ROMA TRE
UNIVERSITÀ DEGLI STUDI

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea

In-Memory Database: Competitive Landscape and Performance Analysis

Laureando

Valerio Barbagallo

Relatore

Prof. **Paolo Merialdo**

Universit Roma Tre

Tutor Aziendale

Dr. **Michele Aiello**

Applications Director & Co. Founder

ERIS4 s.r.l.

Anno Accademico 2007-2008

Happiness is not having what you want,

but wanting what you have.

Acknowledgements

Thank you for your attention!

Abstract

Naturally this is a joke. The abstract must be written after all chapters are written. Here i'm going to tell you what i'm talking about. The thesis is divided into the following chapters:

1. In-Memory Database
    - A brief introduction
        - Definition
        - Usage
    - IMDBs overview
        -
        -
2. Database Performance Analysis
    -
    -
3. High availability with in-memory database?

Contents

List of Figures

List of Tables

Part I

Competitive Landscape

Chapter 1

A Brief Introduction

"While familiar on desktops and servers, databases are a recent arrival to embedded systems. Like any organism dropped into a new environment, databases must evolve. A new type of DBMS, the in-memory database system (IMDS), represents the latest step in DBMSes' adaptation to embedded systems" [Graves 02].

In this chapter we are going to make a brief introduction to the in-memory databases, explaining what they are, their use, their strength and weakness.

1.1 Definition

An in-memory database (IMDB), also called main memory database (MMDB), in-memory database system (IMDS) or real-time database (RTDB), is a database management system that relies on main memory for data storage. While the bandwidth of hard disks is just 1 order of magnitude slower than the main memory's bandwidth, the disk access time is about 3 order of

magnitude slower than the RAM access time, and thus in-memory databases can be much more faster than traditional database management systems (DBMS).

1.2 History

Initially embedded systems developers produced their own data management solutions. But with the market competition requiring smarter devices, applications with expanding feature set will have to manage increasingly complex data structures. As a consequence, these data management solutions were outgrowing, and became difficult to maintain and extend.

Therefore embedded systems developers turned to commercial databases. But the first embedded databases were not the ideal solution. They were traditional DBMS with complex caching logic to increase performance, and with a lot of unnecessary features for the device that make use of embedded databases. Furthermore these features cause the application to exceed available memory and CPU resources.

In-memory databases have emerged specifically to meet the performance needs and resource availability in embedded systems.

1.3 Application Scenario

Often in-memory databases run as embedded database, but it's not their only use. Thanks to their high performance, these databases are particularly useful for all that kind of applications that need fast access to the data. Some

examples:

- real time applications which don't need to be persisted either because it doesn't change, or the data can be reconstructed: imagine a routing table of a router with millions of record and data access in less than few milliseconds; the routing table can be rebuilt [Fowler 05].

- real time applications with durability needs which capture, analyze and respond intelligently to important events, requiring high performance in terms of throughput and mainly latency (traditional DBMS can be clustered to increase the througput, but with no great benefits in terms of latency). Infact almost all IMDBs can be persistent on disk while still keeping higher performance compared to traditional DBMSs.

- in-memory databases are also very useful for developers of traditional database systems for testing purpose: in a enterprise application running a test suite can take long; switching to an IMDB can reduce the whole build time of the application.

1.4 Comparison against Traditional DBMS

In-memory databases eliminate disk I/O and exist only in RAM, but they are not simply a traditional database loaded into main memory. Linux systems already have the capability to create a RAM disk, a file system in main memory. But a traditional database deployed in a such virtual hard disk doesn't provide the same benefits of a pure IMDB. In-memory databases are

less complex than a traditional DBMS fully deployed in RAM, and lead to a minor usage of CPU and RAM.

Comparing IMDBs with traditional databases we can find at least 3 key differences:

- Caching.
- Data-transfer overhead.
- Transaction processing.

1.4.1 Caching

All traditional DBMS software incorporates caching mechanisms to keep the most used records in main memory to reduce the performance issue introduced by the disk latency. The removal of caching brings to the elimination of the following tasks:

- cache synchronization, used to keep the portion of the database loaded in main memory consistent with the physical database image.
- cache lookup, a task that handle the cached page, determining if the data requested is in cache

Therefore, removing the cache, IMDBs eliminate a great source of complexity and performance overhead, reducing the work for the CPU and, comparing to a traditional DBMS fully deployed in main memory, also the RAM.

1.4.2 Data-Transfer Overhead

Traditional DBMS adds a remarkable data transfer overhead due not only to the DB cache, but to the file system and his cache too. In contrast an IMDBs, which eliminate these steps, have little or no data transfer.

There is also no need for the application to make a copy of the data in local memory, because IMDBs give to the application a pointer to the data that reside in the database. In this way the data is accessed through the database API that protect the data itself.

