadline, then it's aborted;
— Soft : The importance of the information is decreased if it's completed after the deadline. But the transaction is anyway continued until it is completely done.

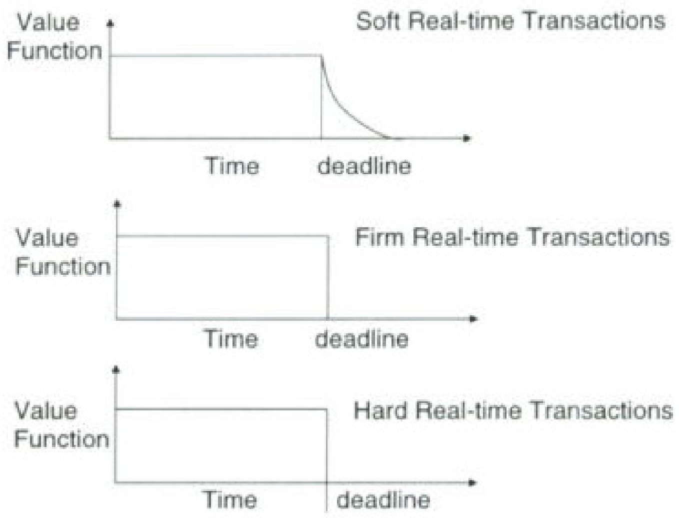

Value Function
Soft Real-time Transactions
Time
deadline
Value Function
Firm Real-time Transactions
Time
deadline
Value Function
Hard Real-time Transactions
Time
deadline

Figure 1: Deadlines represented with value functions

A timing constraints can be specified by an Event-Condition-Action rule :

```
ON ( event )
IF ( condition )
DO ( action )
```

There are three types of transaction read-only (read and transmit the data from the database to the controlling system), write-only (obtain the state of the environment from the controlled system and report it to the database) and updates (derive a new data item and report it to the database).

Now that these notions are well defined, we can say that a transaction is specified by:
— its implication of missing deadline, which can be hard, critical or soft;
— its arrival pattern which can be periodic or not;
— its data access pattern which can be read-only, write-only, update or random.
— its runtime requirement which can be known or unknown

6 Processing

When using a real-time database, one of the main issue is predictability. We need to be able to predict if a transaction will respect its deadline or not, thus the only way to make sure of it, would be to know the worst-case execution time. However there are many sources of unpredictability present:

— dependence of the transaction's execution sequence
— data and resource conflicts
— dynamic paging and I/O
— transactions abort and the resulting rollbacks and restarts
— communication delays and site failures

which sometimes make it impossible to predict the worst-case scenario.

Yet we can appease the effect of those unpredictibility sources. We can use a main memory database and give priority to the I/O controllers to solve the problem linked to dynamic paging and I/O. We can also allow a transaction to write-only in its memory area before writing the transaction's changes to the database which could cause a rollback to occur.

The queuing of the transactions are based on their criticalness and priority.

6.1 Priority

We can approach priority with different methods :

— EDF (Earliest Deadline First) : This algorithm consists in a priority queue where the next task to accomplish is chosen based on the closest deadline. The algorithm updates the task to handle every time a new task is added to the priority queue;
— Highest value first : The task defined as the most important has the priority on the other tasks;
— Highest computation time first : The task which is estimated to take more time to execute than the others is executed first;
— complex function of deadline, value.

Those different methods will have a different impact on the database system performance.

6.2 Execution time

To compute the execution time of a transaction, we can use the following formula :

$$t_{exec} = t_{db} + t_{I/O} + t_{int} + t_{appl} + t_{comm}$$

where :
t_{db} is the processing of DB operations
$t_{I/O}$ is the I/O processing
t_{int} is the transaction interference
t_{appl} is the non-DB application processing
t_{comm} is the communication time

7 Comparing differents softwares

There are many softwares which can be used to manage a real-time database. We chose to get a closer look at four of them, CouchDB, MariaDB, RethinkDB and Firebase.

CouchDB is a document storage open-source database system using JSON standard file format developped by Apache software foundation and written in Erlang. It provides an intuitive API which is restfull HTTP to read and write database document. The documents are stored using B-Trees [1] and hashed according to the files' name. It allows to reliably store data from different places in sync over unreliable network. A transaction is either totally completed or completely failed, thus a document will never be partially stored. This database system is schema-free which means that a data can be stored without a previous defined structure [9]. This feature facilitates the migration of the data.

