Main-memory database VS Traditional database

Abstract

There has been a surge of new databases in recent years. Applications today create a higher demand on database performance than ever before. Main-memory databases have come into the market quite recently and they are just now catching a lot of interest from many different directions.

Main-memory databases are a type of database that stores all of its data in the primary memory. They provide a big increase in performance to a lot of different applications. This work evaluates the difference in performance between two chosen candidates. To represent main memory databases we chose VoltDB and to represent traditional databases we chose MySQL.

We have performed several tests on those two databases. We point out differences in functionality, performance and design choices. We want to create a reference where anyone that considers changing from a traditional database to a main memory database, can find support for their decision. What are the advantages and what are the disadvantages of using a main-memory database, and when should we switch from our old database to a newer technology.

Authors:

Marcus Rehn
Emil Sunesson

Supervisor:

Mats-Ola Landbris
Contents

Abstract ................................................................................................................................. 1

1. Introduction ...................................................................................................................... 4
  1.1 Terminology .................................................................................................................. 5
      1.1.1 DBMS .................................................................................................................. 5
      1.1.2 RDBMS ............................................................................................................... 5
      1.1.3 SQL ..................................................................................................................... 5
      1.1.4 Index ................................................................................................................... 5
      1.1.5 Primary key ......................................................................................................... 5
      1.1.6 Surrogate key ..................................................................................................... 5
      1.1.7 AUTO_INCREMENT ........................................................................................... 5
      1.1.8 Foreign key ......................................................................................................... 5
      1.1.9 OLAP .................................................................................................................. 6
      1.1.10 OLTP ............................................................................................................... 6
      1.1.11 ACID ............................................................................................................... 6
  1.2 Background .................................................................................................................. 7
  1.3 Goal and purpose ........................................................................................................ 8

2 Method ............................................................................................................................... 9

3 Experiment design ......................................................................................................... 10
  3.1 Hardware .................................................................................................................... 10
  3.2 Setup .......................................................................................................................... 10
  3.3 Tests .......................................................................................................................... 10
  3.4 Tables ........................................................................................................................ 11
  3.5 VoltDB Implementation ............................................................................................ 11
  3.6 MySQL implementation .......................................................................................... 12
  3.7 Test cases ................................................................................................................... 13
      3.7.1 Inserts ............................................................................................................... 13
      3.7.2 AVG .................................................................................................................. 14
      3.7.3 SUM ................................................................................................................... 14

4 Result ............................................................................................................................... 15
  4.1 Shared tests ................................................................................................................ 15
      TABLE 1 ....................................................................................................................... 15
  4.2 VoltDB test results ..................................................................................................... 16
I. Introduction

In recent years there has been a surge of new database management systems (DBMS). The traditional databases like MySQL is still used widely but they lack in performance. There is several alternatives on the market today in the form of nosql or refined forms of relational databases. One of the best ways to increase performance is to use a main-memory database. In a main-memory database the entire database is stored in the primary memory and not on disk. The biggest advantage with storing the entire database in memory is that it eliminates the need for disk I/O operations and this will dramatically increase the performance of the database. The concept of storing the entire set of data in main memory has been around for many years, but it has only been economically justifiable in recent years, because of the drastic decrease in memory cost.

Our main method for evaluating will be running tests on two different database implementations. One traditional database MySQL and one main-memory database VoltDB. VoltDB is a main-memory database that focuses on speed, throughput and scalability for OLTP (online-transactional processing). We will simulate a simple online shop where we run our different use cases to measure the performance of the databases.

This study will evaluate main-memory database advantages and possible disadvantages such as lack in functionality. We will also study other ways to increase performance for example column-based databases. The main focus in this study will be to try to determine when switching from a traditional database to a main-memory database is beneficial.

There are two different types of workload for databases, OLTP and OLAP. OLTP or online transactional processing consists of basic selects, deletes and updates. OLAP online analytical processing is analyzing data like joins and aggregations and is on large sets of data. OLAP consumes a lot more resources. Analytical processing is usually run over terabytes of data and is thus more costly to run [2] [6].

Both VoltDB and MySQL use ACID transactions. ACID stands for:

Atomicity: Either all operations in a transaction complete or no operations complete.
Consistency: The state of the database must be consistent at the start and the end of the transaction.
Isolation: Transactions behaves as if they were the only running operation on the database.
Durability: When a transaction is complete it will not be reversed [5].
1.1 Terminology

1.1.1 DBMS
A database management system (DBMS) is software that manages, organizes and manipulates data.[15]

1.1.2 RDBMS
Relational database management systems are a type of DBMS that follows the relational database approach. One of the main functions of the relational approach is to make sure that data is not to be duplicated. Instead of writing the same piece of information multiple times, only the information of where to find it is kept, typically a foreign key is used to point to it [15].

1.1.3 SQL
SQL is the acronym for structured query language. The language is used for fetching, organizing and manipulating data in several relational database management systems [16].

1.1.4 Index
Indexes are used to speed up searches. Two common indexes are INDEX and KEY. When using indexes on a field, a specification can be made on how much of the field is going to be indexed. Indexes work from left to right, so searches beginning with a wild-card will not be positively affected [9].

1.1.5 Primary key
In database management systems a primary key is a property that can be assigned to a single field, or a group of fields in a table. Using this property forces the value of the key of each row to be unique. The primary key is also a type of index and it is used to identify each specific row [9].

1.1.6 Surrogate key
In some cases when there are no naturally unique fields in a table, a surrogate can be used instead. This is achieved by adding an extra field to the table and assigning it the role of primary key [17].

1.1.7 AUTO_INCREMENT
When using a surrogate key, that is a number, you can use AUTO_INCREMENT to automatically assign a value to the key. Each new row gets a value increased by 1 [9].

1.1.8 Foreign key
Much like the primary key, the foreign key is used to identify a specific row in a table. But in the case of a foreign key, the key identifies a row in another table. To identify the row the foreign key references a field in the other table, with a value that must exist in the other table. Foreign keys can also be constructed using several fields, as long as they correlate with fields in the other table [16].
1.1.9 OLAP
OLAP is an acronym for online analytical processing. A database with focus on OLAP has a focus on performing analytical queries quickly [2].

1.1.10 OLTP
OLTP is an acronym for online transactional processing. A database with an OLTP focus aims primarily to perform quick transactions, such as insertions and selections [2].

1.1.11 ACID
Atomicity: Either all operations in a transaction must complete or no operations will complete.
Consistency: The state of the database has to be consistent at the start and the end of the transaction.
Isolation: Transactions behaves as if they were the only running operation on the database.
Durability: When a transaction is complete it will not be reversed [5].
1.2 Background

Main-memory databases are a relatively new type of databases. The demand on database performance has been increasing, many applications running today has high demands on their database that was impossible just a few years ago [7]. For huge databases like the database of Facebook the standard relational databases is not fast enough to meet the demands from its users. Main-memory databases use the primary memory to store its data this reduces the I/O operations to the disk. There will still be some I/O operations because of the database needing to save its data to disk to prevent power failure from wiping the entire database.

The databases today need to handle huge amount of data in real-time environments. The standard relational SQL databases are easy to use and the technique is well documented. They have a good structure and is easy to make changes in, and is used everywhere. They are however slower and use a structure that does not suit all applications.

The problem we would like to address in this study is when you should switch from the standard well documented technology to the newer technique with higher performance. There is currently dozens of different databases that use main-memory techniques. The performance of databases does not rely solely on the actual speed of the database. A big part of how effective a database is comes from how you use it. Different databases are good at different things some specialize in analyzing values using joins and other techniques to make advanced analysis on data. Others use only reads and writes. Different types of databases focus on optimizing different utilities. Some focus on being good at reading and changing data, some on analyzing existing data.