1.4.3 Transaction Processing

In an hard-disk database the recovery process, from a disaster or a transaction abort, is based on a log file, that is update every time a transaction is executed. To provide transactional integrity, IMDBs mantain a before image of the objects updated and in case of a transaction abort, the before image are restored in a very efficient way. Therefore another complex, memory-intensive task is eliminated from the IMDB, unless the need to add durability to the system, because there is no reason to keep transaction log files.

1.5 ACID Support: Adding Durability

When we choose a database we expect from it to provide ACID support: atomicity, consistency, isolation and durability. While the first three features are usually supported by in-memory databases, pure IMDBs, in their simplest

form, don't provide durability: main memory is a volatile memory and loses all stored data during a reset.

With their high performance, IMDBs are a good solution for time-critical applications, but when the durability need arises they may not seem a proper solution anymore. To achieve durability [Gorine 04], in-memory database systems can use several solutions:

- On-Line Backup.
- Transaction logging.
- High availability implementations.
- Non volatile RAM (NVRAM).

1.5.1 On-Line Backup

On-line backup is a backup performed while the database is on-line and available for read/write. This is the simplest solution, but offer a minimum degree of durability.

1.5.2 Transaction Logging

A transaction log is a history of actions executed by a database management system. To guarantee ACID properties over crashes or hardware failures the log must be written in a non-volatile media, usually an hard disk. If the system fails and is restarted, the database image can be restored from this log file.

The recovery process acts similarly to traditional DBMS with a roll-back or a roll-forward recovery. Checkpoint (or snapshot) can be used to speed up this process. However this technique implies the usage of persistence memory such as an hard disk, that is a bottleneck, especially during the resume of the database.

Although this aspect, IMDBs are still faster than traditional DBMS:

1. transaction logging requires exactly one write to the file system, while disk-based databases not only need to write on the log, but also the data and the indexes (and even more writes with larger transactions).

2. transaction logging may be usually set to different level of transaction durability. A trade-off between performance and durability is allowed by setting the log mechanism to be synchronous or asynchronous.

1.5.3 High Availability Implementation

High availability is a system design protocol and associated implementation: in this case it is a database replication, with automatic fail over to an identical standby database. A replicated database consists of failure-independent nodes, making sure data is not lost even in case of a node failure. This is an effective way to achieve database transaction durability.

In a simile way to transaction logging, a trade-off between performance and durability is achieved by setting the replication as eager (synchronous) or lazy (asynchronous).

1.5.4 NVRAM

To achieve durability a IMDB can support non volatile ram (NVRAM): usually a static RAM backed up with battery power, or some sort of EEPROM. In this way the DBMS can recover the data also after a reboot.

This is a very attractive durability option for IMDBs. NVRAM in contrast to transaction logging and database replication does not involve disk I/O latency and neither the communication overhead.

Despite this, vendors rarely provide support for this technology. One of the major problems to this approach is the limited write-cycles of this kind of memory, such as a flash memory. On the other hand there is some new memory device that has been proposed to address this problem, but the cost of such devices is rather expensive, in particular considering the huge amount of memory needed by IMDBs.

Chapter 2

In-Memory Database Overview

There is a variety of in-memory database systems which can be used to maintain a database in main memory, both commercial and open source. Although all of them share the capability to maintain the database in main memory, they offer different sets of feature.

In this chapter we will make a competitive landscape of the most famous in-memory databases.

2.1 The Analysis' Structure

Every in-memory database will be analyzed investigating their advantages and disadvantages, the stability and reliability of the project and how much development is going on. Eventually we will go to the development stage and, in some case, we will also have a look "under the hood" to see how the DB works.

2.1.1 Common Advantages

Not all IMDBs have the same advantages, but generally they share some feature:

Lightweigth is one of the common advantages IMDBs share. This kind of databases is really simple: they don't implement any of the complex mechanism used by traditional database to speed up the disk I/O. Thus, most of the time, an in-memory database consist of a simple jar, whose size is less than 1 MB.

Robustness is a direct consequence of the previous point: (citing Henry Ford) it is impossible to break something that doesn't exist.

High Performance is another common feature for every IMDB, because they all store the whole database in main memory, avoiding disk access.

2.1.2 Common Disadvantages

On the other side, all IMDBs require high quantity of main memory. It is larger as the database image increases, but this doesn't mean that every IMDB uses the same RAM quantity for the same database image.

A common question is about what may happen when the RAM is not enough. Altought this question is really interesting, generally an IMDB makes the assumption there is always enough main memory to hold the database image.

2.1.3 Common references

The source of informations described in the following sections comes from the web, mainly from sourceforge or ohloh.net. Both these web site offer a variety of news such as the size of developer team, the frequency of commits, the date of the last realese, etc.

Another source is, naturally, the IMDBs' official web site. Although this is a huge source, it cannont be considered impartial. Therefore the following analysis may not be consistent with facts.