MariaDB is a relational open-source database system using JSON standard file format since the version 10.2 developped by MariaDB Corporation Ab and written in C and C++ based on MySQL. It is why MariaDB includes core functionality of MySQL. It has a high scalability. It can be accessed via APIs like ADO.NET, JDBC and ODBC. The data is stored using B-Trees, Hash or R-Trees [2]. Unlike most databases system, MariaDB provides a secure environment.

RethinkDB is a document storage open-source database system using JSON standard file format developed by the Linux foundation and written in C++. It does not provide a specific API but can be accessed through external APIs handling JSON files. Alike CouchDB, it is schema-free. RethinkDB is using an embedded domain-specific query language called ReQL.

Firebase is a commercial cloud-hosted document storage database system developped by Google. It provides multiple platforms for the different usage of the sofware (Android, IOS, Javascript, Restfull HTTP,...). It is also schema-free. Alike MariaDB, Firebase hosting provides a secure environment. Finally, this RTDBS is using the NoSQL storage mechanism.

Of the four presented options, we chose to continue with Firebase because it is well-documented and furnish all the features we need for our implementation. Even though Firebase is a commercial database system, a free license is available.

7.1 Comparative table

1. self-balanced tree data structure that keeps data sorted [6]
2. refer to multidimensional B-Trees

	Storage system	SQL	JSON	Open-Source	Schema-free	APIs
CouchDB	Document	x	x	x	x	Restfull HTTP
MariaDB	Document	x	x	x		ADO.NET, JDBC, ODBC
RethinkDB	Relational	ReQL	x	x	x	external handling JSON files
Firebase	Cloud	NoSQL	x	Commercial	x	Android, IOS, Javascript, Restfull HTTP

8 Firebase

8.1 NoSQL

Firebase does not use traditional relational queries but NoSQL which stands for "Not only SQL", it is more scalable and more performant. NoSQL allows horizontal scaling between different server where relational databases only offer vertical scaling which is not optimize when storing loads of information.

There are four types of NoSQL database:

— Document databases oriented, information is represented as documents (JSON, XML,...). Those documents can hold lots of key-value pairs.

— Graph oriented, information is represented as node and relation. It allows the user to recover complex data easily.

— key-value oriented, information is stored as a value (integer, string, object,...) paired with a key.

— wide-column oriented, store columns of data instead of rows.

8.2 Cloud

As a result of firebase being developped by Google, it uses a few Google platform such as the Cloud. It is a cloud computing platform designed by Google enabling storage.

8.2.1 Cloud Messaging

Firebase Cloud Messaging make it possible for the server to send notification to the client app to warn him of a new email or other data messages. The messages sent from the server can either be distributed individually, by group or by topics interest. FCM also provide the possibility for the client to send messages, appreciation and other data back to the server. This feature requires two components to enable sending and receiving messages:

— a trusted environment

— An iOS, Android, or web client app to receive messages.

8.2.2 Cloud Functions

Cloud functions are codes automatically responding to events triggered by Firebase and Google Cloud features or HTTP requests. The triggers include Realtime database triggers, Firebase authentication triggers, HTTP triggers, Cloud Firestore Triggers,...

Cloud Functions are protected from the client, he will never be able to access those private functions because of Firebase security system.

Once deployed on the servers, all resources needed are automatically generated to match the usage patterns. The server does not need manual maintenance. Usually the functions are used to :

— inform the user something is happening

— perform realtime database sanitization and maintenance (backend)
— perform rigorous task in the cloud to not overload the app

Inform the user : Cloud functions can inform the app users with relavant information about the app (updates, followers, confirmation email,...)

The system works as follow:

1 The function is triggered by a new element in the database
2 The function composes a message to send
3 FCM sends the notification to the user's device

Perform Realtime Database sanitization and maintenance : Say a function is triggered by a new entry into the database provoked by a user, then the function is used to keep the system clean and up-to-date.

This is used to scrub any inappropriate language in a chat or to delete and purge an entry in the database.

Perform rigorous task : Firebase client can use the Google cloud resources (CPU or networking) to relieve the user's device. It can be used to periodically delete unused accounts, send email to users,...

It is this feature that will allow us to write events triggers to keep our database consistent

8.3 Authentification

In order to securely store an user data and personalized the user page, we need to know the identity of the user to meet this need, Firebase has its own backend services for the allow authentication. The authentication can be based on password, phone numbers or federated identity (Google, Facebook, Twitter,...).