This study’s purpose is to give insight to the difference between a traditional database and a main-memory database. How big of a performance difference you can expect?
1.3 Goal and purpose

Main-memory databases have become available fairly recently and then there is a lot of different companies and people interested in the potential and advantages of main-memory databases. We want to answer the question of when a main-memory database outperforms traditional databases and in which areas?

We want to conduct tests where we use a main-memory database and a traditional database to create results which points out differences in performance and what advantages and disadvantages can be associated with main-memory databases.

When should you switch from a traditional database to an in-memory database, and how big is the difference performance wise.

Our hypothesis is:
We believe that the in-memory database will be faster in all aspects the question will be if the extra performance will be worth the extra cost in hardware.
2 Method

In order to prove or disprove our hypothesis we want to conduct several tests on two different databases. We make an implementation of the two different databases, MySQL and VoltDB. In this case MySQL is representing the traditional database and VoltDB represents the main-memory database. The most important part in our work is to show how well the two databases perform; using relevant tests to point out what the advantages and disadvantages of the databases of choice are.

Our method to test the databases will be setting up the two databases on the same server. Then conduct the same tests on them to be able to do a fair evaluation. We will look on a number of key features in the databases such as select, insert and delete. That will be the main focus of the tests. The workload we are using is called OLTP or online-transactional processing and will be the workload that we will evaluate in this work [2].
3 Experiment design

3.1 Hardware

Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz
24GiB RAM DIMM DDR3 Synchronous 1067 MHz (0.9 ns)
500GiB Seagate ST500DM002-1BC14 ATA Disk
OS: Ubuntu server 12.4.

3.2 Setup

When you make a VoltDB implementation you create a client application which you run all your procedures from. We choose to do it in a similar way for MySQL.

Client applications have been implemented in java. Database syntax is standard SQL. Volt does not have the complete SQL language implemented and is limited in several areas because of this. VoltDB and MySQL can use several languages for their client implementation. We choose java for our implementation because it is the native language for VoltDB.

VoltDB is a clusterable main-memory database that provides ACID transactions as well as high speed performance. VoltDB has quite recently made it into the market. They focus on OLTP transactions. VoltDB is a database that focuses entirely on maximizing velocity.

VoltDB can be downloaded for free from www.voltdb.com and their community version is licensed under the GNU General Public License version 3. We use the community edition in our tests.

3.3 Tests

The goal with the study is to show an adequate result for database performance. The tests we chose are the following [1].

Insert: Basic insert used to add items to the table. Important to test as it is a basic command.
Delete: Basic deletion used to delete items from the table. Important to test as it is a basic command.
Select: Basic command that allows you to retrieve items from the table. The most important command to test as it is very commonly used command in applications it can also be expensive.
Sum: A function that summarizes a set of values given a column in a table.
Avg: A function that calculates the average value of a set amount of values.
3.4 Tables
We choose the following tables to perform our tests on [A 3].
Customer,
Item,
Orders

The three tables simulate a simple web shop. Customer holds information about the customer, Item information about item and if an order is made you combine the id of customer and item and create a new row in order.

The following ER-diagram shows the structure of the databases. However, as VoltDB as of now is incapable of using foreign key constraints, we decided to leave them out of most of the tests to make sure that inserts on the two databases are as similar as possible.

![ER diagram showing the tables and their relations to each other. Note that in the VoltDB version of the item table, the column called “description” was called “text”. The properties remain the same and can be seen in the appendix section A.3.](image)

3.5 VoltDB Implementation
The main feature in VoltDB that allow it to run so fast is single partitioned tables. You want to partition the table on the column that is used in the most used procedure. If you want to select a value from customer on id, you want to partition on id. When you single partition a table you hash the partition attribute (in this case id) and create an object of the row, hashed on the partition attribute [4]. So basically what happens is that you create a hash table with the partition column as key and the row as value making searches take constant time on the partition value.

However in order to speed up the time of the searches in MySQL, we decided to put use an index on name. We decided to run a test on MySQL with foreign keys and one without indexing the name column, to see what effects it had on the insertion time.

The size of the tables in VoltDB is as follows:
Customer: 70 Byte
Item: 212 Byte
Orders: 12 Byte
Total: 212 + 70 + 12 = 294
1 million rows: 294MB
10 million rows: 2.94GB
15 million rows: 4.41GB

Queries in VoltDB is always executed as procedures the reason being optimization. Procedures in VoltDB get precompiled and are therefore much more effective than ad hoc queries. The procedures are then called from the client application which is written in java [Appendix 1.2]. You can use ad hoc queries in VoltDB but it is not recommended for anything other than testing.

The tests are implemented into procedures and are then called from the client application.

```java
//Code for calling the procedures that inserts the testdata into the tables
for(int i = 1; i < numberOfRows; i++)
{
    int iVal = i % 1000000;
    Clientvolt.callProcedure(new CustomerCallback(), "Customer", i,
                            customerNameArr[iVal], customerBirthArr[iVal],
                            customerEmailArr[iVal]);
    Clientvolt.callProcedure(new ItemCallback(), "Item", i,
                             itemNameArr[iVal], itemTextArr[iVal], itemPriceArr[iVal]);
    Clientvolt.callProcedure(new OrderCallback(), "Orders", i,
                             ordersIdCustomerArr[iVal], ordersIdItemArr[iVal]);
    if(i % 1000000 == 0)
    {
        System.out.println(iVal);
    }
}
```

The tests are run on 1 million, 5 million, 10 million, and 15 million rows.

### 3.6 MySQL implementation

The tests run for MySQL is set up in a similar way where we call the MySQL database from a java application. Because of system limitations we couldn't run 10 and 15 million rows for MySQL.

Running the test on the dataset of one million was reasonably fast, when we ran the test on 5 million rows per table it became obvious that going up to 10 or even 15 million would be impossible due to time constraints. Instead we ran the test on 1 million, 2 million, 3 million, 4 million and 5 million rows of data.
To give MySQL a fair chance we followed the instructions on InnoDB performance on their website [some number]. According to the site, the best way to improve the speed was to use indexes [11].

We wanted to see how different functionalities would affect the speed of MySQL. Would indexing speed up searches and would it slow down insertion? Would foreign key constraints and auto increments affect the speed of insertion? However we did partition the tables, as what we actually to measure was the speed in relation to the amount of data.

To improve performance even more we changed the InnoDB buffer read to 19327352832 so that MySQL could store the tables in its ram, improving the search speeds [10].

3.7 Test cases

3.7.1 Inserts

Inserting large amounts of data in volt is best done using asynchronous procedure calls. There are two different ways to call procedures in volt, synchronous and asynchronous.

Synchronous procedure calls will call the procedure and wait for a response which will be returned from the call.

Synchronous

```java
final ClientResponse response = Clientvolt.callProcedure("Customer", i, customerNameArr[iVal],
            customerBirthArr[iVal], customerEmailArr[iVal]);
```

Asynchronous calls get queued up in VoltDB (up to 1000 calls in queue) and the response is sent to the callback function. The callback function is the first parameter in an asynchronous procedure call [8].

Asynchronous

```java
Clientvolt.callProcedure(new CustomerCallback(), "Customer", i, customerNameArr[iVal],
            customerBirthArr[iVal], customerEmailArr[iVal]);
```

The big advantage with asynchronous calls is that the database can start with another transaction straight away instead of having to wait for the response to go back to the application and then for the application to make another call.