2.2 Prevayler

Prevayler is an object persistence library for Java, written in Java. It keeps data in main memory and any change is written to a journal file for system recovery. This is an implementation of an architectural style that is called System Prevalence by the Prevayler developer team.

Therefore Prevayler, being an object prevalence layer, provides transparent persistence for Plain Old Java Objects. In the prevalent model, the object data is kept in memory in native, language-specific object format, and therefore avoid any marshalling to an RDBMS or other data storage system. To avoid losing data and to provide durability a snapshot of data is regularly saved to disk and all changes are serialized to a log of transactions which is also stored on disk [Wuestefeld 01].

All is based on the Prevalent Hypothesis: that there is enough RAM to hold all business objects in your system.

2.2.1 Advantages

Prevayler is a lightweight java library, just 350 KB, that is extremely simple to use. There is no separate database server to run. With Prevayler you can program with real objects, there is no use of SQL, there is no impedance mismatch such as when programming with RDBMS. Moreover Prevayler doesn't require the domain objects to implement or extend any class in order to be persistent (except `java.io.Serializable`, but this is only a marker interface).

It is very fast. Simply keeping objects in memory in their language-specific format is both orders of magnitude faster and more programmer-friendly than the multiple conversions that are needed when the objects are stored and retrieved from an RDBMS. The only disk access is the streaming of the transactions to the journal file that should be about one thousand[1] transactions per second on an average desktop computer.

The thread safety is guaranteed. Actually, in the default Prevayler implementation all writes to the system are synchronized. One write at a time. So there's no threading issues at all. Therefore there is no more multithreading issue such as locking, atomicity, consistency and isolation.

Finally Prevayler supports the execution only in RAM, like a pure in-memory database should work. But it can also provide persistence through a journal file, as we described above. Moreover Prevayler supports snapshots of the database's image, which serves to speed up the database's boot, and server replication, enabling query load-balancing and system fault-tolerance [Prevayler 08]. This last feature is really promising, because in-

[1]This is what Prevayler team says on their official web site on Mar 26, 2008.

memory databases suffer the start up process, allowing it to be used in an enterprise application. Althought this feature is not ready yet: see paragraph 2.2.4 for further details at page 19.

2.2.2 Disadvantages

On the other hand of Prevayler's simplicity, there is some restriction due to the fact that only changes are written in the journal file: transaction must be completely deterministic and repeteable in order to recover the state of the object (for instance it's not possible to use `System.getCurrentMillis()`).

In addition, when Prevayler is embedded in your application, the only client of your application's data is the application itself [Hobbs 03]. While deploying Prevayler as a server, the access to the data is still limited to whom who speaks the host language (you can't use another programming language unless you make your own serialization mechanism) and, at the same time, knows the model objects. For all these reasons there are no administration and migration tools.

Another problem is related to the Prevalent Hypothesis. If, for any reason, the RAM is not enough and an object model is swapped out of main memory, Prevayler will become very slow, more than traditional RDBMS's [Miller 03]. In fact Prevayler doesn't use any mechanism to optimize the disk I/O, such as traditional DBMS's mechanisms (eg: indexing, caching etc.) Anyway this issue belongs to all pure in-memory databases.

2.2.3 Project Info

Klaus Wuestefeld is the founder and main developer of Prevayler, an open source project. This project started in 2001, from what sourceforge reports, and had a great development until 2005. The community is still active, but the last update is dated at 25/05/2007 when version 2.3 (the latest) was released. The development team is composed of 8 developers, including Klaus Wuestefeld.. The current development status is declared to be production/stable.

2.2.4 Usage

In this example, and in all the followings, we are going to use a simple Plain Old Java Object as business object that need to be persisted: `Number`. It has one single private field, which is the value of the number itself, and the relative getter and setter methods.

```
public class Number{
  private int value;
  public Number(){}
  public Number(int value) {
    this.value = value;
  }
  public int getValue() {
    return value;
  }
  public void setValue(int number) {
    this.value = number;
  }
```

```
}
```

Prevayler usage is quite different from a traditional RDBMS with JDBC driver. Prevayler is accessed through a native driver, and it acts similarly to a framework: your business objects must implement the interface `java.io.Serializable` in order to be persisted (which is only a marker interface); and all modifications to these objects must be encapsulated in transaction objects. Therefore our business object will appear as follow:

```
public class Number implements Serializable { ...
```

Basic: main memory and transaction logging

This example is about the default Prevayler's usage: how to create an in-memory database without losing durability. This property is achieved with a transaction log file, whose usage is totally transparent. To initialize Prevayler Database you need to create a Prevayler, providing it with the directory where you want to save your database, and the object which is going to be saved in this Prevayler database. This step can be compared to a `CREATE TABLE` in SQL, but only one object will be saved in your Prevayler database. Therefore, from a RDBMS perspective where the table is a class and each row is one instance, you may want to initialize Prevayler with a `List` or a `Map` of `Number`. Here is an example:

```
public class Main{
  private static final String DIRECTORY_DB = "numberDB";
  public static void main(String[] args) throws Exception {
    Prevayler prevayler = PrevaylerFactory.createPrevayler(
      new HashMap<Integer,Number>(),DIRECTORY_DB);
```

```
    new Main().fillPrevaylerDb(prevayler);
    new Main().readPrevaylerDb(prevayler);
  }
  ...
```