You can identify the user in multiple ways:

Email and password
Federated identity provider integration
Phone number
customize your own system integration
or Anonymous

There are several way to sign-in, to integrate the feature, we can either use FirebaseUI or use the Firebase Authentication SDK.

FirebaseUI : An open-source interface containing loads of authentication solution that can be customize to match the app style. It is the easy solution since it provides a complete sign-in system to the app and it is directly integrated into the system.

Firebase SDK Authentication : This way is used when we want to manually integrate one or several sign-in methods.

8.4 Realtime Database

As a real-time database, this system synchronizes the database with every connected client instead of using HTTP requests. This means there's only one common instance of the database for every connected user which automatically receive updates. Thus, a deadline on a transaction may not be implemented as the Firebase system because a notification of new information is sended to every connected user. Every user will receive the update as soon as possible[17].

Once offline, every device maintains a local copy of the database and stores new data on a local disk until it recovers the connectivity. Then information is uploaded to the online real-time database, the online system resolves conflicts by itself. And the user's device retrieves the changes he missed while he was offline.

8.5 Storage

As explained before, Firebase Real-Time Database system is cloud-stored and encoded as JSON objects. The database can be seen as a tree where each information is stored as a node coded in JSON with an associated key. The data itself is structured according regular expression rules language.

In a Firebase database, there is no tables or records. A node is added to the existing structure when a new data is added into the database. This kind of tree can accept up to 32 level of node. In order to keep the data safe, it is important to know that giving access to a node is also giving access to all children of this node. This is why Firebase documentation recommands to keep a relatively flat JSON tree.

8.6 Hosting

A Firebase Database cannot be hosted locally and has to be hosted on Firebase servers. But it is possible to test it locally via the Firebase Database Manager called *Firebase Command Line Interface*. This interface allows to manage, test and deploy a database.

8.7 Performance

The Realtime Database Firebase API is design to only allow operations that can be quickly achieved. And to control that, Firebase provides a system which allows the administrator to monitor the performance of the database. This monitoring is charaterizing performance by different metrics which are :

— Response time : Time between the client's request and the end of the full transmition of the answer to this request;
— Payload size : charge down- and uploaded by the app in byte;
— Success rate : Ratio between successfully transmited responses and total amount of responses. Allows to know the percentage of failed transmitions.

To every data of the performance monitoring is associated a set categories which describes on which device, operating system, version of the application and in which country, etc. the request has been made.

9 Applications

9.1 Stock Exchange Markets

One application of a real-time database is in the context of Stock Exchange Markets transactions. In this particular example, a database has to save every:
— variation of price of each stock quote;
— trade of a quantity of stock;
— offer of a quantity of stock;
— bid on a quantity of stock;
— ...

Even if the number of entries is depending on the granularity [3] chosen, the state of the current price of a stock quote seems to be very unstable and constantly changing. Furthermore, this information can be needed by a buyer, seller or owner in a short period of time. Thus, a stock exchange markets is an example of environment which requires the kind of database we are currently studying.

9.2 Automatic tracking and object positioning

Industries have the need to know where are their assets (trucks, busses, boat, plane, etc.). To do so, they use real-time positioning system combined with a real-time database which stores all In-Vehicle information and is updating each time the GPS sends position data.

For example such a database could store[15] :
— Information about users of the system
— Information about vehicles (fuel, millage, etc.)
— Information about received from vehicles (position)

9.3 Banking systems

Another application of a real-time database is for money transactions. Nowadays, every transaction is stored to ensure that the balance of every costumer's

3. The time interval between two observations (e.g.: Daily closing prices, prices for each minute of the day)

account is accurate at every moment. This database has to be reacheable from every interface where a user can manage his account. Thus, the bank can ensure that a costumer cannot spend the same money twice for example.

Furthermore, a real-time database system can be even more useful in ”real-time banking”[14]. The use of real-time databases can give to the banks the opportunity to offer immediate money transfers.

We will implement this application in a Firebase real-time database.

9.3.1 Database model

We first tried to obtain an existing database. Unfortunately, no banks will share their data because of privacy policy. Therefore we decided to create generic data for our example. Figure 2 represents the relational model of our database.