The inserts into MySQL were done with 10 000 rows at a time. In order for MySQL to accept large amounts of data in a package we increased the value of the `max_allowed_packet` to 1073741824. We used multi inserts purely for performance reasons.
3.7.2 **AVG**

Avg is a function that returns the average value of a certain column, in this case the price column of the ITEM table.

3.7.3 **SUM**

The sum function calculates the sum of a column, in this case the price column of the ITEM table.

Selects are done 100 times and done by name that matches an id that matches and an id that does not match.
4 Result
We use tables to present the result of our tests. The tests are color-coded and grouped by the type of test they belong to. Insertions are green, selects are blue, deletes are red, tests using AVG are yellow and tests on sum are purple.

4.1 Shared tests
The inserts were done differently on the two machines due to restrictions in the java API for MySQL not allowing, asynchronous calls. As we still wanted to give MySQL a fair chance we let MySQL insert 10 000 rows at a time in the cases of inserting customer and item. When inserting orders, MySQL inserted 1000000 rows at once. The customer tables differ in that the customer table on MySQL had an index on the name column.

The select statements selecting 100 rows by searching for id was done in the same way on both machines.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of rows</th>
<th>Time in seconds VoltDB</th>
<th>Time in seconds MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert customer</td>
<td>1000000</td>
<td>13.651</td>
<td>119.152</td>
</tr>
<tr>
<td>Insert customer</td>
<td>2000000</td>
<td>24.416</td>
<td>308.352</td>
</tr>
<tr>
<td>Insert customer</td>
<td>3000000</td>
<td>36.806</td>
<td>641.861</td>
</tr>
<tr>
<td>Insert customer</td>
<td>4000000</td>
<td>49.135</td>
<td>983.754</td>
</tr>
<tr>
<td>Insert customer</td>
<td>5000000</td>
<td>61.106</td>
<td>1419.274</td>
</tr>
<tr>
<td>Insert item</td>
<td>1000000</td>
<td>13.401</td>
<td>120.104</td>
</tr>
<tr>
<td>Insert item</td>
<td>2000000</td>
<td>26.978</td>
<td>176.296</td>
</tr>
<tr>
<td>Insert item</td>
<td>3000000</td>
<td>39.818</td>
<td>375.442</td>
</tr>
<tr>
<td>Insert item</td>
<td>4000000</td>
<td>53.921</td>
<td>433.722</td>
</tr>
<tr>
<td>Insert item</td>
<td>5000000</td>
<td>67.710</td>
<td>442.656</td>
</tr>
<tr>
<td>Insert orders</td>
<td>1000000</td>
<td>12.784</td>
<td>26.463</td>
</tr>
<tr>
<td>Insert orders</td>
<td>2000000</td>
<td>21.573</td>
<td>46.401</td>
</tr>
<tr>
<td>Insert orders</td>
<td>3000000</td>
<td>33.352</td>
<td>87.547</td>
</tr>
<tr>
<td>Insert orders</td>
<td>4000000</td>
<td>50.728</td>
<td>107.733</td>
</tr>
<tr>
<td>Insert orders</td>
<td>5000000</td>
<td>56.95</td>
<td>116.32</td>
</tr>
<tr>
<td>Select 100 rows by id from customer</td>
<td>1000000</td>
<td>0.003</td>
<td>0.22</td>
</tr>
</tbody>
</table>
4.2 VoltDB test results

TABLE 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of rows</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select 100 rows by id from customer</td>
<td>2000000</td>
<td>0.004</td>
</tr>
<tr>
<td>Select 100 rows by id from customer</td>
<td>3000000</td>
<td>0.004</td>
</tr>
<tr>
<td>Select 100 rows by id from customer</td>
<td>4000000</td>
<td>0.004</td>
</tr>
<tr>
<td>Select 100 rows by id from customer</td>
<td>5000000</td>
<td>0.003</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of rows</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert all tables 1000000</td>
<td>1000000</td>
<td>40.578</td>
</tr>
<tr>
<td>Insert all tables 5000000</td>
<td>5000000</td>
<td>208.203</td>
</tr>
<tr>
<td>Insert all tables 10000000</td>
<td>10000000</td>
<td>416.555</td>
</tr>
<tr>
<td>Insert all tables 15000000</td>
<td>15000000</td>
<td>637.740</td>
</tr>
<tr>
<td>Select from customer 1000000</td>
<td>1000000</td>
<td>12.494</td>
</tr>
<tr>
<td>Select from customer 5000000</td>
<td>5000000</td>
<td>65.972</td>
</tr>
<tr>
<td>Select from customer 10000000</td>
<td>10000000</td>
<td>127.8</td>
</tr>
<tr>
<td>Select from customer 15000000</td>
<td>15000000</td>
<td>192</td>
</tr>
<tr>
<td>Select elements that do not exist</td>
<td>1000000</td>
<td>6.405</td>
</tr>
<tr>
<td>Select elements that do not exist</td>
<td>5000000</td>
<td>41.515</td>
</tr>
<tr>
<td>Select elements that do not exist</td>
<td>1000000</td>
<td>83.221</td>
</tr>
<tr>
<td>Select elements that do not exist</td>
<td>15000000</td>
<td>126.805</td>
</tr>
<tr>
<td>SUM() on item price</td>
<td>1000000</td>
<td>0.189</td>
</tr>
<tr>
<td>SUM() on item price</td>
<td>5000000</td>
<td>0.763</td>
</tr>
<tr>
<td>SUM() on item price</td>
<td>10000000</td>
<td>1.47</td>
</tr>
<tr>
<td>SUM() on item price</td>
<td>15000000</td>
<td>2.218</td>
</tr>
<tr>
<td>AVG() on item price</td>
<td>1000000</td>
<td>0.246</td>
</tr>
<tr>
<td>AVG() on item price</td>
<td>5000000</td>
<td>-</td>
</tr>
<tr>
<td>AVG() on item price</td>
<td>10000000</td>
<td>-</td>
</tr>
<tr>
<td>AVG() on item price</td>
<td>15000000</td>
<td>-</td>
</tr>
<tr>
<td>Delete on all 1000000</td>
<td>1000000</td>
<td>37.672</td>
</tr>
<tr>
<td>Delete on all 5000000</td>
<td>5000000</td>
<td>190.759</td>
</tr>
<tr>
<td>Delete on all 10000000</td>
<td>10000000</td>
<td>359.266</td>
</tr>
<tr>
<td>Delete on all 15000000</td>
<td>15000000</td>
<td>569.965</td>
</tr>
</tbody>
</table>
### Summary VoltDB tests

During both insertion and selection the time increase appear to be fairly close to linear with the increase if data. The sum function, that calculates the sum of every entry of the specified column (in this case price), failed to complete on the tables of 5 million rows and more. Deletion of every single element was also close to linear in relation the amount of data.