It's important to understand that the state of the object you use to create this database will not be saved in the database itself. To insert any `Number` in the database a transaction must be executed:

```
private void fillPrevaylerDb(Prevayler prevayler) throws ↘
    InterruptedException {
  for (int i = 0; i<100; i++){
    prevayler.execute(new InsertNumberTransaction(new Number↘
        (i)));
    System.out.println("The␣value␣of␣the␣number␣inserted␣is␣↘
        =␣"+i);
  }
}
```

The parameter of the method `execute` must be a class that extends `org↘.prevayler.Transaction`. Every insert, update or delete (in other words: any write operation) requires to be executed inside a transaction. In this particular case this is the the code:

```
public class InsertNumberTransaction implements Transaction{
  private Number number;
  public InsertNumberTransaction(Number number) {
    this.number = number;
  }
  public void executeOn(Object prevalentSystem, Date ignored↘
      ) {
```

```
    Map<Integer,Number> map = (Map<Integer, Number>) ↘
        prevalentSystem;
    map.put(number.getValue(), number);
  }
}
```

It's important to note that when you stop the database, and then you restart it, Prevayler will execute all the transactions exactly the same number of times they were executed before shutting down the process [2]. While a snapshot should avoid this behavior.

Finally, reading from Prevayler database is really simple and doesn't require any transaction:

```
private void readPrevaylerDb(Prevayler prevayler) {
  Map<Integer,Number> map= (Map<Integer,Number>) prevayler.↘
     prevalentSystem();
  Set<Integer> keys = map.keySet();
  for (Integer key : keys) {
    Number number = map.get(key);
    System.out.println("Reading␣the␣number␣" + number.↘
        getValue());
  }
}
```

Only RAM

This example show how to make Pervayler run only in main memory, without the writes to the journal file, in the case you want a database even faster and you don't care for durability or you don't have write permission. There is

[2] This is the reason why the transactions must be deterministic

only one line of code which need to be modified to be able to run Prevayler in such a way:

```
Prevayler prevayler = PrevaylerFactory.↘
    createTransientPrevayler(new HashMap<Integer,Number>());
```

Instead of creating a persistent prevayler, you just need to `createTransientPrevayler`↘. You can also decide to create the transient prevayler from a snapshot, and then work only in main memory: really useful for the execution of your test case. But you can't take a snapshot while using transient prevayler, therefore you need to disable the snapshot in your code when switching from the default to the transient prevayler, otherwise a `IOException` will raise.

With this method you have no durability, but it is even faster than the first example, about 10 times faster. And moreover it is more scalable, because there is no more bottleneck caused by the hard disk.

Server Replication

Prevayler can support also a server replication modality. Also in this case, only a small change to the database initialization is needed. Quite obviously you need two different kind of initialization: server side and clients side.

As regards the server, you need to specify the port number and, then, simply create the object `Prevayler`:

```
public class MainServer {
  private static final String DB_DIRECTORY_PATH = "↘
     numberReplicaDB";
  private static final int PORT_NUMBER = 37127;
  public static void main(String[] args) throws Exception {
    PrevaylerFactory factory = new PrevaylerFactory();
```

```
    factory.configurePrevalentSystem(new HashMap<Integer,↘
        Number>());
    factory.configurePrevalenceDirectory(DB_DIRECTORY_PATH);
    factory.configureReplicationServer(PORT_NUMBER);
    Prevayler prevayler = factory.create();
    ...
    // execute your transactions
    ...
    // The server will continue to listen/run for incoming ↘
        connections
    ...
  }
}
```

As for the clients, you have to tell Prevayler not only the port number, but the ip address too. Only one line of code changes from the server:

```
public class MainReplicant {
  private static final String DB_DIRECTORY_NAME = "numberDB"↘
      ;
  private static final int PORT_NUMBER = 37127;
  private static final String SERVER_IP = "10.0.2.2";

  public static void main(String[] args) throws Exception {
    PrevaylerFactory factory = new PrevaylerFactory();
    factory.configurePrevalentSystem(new HashMap<Integer,↘
        Number>());
    factory.configurePrevalenceDirectory(DB_DIRECTORY_NAME);
    factory.configureReplicationClient(SERVER_IP, ↘
        PORT_NUMBER);
    Prevayler prevayler = factory.create();
```

```
    ...
    //execute your transactions
  }
}
```

With this setup, you can stop, restart and add clients without losing your data, and keeping it synchronized with the server and the other clients, apparently without any concurrent exception. But when you try to kill the server, or for any other reason the server stops, while any client is still active this is the message you get:

```
The replication logic is still under development.
java.io.EOFException
```

Reading the javadoc comes out that this feature is *reserved for future implementation*. Contacting Prevayler's author, Klaus Wuestefeld, by email, he said this feature will be probably done within the 2009, because he needs it for other projects.