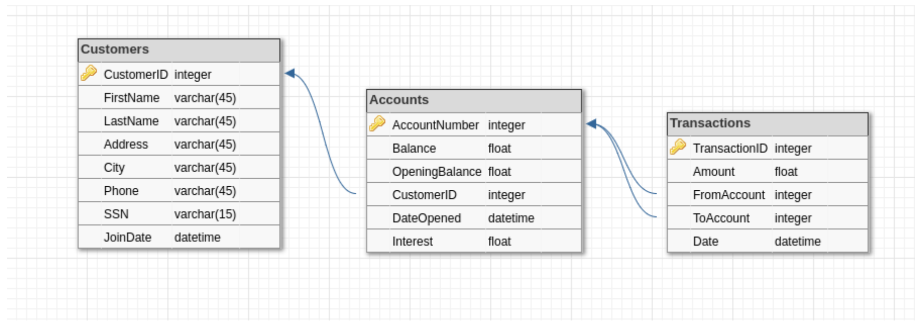


Figure 2: Relational diagram of our database

We implemented the three main tables necessary to maintain information of a basic banking system. For the simplicity of our example, we are considering that all transactions are made between two accounts from the same bank.

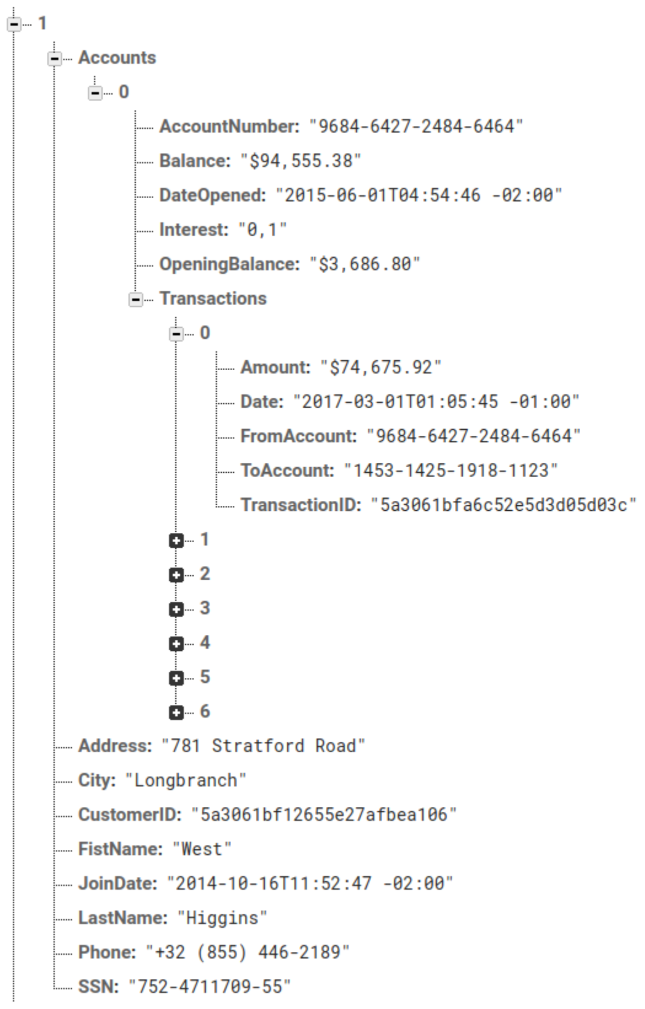
9.3.2 Levels of nodes

On Figure 3 we can observe that our database has six levels of nodes :

— Customer ID : unique reference for a bank’s customer
— Customer’s information including the customer’s account
— Account ID : unique reference for a customer’s account
— Account’s information including transactions
— Transaction ID : unique reference for a transaction between two accounts
— Transaction’s information

9.3.3 Constraints and triggers

To implement the following triggers and constraints we wrote some scripts written in Javascript. Then we will have to deploy those scripts on our Firebase database.



```
1
  Accounts
    0
      AccountNumber: "9684-6427-2484-6464"
      Balance: "$94,555.38"
      DateOpened: "2015-06-01T04:54:46 -02:00"
      Interest: "0,1"
      OpeningBalance: "$3,686.80"
      Transactions
        0
          Amount: "$74,675.92"
          Date: "2017-03-01T01:05:45 -01:00"
          FromAccount: "9684-6427-2484-6464"
          ToAccount: "1453-1425-1918-1123"
          TransactionID: "5a3061bfa6c52e5d3d05d03c"
        1
        2
        3
        4
        5
        6
  Address: "781 Stratford Road"
  City: "Longbranch"
  CustomerID: "5a3061bf12655e27afbea106"
  FistName: "West"
  JoinDate: "2014-10-16T11:52:47 -02:00"
  LastName: "Higgins"
  Phone: "+32 (855) 446-2189"
  SSN: "752-4711709-55"
```

Figure 3: Levels of nodes in our database

— A transaction must be commited within 5 seconds. This is a hard deadline. By doing so, we ensure that one client cannot transfer two times the same money. A transaction consists in a transfer of money from one account to another. In a five seconds lap time, we must ensure the client's balance and deduct the amount. If the constraint is not respected, the operation is aborted.