#### 4.3 MySQL

### Table 3

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select 100 rows by name</td>
<td>1000000</td>
<td>0.221</td>
</tr>
<tr>
<td>Select 100 rows by name</td>
<td>2000000</td>
<td>0.168</td>
</tr>
<tr>
<td>Select 100 rows by name</td>
<td>3000000</td>
<td>0.367</td>
</tr>
<tr>
<td>Select 100 rows by name</td>
<td>4000000</td>
<td>0.248</td>
</tr>
<tr>
<td>Select 100 rows by name</td>
<td>5000000</td>
<td>0.176</td>
</tr>
<tr>
<td>Select 100 rows by non-existing id</td>
<td>1000000</td>
<td>22.106</td>
</tr>
<tr>
<td>Operation</td>
<td>Time (ms)</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>Select 100 rows by non-existing id</td>
<td>2000000</td>
<td></td>
</tr>
<tr>
<td>Select 100 rows by non-existing id</td>
<td>3000000</td>
<td></td>
</tr>
<tr>
<td>Select 100 rows by non-existing id</td>
<td>4000000</td>
<td></td>
</tr>
<tr>
<td>Select 100 rows by non-existing id</td>
<td>5000000</td>
<td></td>
</tr>
<tr>
<td>AVG on item price</td>
<td>100000</td>
<td></td>
</tr>
<tr>
<td>AVG on item price</td>
<td>2000000</td>
<td></td>
</tr>
<tr>
<td>AVG on item price</td>
<td>3000000</td>
<td></td>
</tr>
<tr>
<td>AVG on item price</td>
<td>4000000</td>
<td></td>
</tr>
<tr>
<td>AVG on item price</td>
<td>5000000</td>
<td></td>
</tr>
<tr>
<td>SUM on item price</td>
<td>100000</td>
<td></td>
</tr>
<tr>
<td>SUM on item price</td>
<td>2000000</td>
<td></td>
</tr>
<tr>
<td>SUM on item price</td>
<td>3000000</td>
<td></td>
</tr>
<tr>
<td>SUM on item price</td>
<td>4000000</td>
<td></td>
</tr>
<tr>
<td>SUM on item price</td>
<td>5000000</td>
<td></td>
</tr>
<tr>
<td>Delete 100 rows from customer</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>Delete 100 rows from customer</td>
<td>2000000</td>
<td></td>
</tr>
<tr>
<td>Delete 100 rows from customer</td>
<td>3000000</td>
<td></td>
</tr>
<tr>
<td>Delete 100 rows from customer</td>
<td>4000000</td>
<td></td>
</tr>
<tr>
<td>Delete 100 rows from customer</td>
<td>5000000</td>
<td></td>
</tr>
<tr>
<td>Insert orders with relations</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>Insert customer without index</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>Insert customer with auto increment</td>
<td>1000000</td>
<td></td>
</tr>
</tbody>
</table>

The insert times appears to increase almost linearly in proportion to the data inserted into them in the cases of ITEM and ORDERS. In CUSTOMER the result appears to be that it takes an increasing amount of time, disproportional to the growth of the table. When inserting the data we found that VoltDB was faster than MySQL when using asynchronous calls. When VoltDB used synchronous calls it was considerably slower than when using asynchronous calls.
5 Discussion

5.1 Graphs

The time taken inserting 1000000 rows into the CUSTOMER table is affected by the method of insertion. Inserting the rows asynchronously one by one on VoltDB was much faster than inserting 10000 rows at a time on MySQL. Synchronous insertion one by one took the longest amount of time by far.

Running the tests on one through five million rows of data comparing insertion of 10 000 rows at the time and asynchronous calls of VoltDB were in clear favor of VoltDB.

In the cases of searching for 100 elements in the set of one through five millions of rows, VoltDB was the clear winner as well.

Looking at the results we can see a clear advantage in using VoltDB. In both writes and reads a significant difference can be seen in performance.

5.2 Writes

Values from first row in table 1 in results.

Asynchronous writes (VoltDB, MySQL has no built in way to run transactions asynchronously)
VoltDB inserting 1 million rows into the customer table: 13.651 seconds.
MySQL inserting 1 million rows into the customer table: 119.152 seconds.
VoltDB is approximately 9 times faster.

VoltDB use asynchronous inserts, the advantage with asynchronous inserts is that the database does not have to wait for the client application to receive the answer. This is however nullified by the bulk inserts we used with MySQL which inserts 10000 rows at the
time reducing the wait time for the database to almost nothing for MySQL, as well as for VoltDB. We believe the result is accurate and this means that the results points hugely in favor of VoltDB.

An interesting thing about the insertion times on the customer and item table grew differently in proportion to the amounts of data. The amount of data in the item table was larger than the amount in the customer table and yet the insertion times rapidly grew longer than the insertion of items. This is due to the index of the name column in the customer table.

5.2 Reads
Values can be found in table 1 and table 2

Asynchronous reads (VoltDB, MySQL has no built in way to run transactions asynchronously)
VoltDB select 1000000 rows: 12.494 seconds (100 is 10000 times less than 1000000)
$$\frac{12.494}{10000} = 0.0012$$
MySQL select 100 rows: 0.22 seconds
VoltDB is approximately 180 times faster.

Synchronous reads
VoltDB select 100 rows from 1000000 rows in customer: 0.0032 seconds.
MySQL select 100 rows from 1000000 rows in customer: 0.22 seconds.
VoltDB is approximately 70 times faster.

In order to demonstrate how much MySQL in fact benefitted from indexes in terms of reads we recreated the customer table without and filled it with 1000000 rows and ran the search of 100 names once more. The difference was 0.221 with the index and 48,348 without it, more than a 48 second difference.

5.3 SUM
Values can be found in table 2 and table 3

Summarize 1000000 rows in table items on price
VoltDB: 0.189 seconds.
MySQL: 0.157 seconds.
MySQL is 1.2 times faster

Summarize 5000000 rows in table items on price
VoltDB: 0.763
MySQL: 0.836
VoltDB is 1.1 times faster.

When the database does not have to search through its data, it looks like MySQL can perform on the same level as volt. It even outperformed volt on 100000 rows.

5.4 AVG

VoltDB was slower than MySQL in the case of 100000 million rows. In this case VoltDB finished in 0.246 and MySQL finished in 0.163. VoltDB failed to finish the function in the rest of its cases, leaving MySQL the clear victor in this test.

5.5 Summary

We can clearly see that VoltDB has outperformed MySQL in the large majority of the tests, especially when the tests where OLTP focused like insert and select. But what are the possible drawbacks of using VoltDB?

First of all, if you need to do analytical work on your database (OLAP) VoltDB is not suitable, as they focus on OLTP. VoltDB will be most effective when your goal is to get as high velocity as possible and not huge amounts of data. VoltDB is not suitable to use on big data banks and that is not their use case. The hardware required is more expensive VoltDB use primary memory to store its data while databases more suitable for storage use secondary memory (hard drives).

It might be hard to migrate from your old database straight into VoltDB. VoltDB does not have all functionality that MySQL have. VoltDB does not have auto increment, foreign key constraint and does not support the complete SQL syntax for example join is not supported (you can still use joins without the keyword join to some extent).

When using VoltDB everything needs to be made into a procedure, ad hoc queries is not supported, you can use ad hoc queries but they don't benefit from what makes VoltDB fast, they are only there for test purposes.

When we compare the two different databases we can clearly see that VoltDB is faster. But how much of that improvement is due to primary memory versus secondary memory, and how much of the difference is due to VoltDB implementation versus MySQL implementation? If you do not take into account the database but only the actual memory access primary memory is around 7 times faster than secondary memory for reads. VoltDB is thus faster than those 7 times that the primary memory should provide in increased speed. [13]
6 Conclusions

This study has investigated the difference in performance and functionality of a traditional database (MySQL) and a main-memory database (VoltDB) the tests has been on an OLTP workload. The main method we have been using is running tests and measuring performance on the two different databases. How the tests have been setup is described in the experiment design chapter. The source code can be found in appendix for more detailed information.

Deciding a winner may look like a simple task when looking at the results of our tests, and VoltDB sure is vastly faster than MySQL. So if velocity is a huge priority, then VoltDB may be the database most suitable for your application. However VoltDB requires more expensive hardware and lacks in functionality. Auto increment does not exist, some standard SQL statements cannot be run because they have not been implemented as of today and you can’t create foreign key constraints. Lacking these functions may be a big enough problem for you.