2.3 HSQLDB

HSQLDB is a relational database written in Java. It is based on Thomas Mueller's Hypersonic SQL project and therefore its name [3].

2.3.1 Advantages

advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages

[3] HSQLDB stands for Hypersonic SQL DataBase.

advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages advantages

2.3.2 Disadvantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.3.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.4 Db4o

intorDB intorDB

2.4.1 Adavantages

advantages advantages

2.4.2 Disadavantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.4.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.5 Hxsql

intorDB intorDB

2.5.1 Advantages

advantages advantages

2.5.2 Disadvantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.5.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.6 H2

intorDB intorDB

2.6.1 Adavantages

advantages advantages

2.6.2 Disadavantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.6.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.7 Derby

intorDB intorDB

2.7.1 Adavantages

advantages advantages

2.7.2 Disadavantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.7.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.8 SQLite

intorDB intorDB

2.8.1 Adavantages

advantages advantages

2.8.2 Disadavantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.8.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.9 Firebird

intorDB intorDB

2.9.1 Adavantages

advantages advantages

2.9.2 Disadavantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.9.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.10 MySql

intorDB intorDB

2.10.1 Adavantages

advantages advantages

2.10.2 Disadavantages

disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages disadvantages

2.10.3 Project Info

info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info info

2.11 ExtremeDB

intorDB intorDB

2.12 Polyhedra

intorDB intorDB

2.13 TimesTen

intorDB intorDB

2.14 Csql

intorDB intorDB

2.15 SolidDB

intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB

intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB intorDB

2.16 MonetDB

intorDB intorDB

2.17 RDM Embedded

intorDB intorDB

2.18 FastDB

intorDB intorDB

2.19 QuiLogic

intorDB intorDB

2.20 Pico4

intorDB intorDB

2.21 Pico4v2

intorDB intorDB

2.22 Conclusion

Tabular result here!

Part II

Performance Analysis

Chapter 3

Performance Analisys Introduction

"Every database vendor cites benchmark data that proves its database is the fastest, and each vendor chooses the benchmark test that presents its product in the most favorable light" [Burleson 02].

In this chapter we will introduce the database performance analysis' problem, and so we will discuss about how to measure the performance, how to understand who is the fastest and how to choose the best database depending on your needs. This preface is not specific to in-memory databases, but can be generalized for every DBMS.

3.1 Impartiality Problem

The most important problem related to performance analysis, and benchmarking, is the validity, understood as impartiality, of the results. It's im-

possible to develop a benchmark which produces the same results obtained by real applications, and it's very hard to have the results similar to real performance too. Moreover every benchmark may produce valid results only for a small set of applications, and therefore there is the need to use different benchmarks. All these difficulties were emphasized by vendors who, with benchmarks wars and benchmarketing, complained for fake results, and consequently brought to a lack of trust in benchmarks. In this section we discuss about these difficulties, starting from a benchmark's definition until an analysis of the main differences in benchmarks which produce different results.

3.1.1 Benchmark Introduction

Before starting all this discussion, it's necessary to explain the benchmark's meaning in computer science. The term benchmark refers to the act of running a program, or a set of programs, against an object in order to evaluate its performance, but it is also mostly used to represent the program or programs themselves. In this work we refer to software benchmarks run against database management systems. The main purpose of a benchmark is to compare the performance of different systems across different platforms. With the advance of computer architecture, it is become more difficult to evaluate system's performance only by reading its specifications. Therefore a suite of tests is developed to compare different architecture. This suite of tests, intended as validation suite, is also used to verify the correctness, the properties of a software.

In particular benchmarks mimics specific workload giving a good measure of real world performance. But ideally a benchmark should substitute the application simulated only if it is unavailable, because only the real application can show real performance. Moreover when performance is critical, the only benchmark that matters is the inteded workload.

The benchmark's world is full of challenges starting from a benchmark development, to the interpretetion of the results. Famous benchmark tend to become useless after some time, because the vendors tune their products specifically for that benchmark. In addition most benchmarks focus only on the speed which is expressed in term of throughput, and forget many other important features. Moreover many server architectures degrade drastically near 100% level of usage, and this is a problem which is not always taken into consideration. Some vendor may publish benchmarks at continuous at about 80% usage, and, on the other hand, some other benchmark doesn't take this problem into account, and execute only tests at 100% usage.

3.1.2 Benchmark History

As already highlighted in the previous paragraph, with the advance of computer architecture the performance analisys became more difficult: it's not possible to evaluate a system only by reading its specifications. A computer program, a benchmark, simulating a specific workload, became a solution to this problem.