— The database will have to check that the beneficiary is existing in the database. This means the receving end exists and is a client of the bank. If the constraint is not respected, the operation is aborted.

The two triggers presented above were implented in the same following script.

```
 //Constraint on the beneficiary, check if he is himself a
     bank's customer and update its balance
 exports.beneficiary = functions.database
.ref('/{Id}/Accounts/{AcId}/Transactions/{TrId}')
.onCreate(event => {
    const post = event.data.val()
    if (post.benef){
   return
    }
    console.log("Checking beneficiary " + event.params.TrId)
    console.log(post)
    post.benef = true
    if (checkBenef(post.ToAccount)){
       ref.child("/{Id}/Accounts/{AcId}/AccountNumber")
     .equalTo(accountNumber)
          .once("value",snapshot => {
       const userData = snapshot.val()
   var newAmount = userdata.parent.Balance + post.Amount
   if(Date.now()-event.timestamp<5000){
          return event.data.adminRef.getRoot()
                .ref("/{Id}/Accounts/{AcId}/")
                .update({Balance: newAmount})
       }
  })
    }
    console.log("Aborting transaction. Beneficiary does not
         exist")
    return event.data.adminRef.getRoot()
          .ref("/{Id}/Accounts/{AcId}/Transactions/{TrId}")
          .cancel()
})

 function checkBenef(s){
var accountNumber = s

ref.child("/{Id}/Accounts/{AcId}/AccountNumber")
```

```
    .equalTo(accountNumber)
    .once("value",snapshot => {
     const userData = snapshot.val()
     if (userData){
    console.log("Beneficiary exists!")
    return true
     }
     return false
     })
  }
```

— The date and time of a transaction must be prior to the date of the account's openning ; which must be prior to the date of the client's registering.

```
//date and time of transaction prior to date of account's
    opening
exports.transactionDate = functions.database
.ref('/{Id}/Accounts/{AcId}/Transactions/{TrId}')
.onCreate(event => {
    const post = event.data.val()
    if (post.date){
       return
    }
    console.log("Checking transaction date
        "+event.params.TrId)
    console.log(post)
    post.date=true
    if(checkTransDate(post.Date)){
       return null
    }else{
       console.log("Aborting transaction. Error in date")
       return event.data.adminRef.getRoot()
             .ref("/{Id}/Accounts/{AcId}/Transactions/{TrId}")
             .cancel()
    }

})
 function checkTransDate(trDate, acDate){
     var transDate = trdate
     var openDate = acDate

     ref.child("/{Id}/Accounts/{AcId}/Date")
    .equalTo(transDate)
    .once("value",snapshot => {
        if (transDate < acDate){
          return true
        }
     return false
     });
  }
```

— We don’t allow our customer to have a negative balance on their accounts. If any transaction implies that the balance is becoming negative, the operation is aborted.

```
   //Deduct transaction from account and avoid negative balance
   exports.transactionDate = functions.database
  .ref('/{Id}/Accounts/{AcId}/Transactions/{TrId}')
  .onCreate(event => {
     const post = event.data.val()
     if (post.amount){
        return
     }
     console.log("Checking transaction amount available
          "+event.params.TrId)
     console.log(post)
     post.amount=true
     if(checkTransAmount(post.Amount, event.params.AcId)){
           var newAmount = post.parent.parent.Balance -
               post.Amount
        return event.data.adminRef.getRoot()
              .ref("/{Id}/Accounts/{AcId}")
              .update({Balance: newAmount})
     }else{
        console.log("Aborting transaction. Not enough money on
             account")
        return event.data.adminRef.getRoot()
              .ref("/{Id}/Accounts/{AcId}/Transactions/{TrId}")
              .cancel()
     }

  })

   function checkTransAmount(amount, account){
      var transAmount = amount
      var accountId = account
      ref.child("/"+accountId.parent.parent+"/Accounts/"+accountId+"/Balance")
   .startAt(transAmount)
   .once("value",snapshot => {
      const balance = snapshot.val();
      if (balance){
     console.log("Enough money");
     return true
      }
      return false
});
}
```