Something else that speaks against VoltDB is the fact that you have to use the java API provided to create procedures. Procedures that must be written as running ad-hoc queries is something that cannot be done within an application. Even if your application is going to be written in another language you must use java for this initial step.

Setting up the databases didn’t take very long on either of the servers, not much time spent on installing either. For obvious reasons we cannot speak about what database is harder so set up when it comes to larger and more complex systems with multiple computers working together. Although there was some tweaking to be done to MySQL in order to raise performance through settings and variables. VoltDB is in our minds the clear winner when it comes to having a simple manual to follow on setup for performance.

Coming back to the original question; when should you switch from your traditional database to a main-memory database? The answer as we see it is when the benefits in performance outweigh the cost of hardware and the cost of the time spent migration and possible additional application logic to make up for the sometimes lack of functionality.

Our opinions on the matter may be skewed too much to the specific databases we used in our tests, perhaps other traditional databases can perform better than MySQL and perhaps other main-memory databases have a more functionality than VoltDB has. But with our tests the VoltDB is clearly the better database.
7 Future works

There are many different databases to choose from when conducting tests like these. There are SQL databases similar MySQL, such as SQL server or oracle.

A study could be made comparing SQL, NoSQL and NewSQL. Or one could compare different in-memory database to each other, focusing on speed or functionality.

A study similar to this could be made comparing the speed and the functionality of VoltDB to SAP HANA, an in-memory database that is also able to use SQL.

As VoltDB can be used with JSON instead of SQL, speed, ease of use and functionality could be compared to MongoDB or another NoSQL database.

There are a lot of tests that has not been run on VoltDB in this study, performing test just to evaluate VoltDB could be a whole other study.
8 References

Rapport:
[1] J+-Tree: A New Index Structure in Main Memory,
Hua Luan1,2, Xiaoyong Du1,2, Shan Wang1,2, Yongzhi Ni1,2, and Qiming Chen3
1 School of Information, Renmin University of China, Beijing, China
2 Key Laboratory of Data Engineering and Knowledge Engineering, Renmin
University of China, MOE, Beijing, China
3 HP Labs China, Beijing, China
Springer-Verlag Berlin Heidelberg 2007

[2] An In-Depth Analysis of Data Aggregation Cost Factors in a Columnar In-Memory
Database,
Stephan Müller, Hasso-Plattner-Institut, University of Potsdam
Hasso Plattner, Hasso-Plattner-Institut, University of Potsdam
DOLAP '12 Proceedings of the fifteenth international workshop on Data warehousing and
OLAP
Pages 65-72
ACM New York, NY, USA ©2012

[3] VoltDB Documentation Chapter 4. Sizing Memory,
http://VoltDB.com/docs/PlanningGuide/ChapMemoryRecs.php
2008-2013 VoltDB, Inc.
Visited 01 June 2013

[4] Understanding VoltDB Data Partitioning,
Ryan Betts,
June 27th, 2011,
VoltDB,
Visited 01 June 2013.

DAN PRITCHETT
© 2008 ACM 1542-7730/08/0500

[6] IN-MEMORY DATABASES: An industry perspective,
A. Pizzarello, Bull Worldwide Information Systems Phoenix, AZ 85066,
F. Golshani Arizona State University Tempe, AZ 85287-5406,
Research Issues on Data Engineering, 1992: Transaction and Query Processing, Second
International Workshop

[7] Hybrid In-Memory and On-Disk Tables for Speeding-Up Table Accesses
Joan Guisado-Gámez, Antoni Wolski, Calisto Zuzarte, Josep-Lluís Larriba-Pey, Victor Muntés-Mulero
Database and Expert Systems Applications
Lecture Notes in Computer Science Volume 6261, 2010, pp 231-240

2008-2013 VoltDB, Inc.
Visited 01 June 2013.

[9] Beginning PHP and MySQL From Novice to Professional FOURTH EDITION
W. Jason Gilmore
2010 Apress
ISBN: 978-3114-1

[10] MySQL System Variables buffer pool size
http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size
Copyright © 1997, 2013, Oracle and/or its affiliates.
Visited 01 June 2013.

Copyright © 1997, 2013, Oracle and/or its affiliates.
Visited 01 June 2013.

[12] C-store: a column-oriented DBMS
Mike Stonebraker MIT CSAIL, Cambridge, MA
Daniel J. Abadi MIT CSAIL, Cambridge, MA
Adam Batkin Brandeis University, Waltham, MA
Xuedong Chen UMass Boston, Boston, MA
Mitch Cherniack Brandeis University, Waltham, MA
Miguel Ferreira MIT CSAIL, Cambridge, MA
Edmond Lau MIT CSAIL, Cambridge, MA
Amerson Lin MIT CSAIL, Cambridge, MA
Sam Madden MIT CSAIL, Cambridge, MA
Elizabeth O'Neil UMass Boston, Boston, MA
Pat O'Neil UMass Boston, Boston, MA
Alex Rasin Brown University, Providence, RI
2009

[13] The pathologies of big data
Adam Jacobs
1010data Inc. and Cornell University
2009
CACM Homepage archive
Volume 52 Issue 8, August 2009
Pages 36-44

[14] The entity-relationship model—toward a unified view of data
Peter Pin-Shan Chen
Massachusetts Institute of Technology, Cambridge
ACM Transactions on Database Systems (TODS) - Special issue: papers from the international conference on very large data bases: September 22–24, 1975, Framingham, MA
TODS Homepage archive
Volume 1 Issue 1, March 1976
Pages 9-36

[15] Relational Databases for Querying XML Documents: Limitations and Opportunities
Jayavel Shanmugasundaram
Kristin Tufte, Gang He, Chun Zhang, David DeWitt, Jeffrey Naughton

Visited 10 June 2013

Copyright © 1997, 2013, Oracle and/or its affiliates.
Visited 14 June 2013
Appendix

A 1.1 VoltDB SQL code for creating tables

```sql
CREATE TABLE CUSTOMER(
    Id INT PRIMARY KEY NOT NULL,
    Name VARCHAR(30) NOT NULL,
    Birth INT NOT NULL,
    Email VARCHAR(30) NOT NULL
);

CREATE TABLE ITEM(
    Id INT PRIMARY KEY NOT NULL,
    Name VARCHAR(50) NOT NULL,
    Text VARCHAR(200) NOT NULL,
    Price FLOAT NOT NULL
);

CREATE TABLE ORDERS(
    IdOrder INT PRIMARY KEY NOT NULL,
    IdCustomer INT NOT NULL,
    IdItem INT NOT NULL
);

CREATE PROCEDURE FROM CLASS Customer;
CREATE PROCEDURE FROM CLASS Item;
CREATE PROCEDURE FROM CLASS Orders;
CREATE PROCEDURE FROM CLASS CustomerSELECT;
CREATE PROCEDURE FROM CLASS ItemSELECT;
CREATE PROCEDURE FROM CLASS OrdersSELECT;
CREATE PROCEDURE FROM CLASS CustomerDELETE;
CREATE PROCEDURE FROM CLASS ItemDELETE;
CREATE PROCEDURE FROM CLASS OrdersDELETE;
CREATE PROCEDURE FROM CLASS ItemSUM;
CREATE PROCEDURE FROM CLASS ItemAVG;

PARTITION TABLE CUSTOMER ON COLUMN Id;
PARTITION PROCEDURE Customer ON TABLE CUSTOMER COLUMN Id;
PARTITION PROCEDURE CustomerSELECT ON TABLE CUSTOMER COLUMN Id;
PARTITION PROCEDURE CustomerDELETE ON TABLE CUSTOMER COLUMN Id;

PARTITION TABLE ITEM ON COLUMN Id;
PARTITION PROCEDURE Item ON TABLE ITEM COLUMN Id;
PARTITION PROCEDURE ItemSELECT ON TABLE ITEM COLUMN Id;
PARTITION PROCEDURE ItemDELETE ON TABLE ITEM COLUMN Id;
```
PARTITION TABLE ORDERS ON COLUMN IdOrder;
PARTITION PROCEDURE Orders ON TABLE ORDERS COLUMN IdOrder;
PARTITION PROCEDURE OrdersSELECT ON TABLE ORDERS COLUMN IdOrder;
PARTITION PROCEDURE OrdersDELETE ON TABLE ORDERS COLUMN IdOrder;