The first benchmarks were developed internally by each vendor to compare their database's performance against the competitors. But when these

results were published, they weren't considered reliable, because there was an evident conflict of interest between the vendor and its database [Gray 93]. This problem was not solved by a benchmark pubblication by a third party, which usually brought to a benchmark war.

Benchmark Wars

Also when benchmarks were published by third parties, or even by other competitors, the results were always discredited by losing vendors, who complained for the numbers, starting often a benchmark war. A benchmark war started when a loser of an important and visible benchmark reran it using specific enhancements for that particular test, making them get winning numbers. Then the opponent vendor again reran the test using better enhancements made by a "one-star" guru. And so on to "five-star" gurus. Every result so obtained by a benchmark could not be considered a valid result, firstly because the vendors themselves don't give any credit to the result, secondly because the result, with its relative numbers, changes too much frequently.

Benchmarketing

Benchmarketing is a variation of benchmark wars. Due to domain-specific benchmarks there was always a benchmark which rated a particular system the best. Such benchmarks may perform only a specific operation promoting one database instead than others. For example a test may execute only one type of transaction, or more reads than writes and therefore promote an in-memory database over a traditional DBMS. To summarize, a benchmark

may simulate different scenarios favouring a particular database. Therefore each vendor promotes only the benchmarks which highlight the strengths of their product, trying to impose them as a standard. This leaded to a ploriferation of confuse benchmarks and didn't bring any benefit to the benchmark's reputation.

Although these phenomenons were drastically reduced by the foundation of the Transaction Processing Performance Council (TPC) in 1988, they are still alive nowday.

3.1.3 Benchmark Differences

It is now evident how hard is to understand which database is the fastest. Benchmarks cannot be properly used to analyze the performance of several databases and choose simply the best. Every benchmark has a bias, testing some particular aspect of database systems, such as writes, reads, transactions and so on. Moreover every benchmark operation can be implemented in different ways by the same database: creating a connection for every operation or using a connection pool; use different transaction isolation level; load the whole database in RAM or split it in different hard disk partition; etc. Furthermore the same benchmark, with the same implemention for every database, can show dissimilar results when it runs on different platforms (hardware and software).

All these differences are grouped by three categories:

1. Test scenario: reads, writes, etc.

2. Test implementation: there are different way to implement the same transaction.

3. Execution platform.

3.1.4 The Axiom

By this reasoning comes out the axiom which will guide the following chapters, and the work behind this thesis:

"There is not a slower or a faster database: databases are only slower or faster given a specific set of criteria in a given benchmark".

This sentence comes from an article by Bernt Johnsen [Bernt 05], who, in response to a benchmark comparing HSQL and Derby, stated that it's easy to make a benchmark, but it is always hard to communicate the meaning of the results. The interpretation depends on too many criteria, so that it's not easy to say which database is the fastest, unless specifying the set of criteria used in the benchmark.

3.2 Measures

Even if we cannot state which database is the best by analyzing their performances on standard benchmarks, we can still measure other parameters that can be later interpreted to choose the database system which best fit our needs.

When choosing a database, the most important features to evaluate and compare are:

Throughput : is the number of transactions per second a database can sustain. This is the most important feature to consider when evaluating a database system. The most representative application scenario to understand the meaning of the throughput is an on-line transaction processing system. This kind of application requires the database to sustain a certain number of transactions per second, based on the number of users, and not every database can suite the needs of the application itself. Another example is real time applications: they require even higher throughput as well as very low latencies. Therefore it is crucial to understand if a certain database is suitable for an application in terms of transactions per second.

Latency/responsiveness : is the time the database takes to execute a single transaction. This is the most important parameter in real application where the response time is a vital feature. In some case, where transactions are executed synchronizedly by a single user, latency can be measured as the inverse of throughput.

CPU load : is the average percentage of the CPU processing time used by the database. It becomes an even more important parameter when the database is not the only service running on a server. CPU is a precious and limited resource, shared by all processes in a particular machine, and although database systems are usually deployed on dedicated servers, embedded databases live collated with the application that uses them. Many in-memory database don't even have a "stand aone" version and are mainly used as embedded db.

Disk I/O : is the measure of the amount of data transferred to and from the disk. It is often a bottleneck for every traditional databases. Altought pure in-memory databases never access to the disk, when adding durability through a transaction log file, disk I/O is a bottleneck for IMDBs too. This parameter could be considered less important since all databases will incure in the same performance bottleneck, but it is possible that different DB coudl use the disk in different ways and suffer more or less impacts from it.

File size : is the size of the database image on the hard disk. While traditional DBMS' store objects (tables), indexes and a transaction log file on the file system, in-memory databases usually store only a journal file containing all the transaction executed on the database. Altought this may seem a small file, it can become very huge, even more than the database image. This measure is interesting whereas each database use different data structures.