— A transaction must never be negative. If the constraint is not respected, the operation is aborted.

```
 //Avoid negative transaction
 exports.transactionNotNeg = functions.database
.ref('/{Id}/Accounts/{AcId}/Transactions/{TrId}')
.onCreate(event => {
    const post = event.data.val()
    if (post.amount){
       return
    }
    console.log("Checking transaction amount available
         "+event.params.TrId)
    console.log(post)
    post.amount=true
    if(post.Amount > 0){
          console.log("Positive transaction");
       return null
    }else{
       console.log("Aborting transaction. Negative
            transaction")
       return event.data.adminRef.getRoot()
             .ref("/{Id}/Accounts/{AcId}/Transactions/{TrId}")
             .cancel()
    }

})
```

— A customer must have a minimun of one account in our bank. If the constraint is not respected, the customer is deleted from the database.

```
 //Check if customer has at least one account
 exports.OneAccount = functions.database
.ref('/{Id}/Accounts/{AcId}')
.onDelete(event => {
    const post = event.data.val()
    if (post.accountId){
   return
    }
    console.log("Checking at least one account " +
         event.params.AcId)
    console.log(post)
    post.accountId = true
    event.data.adminRef.getRoot()
        .ref('/{Id}/Accounts/{AcId}')
        .remove()
    if (post.parent.hasChildren()){
          return null
           }else{
          console.log("Removing client. Has no account anymore")
          return event.data.adminRef.getRoot().ref("/{Id}").remove()
           }
       })
```

— A customer may have at most five accounts. If the constraint is not respected, the new account is not created.

```
        //Check if customer has at most five accounts
        exports.FiveAccount = functions.database
       .ref('/{Id}/Accounts/{AcId}')
       .onCreate(event => {
           const post = event.data.val()
           if (post.accountId){
          return
           }
           console.log("Checking at most five accounts " +
                event.params.AcId)
           console.log(post)
           post.accountId = true
           if (post.parent.numChildren() < 5){
          return null
           }else{
          console.log("Canceling account creation. Already five
               accounts")
          return event.data.adminRef.getRoot()
                     .ref("/{Id}/Accounts/{AcId}")
                     .cancel()
           }
       })
```

— When a customer is openning an account, a minimum of 500 euros must be deposited. If the constraint is not respected, the new account is not created.

```
        //Check min 500 as opening balance
        exports.openingBalance = functions.database
       .ref('/{Id}/Accounts/{AcId}')
       .onCreate(event => {
           const post = event.data.val()
           if (post.accountId){
          return
           }
           console.log("Checking opening balance " +
                event.params.AcId)
           console.log(post)
           post.accountId = true
```

```
         if (post.OpeningBalance() >= 500){
        return null
         }else{
        console.log("Canceling account creation. Not enough money")
        return event.data.adminRef.getRoot()
                   .ref("/{Id}/Accounts/{AcId}")
                   .cancel()
         }
    })
```

— Every minute, it must be checked if there is a new incoming transaction for an account. This is a soft deadline. Thus we can ensure to our clients that their money will be almost immediately available. By doing so, we try getting closer to a real-time banking system.

  This trigger must not be implemented since Firebase has a feature which updates the database as soon as possible (cf. Section 8.3).

9.3.4 Configuration

In order to begin our project, we had to create a google account to get access to firebase. Once the account created, we had to add a new project and enter the project in the web interface to obtain the configuration access. From the web, we have access to the database data, so we were able to import our own data with a JSON file generated with JSON-Generator[16].

To implement the trigger functions, we first had to install the firebase command line interface (CLI) through the console. Then we needed to login on the CLI and specify which project we wanted to alter.

Once we entered the said project, 5 options are available:
— Database
— Firestore
— Functions
— Hosting
— Storage
Only two were useful for our example, database and functions. These options allowed us to set rules on the access to the database and to configure the triggers.

The functions feature created a javascript file in which we had to define our triggers before deploying them into the project.

10 Conclusion

This project gave us the oportunity to understand the difference between a temporal and a real-time database system. Their names suggest that those systems are similar and indeed both are linked to time. But we have learned that they are related to two different aspects of time.