A 1.2 Client application

```java
import org.VoltDB.*;
import org.VoltDB.client.*;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Timestamp;
import java.util.Date;

public class Client {
  public static void main(String[] args) throws Exception {
    org.VoltDB.client.Client Clientvolt;
    Clientvolt = ClientFactory.createClient();
    Clientvolt.createConnection("localhost");
    int numberOfRows = 15000000;
    String customerEmailArr[] = new String[numberOfRows];

    String customerNameArr[] = new String[numberOfRows];
    int customerBirthArr[] = new int[numberOfRows];
    String itemNameArr[] = new String[numberOfRows];
    String itemTextArr[] = new String[numberOfRows];
    float itemPriceArr[] = new float[numberOfRows];
    int ordersIdCustomerArr[] = new int[numberOfRows];
    int ordersIdItemArr[] = new int[numberOfRows];
    int customerIdArr[] = new int[numberOfRows];
    String customerNameArrtmp[] = new String[numberOfRows];
    Timer clientTimer = new Timer();
    Timer deleteTimer = new Timer();
    Timer selectTimer = new Timer();
    Timer sumTimer = new Timer();
    Timer avgTimer = new Timer();
    Timer joinTimer = new Timer();
    String console = "";
    String idValue = "";
    String Table = "";
    long insertStart = 0;
    long insertEnd = 0;
    long deleteStart = 0;
    long deleteEnd = 0;
    int nrOfExecutes = 0;
    int nrOfStages = 0;
  }
}
```
long arrInsert[] = new long[15];
long arrCustomerSELECT[] = new long[15];
long arrItemSUM[] = new long[15];
long arrItemAVG[] = new long[15];
long arrDelete[] = new long[15];

while(true)
{
    if(nrOfExecutes >= 5)
    {
        console = inputReader();
        nrOfStages = 10;
    }
    if(nrOfStages == 6)
    {
        console = "getTime";
        nrOfStages = 0;
        nrOfExecutes++;
    } else
if(nrOfStages == 5)
    {
        console = "delete";
        nrOfStages = 6;
    } else
if(nrOfStages == 4)
    {
        console = "ItemAVG";
        nrOfStages = 5;
    } else
if(nrOfStages == 3)
    {
        //console = "ItemSUM";
        nrOfStages = 4;
    } else
if(nrOfStages == 2)
    {
        console = "CustomerSELECT";
        nrOfStages = 3;
    } else
if(nrOfStages == 1)
    {
        console = "insert";
        nrOfStages = 2;
    } else
if(nrOfStages == 0)
    {
        console = "initValues";
        nrOfStages = 1;
    }
    if(console.equals("insert"))
    {
        java.util.Date insDateStart = new java.util.Date();
        clientTimer.startInsert = insDateStart.getTime();
        for(int i = 1; i < numberOfRows; i++)
        {
            int iVal = i % 1000000;
            Clientvolt.callProcedure(new CustomerCallback(),
            "Customer", i,
            customerNameArr[iVal], customerBirthArr[iVal],
            customerEmailArr[iVal]);

            Clientvolt.callProcedure(new ItemCallback(), "Item", i,
Clientvolt.callProcedure(new OrderCallback(), "Orders", i, ordersIdCustomerArr[iVal], ordersIdItemArr[iVal]);
if (i % 1000000 == 0)
{
    System.out.println(iVal);
}
java.util.Date insDateEnd = new java.util.Date();
clientTimer.endInsert = insDateEnd.getTime();
System.out.println("Done");
}
if (console.equals("delete"))
{
    java.util.Date delDateStart = new java.util.Date();
deleteTimer.startInsert = delDateStart.getTime();
for (int i = 1; i < numberOfRows; i++)
{
    Clientvolt.callProcedure(new CustomerCallback(), "CustomerDELETE", i);
    Clientvolt.callProcedure(new ItemCallback(), "ItemDELETE", i);
    Clientvolt.callProcedure(new OrderCallback(), "OrdersDELETE", i);
}
java.util.Date delDateEnd = new java.util.Date();
deleteTimer.endInsert = delDateEnd.getTime();
System.out.println("Done");
}
if (console.equals("initValues"))
{
    for (int i = 0; i < numberOfRows/(numberOfRows/1000000); i++)
    {
        customerBirthArr[i] = SQL.generateBirthCustomer();
customerEmailArr[i] = SQL.generateEmailCustomer();
customerNameArr[i] = SQL.generateNameCustomer();
itemNameArr[i] = SQL.generateNameItem();
itemTextArr[i] = SQL.generateTextItem();
itemPriceArr[i] = SQL.generatePriceItem();
ordersIdCustomerArr[i] = SQL.generateIdCustomerOrders(numberOfRows);
ordersIdItemArr[i] = SQL.generateIdCustomerOrders(numberOfRows);
    }
System.out.println("Done");
}
if (console.equals("CustomerSELECT"))
{
    for (int i = 1; i < numberOfRows; i++)
    {
        customerIdArr[i] = SQL.generateIdCustomerOrders(numberOfRows);
    }
java.util.Date selectDateStart = new java.util.Date();
selectTimer.startInsert = selectDateStart.getTime();
for (int i = 1; i < numberOfRows; i++)
{
    
}
// ClientVolt.callProcedure(new CustomerSELECTCallback(),
"CustomerSELECT", customerIdArr[i]);

// Select on value that doesn’t exist
ClientVolt.callProcedure(new CustomerSELECTCallback(),
"CustomerSELECT", 20000000);

java.util.Date selectDateEnd = new java.util.Date();
selectTimer.endInsert = selectDateEnd.getTime();
System.out.println("Done");

if(console.equals("ItemSUM"))
{
    java.util.Date sumDateStart = new java.util.Date();
    sumTimer.startInsert = sumDateStart.getTime();
    final ClientResponse itemSumResponse =
        ClientVolt.callProcedure("ItemSUM");
    if(itemSumResponse.getStatus() != ClientResponse.SUCCESS)
    {
        System.err.println(itemSumResponse.getStatusString());
    }
    final VoltTable results[] = itemSumResponse.getResults();
    if(results.length == 0 || results[0].getRowCount() != 1) {
        System.out.println("No result from ItemSUM!");
    }
    else {
        VoltTable resultTable = results[0];
        VoltTableRow row = resultTable.fetchRow(0);
        System.out.println(row.getDouble("C1"));
    } 
    java.util.Date sumDateEnd = new java.util.Date();
    sumTimer.endInsert = sumDateEnd.getTime();
    System.out.println("Done");
}