RAM usage : is the quantity of RAM a process uses while running. Talking about in-memory database, this can be another way to measure the size of the database image. In addition, this is a critical value to take in mind: IMDBs only works correctly and efficiently under the hypothesis that the RAM is enough to contain the whole database. Therefore this can be an important parameter to take into consideration when deciding if a DBMS fits your requirements.

Startup time is the time the database needs to become operational. This is a very important parameters for applications which need high avail-

ability, and in this case the startup time plays a crucial role to respect the maximum time the application can be off-line or not completely operative.

Shutdown time is the time the database takes to shut down and kill the process.

3.3 Choosing a Database

From the previous sections, it's now clear how difficult it is to use benchmarks to prove which database is the fastest. Even with a fair benchmark, which can be very useful to understand the performance of databases, it is still difficult to choose a database: performance is only one factor to consider when evaluating a database [Burleson 02].

Other factors to consider are:

- The cost of ownership.
- The availability of trained DBAs.
- The vendor's technical support.
- The hardware on which the database will be deployed.
- The operating system which will support the database.

In other words, it is very difficult to choose the right database for our needs, and, of course, while evaluating databases and benchmarks there is absolutely *no winner*.

For this reason, what we will introduce in the next chapter is a suite of tests which has been developed with these concepts in mind. What we tries to achieve is a way for people to judge, analyze, verify and evaluate any DBMS (especially in-memory databases) with a chance of developing their own set of test and run the test suite on any hardware and under (virtaully) any operating system. The application is Java-based and so it ansures the widest coverage of both hardware and operating systems.

Chapter 4

Test Suite

In this chapter we are going to discuss the first of the three big differences, described in paragraph 3.1.3, which make a database faster or slower: the test scenarios used to run a benchmark and to analyze database's performance. These test scenarios will be used to build test cases, a collection of different scenarios, one or more, executed concurrently. All these test cases will compose the test suite we are going to use for database benchmarking. Therefore in this chapter besides a tests' description, a suite of test will be created, allowing us to analyze the databases' features of our interest.

The tests are divided into three categories: base test case, load test case and acid test case. The first category contains simple tests configured as a race between databases, and it is used to obtain an approximate idea in an early stage. Instead, the second category simulate real application scenario, and it can give us a detailed analysis of the real database's performance. Finally the last category is built to understand if the database is ACID compliant.

4.1 Why Define Test Scenarios

The axiom, previously described in paragraph 3.1.4 at page 39, expresses clearly the difficulty to analyze databases' performance and how every result obtained in a given benchmark depends on the specific set of criteria used in the benchmark itself. In paragraph 3.1.3 there is also a description of the three major categories in which the criteria fall. The first of these categories is test scenarios: different test scenarios may show completely different performance results. It's not possible to avoid this behavior, but we can define clearly every tests so that we will be aware of the differences between them.

Moreover we can model these tests in a way they simulate real application scenarios, so that real performances will be very close to the results obtained by the tests.

4.2 Base Test Case

Base test case is a collection of very simple tests, which are configured mostly as a race. This is exactly what the majority of benchmarks does, particularly Poleposition [Grehan 05].

Every test can execute different read/write operations on the database, such operations are grouped into *tasks*. The word *task*, used extensively in the following paragraphs, refers to a collection of operations. Each task is then repeated, and since all the operations in a task are synchronized and executed consecutively, the number of *task per second* is the same of the *transaction per second* of each operation involved by the same task. Therefore *task, or*

task per second and *transaction, or transaction per second* can be used in this contest without any distinction, and they all refer to the operations' *iteration*.

The key features of these kind of test are:

- A fixed number of tasks' iteration before the test stops: so tests are configured as a race where every database must execute a certain number of transaction.
- A fixed amount of time before the test stop: as an alternative to a fixed number of transaction, every test run for a specific amount of time, executing the maximum number of iteration.
- Different kinds of objects: tests must be able to create/retrieve/update/delete simple flat objects with few fields, or complex flat objects with many fields, or still hierarchical objects, and so on. This feature let us test effectively the performance on the objects used in the domain of our interest.
- Single task: to keep these test simple it's better to avoid concurrent tests, but it is still possible to implement concurrently the different operation executed on the database.

4.2.1 Race Test: Timed

We already said base tests are configured inherently as a race. These tests can be used to show the maximum throughput the database can reach doing a particular operation on a specific object. Metaphorically it's like a rocket

car in the desert trying to reach its speed limit. In a real application this is rarely useful, but can give us an idea of the database's limits.

In order to create a test scenario we have to define the domain object/s involved in the test, the operations on which the test loop and when the test should stop. For example:

- The object represent a person, with only two fields: an id number and a name. This is a simple flat object.
- The only operation executed by the test is a write operation: every time an object will be added to the database.
- 60 seconds is the stop condition of the test: after 60 seconds of iteration of writes (objects person inserted in the database) the test stops.