The usage of RTDBS may touch a lot of different sectors and are more present in our daily life than what we could think. For example, it is used in banking system as we have seen but also in position tracking system, messaging system, stock exchange market, learning websites as Duolingo, commercial aviation, etc.

Even if Firebase is popular and well documented, we encoutered a lot of difficulties to configurate the database. This is due to big differences between Firebase and database systems we have learned up to now and the fact that Firebase requeries up-to-date version of modules. Furthermore to make our application work, Firebase asked to add a credit card to use the Google App Engine but we could not afford it, thus we were unable to try our system.

References

[1] James Martin. *Programming real-time computer systems.* 1965

[2] Saravanakumar Kandasamy. *Real-time database definition* [online]. 2016 [Retrieved 02 October 2017]
Available on : http://www.exploredatabase.com/2016/03/real-time-database-definition.html

[3] Paul A Francis. *An Introduction to Real-Time Stock Market Data Processing* [online]. 2013 [Retrieved 04 October 2017]
Available on : http://www.codeproject.com/Articles/553206/An-Introduction-to-Real-Time-Stock-Market-Data-Pro

[4] Jan Lindström. *Real Time Database System* [online]. 2008 [Retrieved 04 October 2017]
Available on : https://www.cs.helsinki.fi/u/jplindst/papers/rtds.pdf

[5] Wikipedia, the free encyclopedia. *Energy efficiency in transport*[online]. 2017 [Retrieved 08 November 2017]
Available on : https://en.wikipedia.org/wiki/Energy_efficiency_in_transport#Trains

[6] Wikipedia, the free encyclopedia. *BTree*[online]. 2017 [Retrieved 16 November 2017]
Available on : https://en.wikipedia.org/wiki/B-tree

[7] Daniel Salengue. *What is schema-free DB ?*[online]. 2016 [Retrieved 16 November 2017]
Available on : https://www.quora.com/What-is-schema-free-DB

[8] DB-Engines. *System Properties Comparison CouchDB vs. Firebase Realtime Database vs. MariaDB vs. RethinkDB*[online]. 2017 [Retrieved 16 November 2017]
Available on : https://db-engines.com/en/system/CouchDB%3BFirebase+Realtime+Database%3BMariaDB%3BRethinkDB

[9] Grafikart. *Qu'est ce que le NoSQL* [online]. 2016 [Retrieved 21 November 2017]
Available on : https://www.grafikart.fr/blog/sql-nosql

[10] Sang H. Son. *Real-time database systems and Data Services : Issues and challenges* [online]. 2011 [Retrieved 4 December 2017]

Available on : https://www.slideserve.com/jaden/real-time-database-systems-and-data-services-issues-and-challenges

[11] *Firebase Legacy Documentation* [online]. 2016 [Retrieved 15 December 2017]
Available on : https://www.firebase.com/docs/

[12] GOOGLE. *Firebase Documentation* [online]. 2017 [Retrieved 3 December 2017]
Available on : https://firebase.google.com/docs

[13] NIPUNA HEWAMADDUMAGE THILINA, DINUSHIKA RATHNAYAKA, NILSHANI JAYAKODI, YANA YONITHA. *Real-time databases* [online]. 2014 [Retrieved 4 December 2017]
Available on : https://fr.slideshare.net/nipunahewamadduma/real-time-databases

[14] ACCENTURE BANK. *Real-time payments for real-time banking* [online]. 2015 [Retrieved 6 December 2017]
Available on : https://www.accenture.com/t00010101T000000__w__/de-de/_acnmedia/Accenture/Conversion-Assets/DotCom/Documents/Global/PDF/Dualpub_22/Accenture-Banking-Realtime-Payments-Realtime-Bank.pdf

[15] AMBADE SHRUTI DINKAR, S.A SHAIKH. *Design and Implementation Of Vehicle Tracking System Using GPS* [online]. 2011 [Retrieved 6 December 2017]
Available on : http://www.iiste.org/Journals/index.php/JIEA/article/download/798/703

[16] VAZHA OMANASHVILI. *JSON Generator* [online]. [Accessed 7 December 2017] Available on : https://www.json-generator.com/

[17] FRANK VAN PUFFELEN. *Have you met the real time database* [online]. 2016 [Retrieved 16 December 2017]
Available on : https://firebase.googleblog.com/2016/07/have-you-met-realtime-database.html