if(console.equals("ItemAVG"))
{
    java.util.Date avgDateStart = new java.util.Date();
    avgTimer.startInsert = avgDateStart.getTime();
    final ClientResponse itemAvgResponse =
        ClientVolt.callProcedure("ItemAVG");
    if(itemAvgResponse.getStatus() != ClientResponse.SUCCESS)
    {
        System.err.println(itemAvgResponse.getStatusString());
    }
    final VoltTable results[] = itemAvgResponse.getResults();
    if(results.length == 0 || results[0].getRowCount() != 1) {
        System.out.println("No result from ItemAVG!");
    }
    else {
        VoltTable resultTable = results[0];
        VoltTableRow row = resultTable.fetchRow(0);
    }
```java
System.out.println(row.getDouble("C1"));
}
java.util.Date avgDateEnd = new java.util.Date();
avgTimer.endInsert = avgDateEnd.getTime();
System.out.println("Done");

} else if(console.equals("getTime")){
    long calcInsert = clientTimer.endInsert -
    clientTimer.startInsert;
    long calcDel = deleteTimer.endInsert -
    deleteTimer.startInsert;
    long calcSelectCustomer = selectTimer.endInsert -
    selectTimer.startInsert;
    long calcItemSum = sumTimer.endInsert -
    sumTimer.startInsert;
    long calcItemAvg = avgTimer.endInsert -
    avgTimer.startInsert;
    System.out.println("Insert: "+calcInsert);
    System.out.println("Delete: "+calcDel);
    System.out.println("Select customer: "+
calcSelectCustomer);
    System.out.println("Sum item: "+calcItemSum);
    System.out.println("Avg item: "+calcItemAvg);
    arrInsert[nrOfExecutes] = calcInsert;
    arrCustomerSELECT[nrOfExecutes] =
calcSelectCustomer;
    arrItemSUM[nrOfExecutes] = calcItemSum;
    arrItemAVG[nrOfExecutes] = calcItemAvg;
    arrDelete[nrOfExecutes] = calcDel;
}
if(console.equals("output")){
    for(int i = 0; i < nrOfExecutes + 1; i++)
    {
        System.out.println("Insert: "+
        arrInsert[i]);
        System.out.println("Select customer: "+
        arrCustomerSELECT[i]);
        System.out.println("Sum item: "+
        arrItemSUM[i]);
        System.out.println("Avg item: "+
        arrItemAVG[i]);
        System.out.println("Delete: "+
        arrDelete[i]);
    }
}
}

public static String inputReader()
{
    String s = ""
    try{
        BufferedReader bufferRead = new BufferedReader(new
        InputStreamReader(System.in));
        s = bufferRead.readLine();
    }
    catch(Exception e)
    {
```
import orgVoltDB.*;

public class CustomerSELECT extends VoltProcedure {

    public final SQLStmt sql = new SQLStmt(
            "SELECT Name, Id, Birth, Email FROM CUSTOMER " +
            "WHERE Id = ?"
    );

    public VoltTable[] run (int id) throws VoltAbortException{
        voltQueueSQL(sql, id);
        return voltExecuteSQL();
    }
}

A 1.3 Customer Select

A 1.4 Class for generating test data
import java.util.Random;
import orgVoltDB.*;
import orgVoltDB.client.*;
/**
* @author rehnen
*/
public class SQL {
    /**
     * @param args the command line arguments
     */
    public static String generateRandomStringAtoZ(int min, int max) {
        char[] chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
        String s = "";
        Random r = new Random();
        int loopamount = r.nextInt(max - min) + min;
        for (int i = 0; i < loopamount; i++) {
            s += chars[r.nextInt(chars.length)];
        }
        return s;
    }

    public static String generateRandomStringAtoZ0to9(int min, int max) {
        char[] chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789".toCharArray();
        String s = "";
        Random r = new Random();
        int loopamount = r.nextInt(max - min) + min;
        for (int i = 0; i < loopamount; i++) {
            s += chars[r.nextInt(chars.length)];
        }
        return s;
    }

    public static String generateNameCustomer() {
        return generateRandomStringAtoZ(4, 30);
    }

    public static int generateBirthCustomer() {
        return new Random().nextInt(100) + 1900;
    }

    public static String generateEmailCustomer() {
        return generateRandomStringAtoZ0to9(10, 30);
    }

    public static String generateNameItem() {
        return generateRandomStringAtoZ0to9(4, 50);
    }

    public static String generateTextItem() {
        return generateRandomStringAtoZ0to9(10, 200);
    }

    public static float generatePriceItem() {
        return new Random().nextFloat() * 10000;
    }

    public static int generateIdCustomerOrders(int numbers) {

A 2.1 SQL code for MySQL Tables

/*
   return new Random().nextInt(numbers);
*/

DELIMITER ;

-- --------------------------------------------------------
--
-- Table structure for table `CUSTOMER`
--

DROP TABLE IF EXISTS NewTest.CUSTOMER;

CREATE TABLE NewTest.CUSTOMER ( 
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Name` varchar(30) COLLATE utf8_swedish_ci NOT NULL,
  `Birth` int(11) NOT NULL,
  `Email` varchar(30) COLLATE utf8_swedish_ci NOT NULL,
  PRIMARY KEY (`Id`),
  KEY `Name` (`Name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_swedish_ci;

-- --------------------------------------------------------
--
-- Table structure for table `ITEM`
--

DROP TABLE IF EXISTS NewTest.ITEM;

CREATE TABLE NewTest.ITEM ( 
  `Id` int(11) NOT NULL,
  `Name` varchar(50) COLLATE utf8_swedish_ci NOT NULL,
  `Description` varchar(200) COLLATE utf8_swedish_ci NOT NULL,
  `Price` float NOT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_swedish_ci;

-- --------------------------------------------------------
--
-- Table structure for table `ORDERS`
--

DROP TABLE IF EXISTS NewTest.ORDERS;

CREATE TABLE NewTest.ORDERS ( 
  `IdCustomer` int(11) NOT NULL,
  `IdItem` int(11) NOT NULL,
  `Id` int(11) NOT NULL,
  PRIMARY KEY (`Id`),
  FOREIGN KEY (`IdCustomer`) REFERENCES CUSTOMER(`Id`),
  FOREIGN KEY (`IdItem`) REFERENCES ITEM(`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_swedish_ci;

-- --------------------------------------------------------
A 2.2 Class for MySQL client

```java
package MySQL.connect;

import java.sql.CallableStatement;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Random;
import java.util.Scanner;
import java.util.Timer;

/**
 * @author rehnen
 */
public class MySQLConnect {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        System.out.println("Enter the number of millions to be used: ");
        Scanner scanner = new Scanner(System.in);
        int nrOfMillions = scanner.nextInt();
        int numberOfRows = 1000000;
        String customerNameArr[] = new String[numberOfRows];

        Timer clientTimer = new Timer();
        Timer deleteTimer = new Timer();
        Timer selectTimer = new Timer();
        Timer sumTimer = new Timer();
        Timer avgTimer = new Timer();
        Timer joinTimer = new Timer();
        String console = "";
        String idValue = "";
        String Table = "";
        long insertStart = 0;
    }
```
Connection connection = null;
//<editor-fold desc="generate test data">
// Initialize the values
for (int i = 0; i < (numberOfRows); i++) {
    customerNameArr[i] = SQL.generateNameCustomer();
}
System.out.println((new Random().nextInt((numberOfRows * nrOfMillions) - 1)));

try {
    connection = DriverManager.getConnection("jdbc:mysql://194.47.150.207:3306/NewTest", "root", "twilight");

//-------- INSERT CUSTOMERS

int hi = 0;
for (int i = 0; i < nrOfMillions; i++) {
    int h = 0;

    System.out.println("inserting customer");
    for (int o = 0; o < 100; o++) {
        int f = 0;
        String insertStr = "INSERT INTO CUSTOMER (Id, Name, Birth, Email)
VALUES";
        StringBuilder sb = new StringBuilder();
        sb.append(insertStr);
        //System.out.println("h = " + h);
        for (int i = 0; i < (numberOfRows / 100) - 1; i++) {
            sb.append("(" + (hi+1) + "," + customerNameArr[h] + "," + SQL.generateBirthCustomer() + "," + SQL.generateEmailCustomer() + ")");
            h++;
            hi++; f++;
        }
        sb.append("(" + (hi+1) + "," + customerNameArr[h] + "," + SQL.generateBirthCustomer() + "," + SQL.generateEmailCustomer() + ")");
        h++;
        hi++; f++;

    Statement stmt = connection.createStatement();
long start = System.nanoTime();
stmt.executeUpdate(sb.toString());
long end = System.nanoTime();
System.out.println((end - start));
}
}
int k = 0;
System.out.println("insert item");
for (int l = 0; l < nrOfMillions; l++) {
    for (int j = 0; j < 100; j++) {
        String insertStr2 = "INSERT INTO ITEM (id, name, Description, price)
VALUES";
        StringBuilder sb2 = new StringBuilder();
        sb2.append(insertStr2);
        for (int i = 0; i < (numberOfRows / 100) - 1; i++) {
            sb2.append("(" + (k + 1) + "," + SQL.generateNameItem() + "," + SQL.generateTextItem() + "," + SQL.generatePriceItem() + ")"),
            k++;
        }
        sb2.append("(" + "," + (k + 1) + "," + SQL.generateNameItem() + "," + SQL.generateNameItem() + "," + SQL.generatePriceItem() + ")"),
        k++;
        Statement stmt2 = connection.createStatement();
        long start2 = System.nanoTime();
        stmt2.executeUpdate(sb2.toString());
        long end2 = System.nanoTime();
        System.out.println((end2 - start2));
    }
}
System.out.println("insert orders");
k = 0;
for (int l = 0; l < nrOfMillions; l++) {
    String insertStr3 = "INSERT INTO ORDERS (idCustomer, idItem, id) VALUES";
    StringBuilder sb3 = new StringBuilder();
    sb3.append(insertStr3);
    for (int i = 0; i < numberOfRows - 1; i++) {
        sb3.append("(" + "," + (new Random().nextInt((numberOfRows * nrOfMillions) - 1) + 1) + "," + (new Random().nextInt((numberOfRows * nrOfMillions) - 1) + 1) + ",", k + 1 + ")",
        k++;
    }
    sb3.append("(" + "," + (new Random().nextInt((numberOfRows * nrOfMillions) - 1) + 1) + "," + (new Random().nextInt((numberOfRows * nrOfMillions) - 1) + 1) + "," + k + 1 + ")",
        k++;
    long start3 = System.nanoTime();
    Statement stmt3 = connection.createStatement();
stmt3.executeUpdate(sb3.toString());
long end3 = System.nanoTime();
System.out.println((end3 - start3));
}
System.out.println("LOL");
long totaltime = 0;
for (int j = 0; j < 10; j++) {
totaltime = 0;
for (int i = 1; i < 100; i++) {
    String sql = "Select * from CUSTOMER where name Like '" +
customerNameArr[(new Random().nextInt((numberOfRows) - 1) + 1)] + "+";
    Statement statement = connection.createStatement();
    long startSelect = System.nanoTime();
    statement.executeQuery(sql);
    long endSelect = System.nanoTime();
    totaltime += endSelect - startSelect;
}
System.out.println("Total time searching for 100 elements that exists: " + totaltime);
}

for (int l = 0; l < 10; l++) {
totaltime = 0;
for (int i = 1; i < 100; i++) {
    String sql = "Select * from CUSTOMER where Id = '" +
(new Random().nextInt((numberOfRows * nrOfMillions) - 1) + 1) + "+";
    Statement statement = connection.createStatement();
    long startSelect = System.nanoTime();
    statement.executeQuery(sql);
    long endSelect = System.nanoTime();
    totaltime += endSelect - startSelect;
}
System.out.println("Total time searching for 100 elements by id that do exists: " + totaltime);
}
String sql = "Select * from CUSTOMER where Id Like "' + (new Random().nextInt((numberOfRows * nrOfMillions) - 1) + maxval + 100) + ''';
Statement statement = connection.createStatement();
long startSelect = System.nanoTime();
statement.executeQuery(sql);
long endSelect = System.nanoTime();
totaltime += endSelect - startSelect;
}
System.out.println("Total time searching for 100 elements by id that do not exists: " + totaltime);
}
for (int i = 0; i < 10; i++) {

totaltime = 0;
Statement statement = connection.createStatement();
String sql = "SELECT avg( price )FROM ITEM";
long startSelect = System.nanoTime();
statement.executeQuery(sql);
long endSelect = System.nanoTime();
totaltime += endSelect - startSelect;
System.out.println("Total time AVG(): " + totaltime);
}
for (int i = 0; i < 10; i++) {

totaltime = 0;
Statement statement = connection.createStatement();
String sql = "SELECT sum( price )FROM ITEM";
long startSelect = System.nanoTime();
statement.executeQuery(sql);
long endSelect = System.nanoTime();
totaltime += endSelect - startSelect;
System.out.println("Total time sum(): " + totaltime);
}
int o = 1;
for (int i = 0; i < 10; i++) {

totaltime = 0;
for (int l = 0; l < 100; l++) {
    Statement statement = connection.createStatement();
    long startSelect = System.nanoTime();
    statement.executeUpdate("DELETE FROM CUSTOMER WHERE id = " + o);
    long endSelect = System.nanoTime();
    totaltime += endSelect - startSelect;
    o++;
}
        }
        System.out.println("Total time deleting all elements from customer by id: " +
        totaltime);
    }

    } catch (SQLException e) {
        System.out.println(e.getMessage() + "-.-.-.-.-.-.-.-");
        System.out.println(e.getSQLState());
        System.exit(1);
    }
}
## A 3 Tables

### Customer

<table>
<thead>
<tr>
<th>Fieldname</th>
<th>Datatype</th>
<th>Attribute</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
<td>primarykeynonnull</td>
<td>1 to maximumvalue</td>
</tr>
<tr>
<td>name</td>
<td>varchar(30)</td>
<td>notnullindex</td>
<td>4 to 30</td>
</tr>
<tr>
<td>birth</td>
<td>int</td>
<td>notnull</td>
<td>1900 to 1999</td>
</tr>
<tr>
<td>email</td>
<td>varchar(30)</td>
<td>notnull</td>
<td>10 to 30</td>
</tr>
</tbody>
</table>

### Item

<table>
<thead>
<tr>
<th>Fieldname</th>
<th>Datatype</th>
<th>Attribute</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
<td>primarykeynonnull</td>
<td>1 to maximumvalue</td>
</tr>
<tr>
<td>name</td>
<td>varchar(50)</td>
<td>notnullindex</td>
<td>4 to 50</td>
</tr>
<tr>
<td>price</td>
<td>float</td>
<td>notnull</td>
<td>0 to 10000</td>
</tr>
<tr>
<td>text</td>
<td>varchar(200)</td>
<td>notnull</td>
<td>10 to 30</td>
</tr>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>Fieldname</th>
<th>Datatype</th>
<th>Attribute</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
<td>primarykeynonnull</td>
<td>1 to maximumvalue</td>
</tr>
<tr>
<td>idItem</td>
<td>varchar(30)</td>
<td>notnullindex</td>
<td>1 to the maximum of Item.id</td>
</tr>
<tr>
<td>idCustomer</td>
<td>varchar(30)</td>
<td>notnull</td>
<td>1 to the maximum of Customer.id</td>
</tr>
</tbody>
</table>