Clearly the time is not a significant measure, every test's execution run for the same amount of time: 60 seconds. Instead of the time, a more interesting measure to take is the throughput. This test shows the maximum theoretical value for the throughput, which in a real usage scenario will never be outperformed.

This test, and its results, are not useful to understand the real performance and potentiality of the database and so they are not useful to choose the database for our needs, but it can be used in an early stage to reduce the databases' set which will be analyzed extensively with further tests. In other words we can throw away every databases whose maximum throughput is not enough for our needs.

4.2.2 Race Test: Transactions

This test is really similar to the previous test. The only difference is in the stop condition:

- The object represent a person, with only two fields: an id number and a name. This is a simple flat object (same as before).
- The only operation executed by the test is a write operation: every time an object will be added to the database (same as before).
- 1.000.000 of iteration is the stop condition of the test: when 1.000.000 of writes are executed (objects person inserted in the database) the test stops.

While in the previous test the duration was meaningless, this time, like in a race, it shows which database is the fastest (the winner of the race). Despite this, the duration is still of little use. Also for the throughput the same considerations made before are still valid.

But this new test is useful also to take other interesting measures, such as the file size, which, as explained in par. 3.2, can become very huge, even more than the database image, and therefore it is a critical measure. We can evaluate how the file size increase with the number of objects (person) inserted in the database. This is because, differently to the previous one, every test's execution perform a fixed amount of iteration.

4.3 Load Test Case

Restrictions introduced by base tests are very simple to read and they can be useful for a first look and for fast and approximate results. These restrictions are due to the impossibility to simulate real complex application scenarios. Substantially the main restrictions are:

1. Real application scenario are rarely single thread/task: to stick to the axiom at paragraph 3.1.4, the best way to understand which database fits our needs is to make test scenarios as much realistic as possible.

2. The second restriction is a direct result of the first: trying to make test scenarios more realistic, it is necessary to introduce some sort of control for the throughput. When a test stress the database engine to its maximum level, some other mechanisms may not work properly, such as the garbage collector, caching, indexing, etc. In addition, when a test concurrently accesses the database, this restriction is even more evident: if a thread is not limited in its transactions per second, it will impact the performances of other threads, not to mention the locking that will occur on the DB.

Base test case offer very important and useful results, but when it comes to test the average load of a real application of our interest, they are not enough. These restrictions are solved by load test case, which allow a better simulation of the application scenario. The key features of these tests are:

- Multi-tasking: different task, and therefore different sets of operations, can be executed concurrently against the database.

- A bond on the throughput: in order to simulate real application usage, for every task it is possible to specify the maximum throughput in terms of iteration per second, if the database can reach it.

From these features comes the name "load test case", which means the simulation of an average, or specific, load on a certain database. This idea, and the need of this kind of tests, can simply be exaplined by a Bernt's metaphor, who has been already cited for the axiom at paragraph 3.1.4. The metaphor is:

"I don't buy the fastest car in the market. I don't even buy the fastest car I can afford. I buy a car I can afford that fits my needs... e.g. drive to the northern Norway with 2 grown-ups, 3 kids, a kayak, sleeping bags, a large tent, glacier climbing equipment, food, clothes and so on. Can't do that with a Lamborghini" [Bernt 05].

This metaphor explain exactly the need not for the fastest database, but for a database which can sustain the load of the application without any complexity during its normal functioning, such as a database snapshot freezing all writes operation, or a bug/memory leak making the database crash.

4.3.1 Real Time Prepaid System

Keeping these features in mind, we want to create a load test case to analyze exactly which performance a database system offers for a practical use case: a real time prepaid system, such as a telephone company. Basically the test is the concurrent execution of 3 different task, each simulating complex sce-

narios: services authorization and management, balance checks and accounts management.

We start describing the domain objects used by this test, and then we move on to the analysis of the different tasks involved.

Domain Objects

Domain objects involved by this test are typical for a telephone company who handle telephone calls for every person, which is rappresented by an account, and which can access to many service through a web application, offered by the company for its customers. There are four domain objects used byt this test:

Account : this object represent a customer in the real time prepaid system. It contains all the typical information needed by a telephone company, such as the balance, the type of subscription, etc. In other words, this is a complex flat object, that has many private fields but no hierarchies. In a real application there are millions of accounts instantiated in the database, one for every customer. This dimension is also very important to make the simulation as real as possible, in order to get realistic results.

MSISDN : it is the unique number which identify a subscriber in a GSM or UMTS mobile network, it is the telephone number. Each msisdn object is linked to an account. Therefore there are also millions of MSISDN objects in the database, referring to a real application.

Webuser : this represents the customer logged in the company web site. It

contains the username and the password to access to web services. In common with MSISDN, it is linked to an account too, and both are simple flat objects: a very simple object with few fields.

Session : when a customer start a call, an object session is created, and it keep all the information about the call which is going on, until the call ends. After the call this object is deleted. A session contains the start time of the call, the time of the last unit and all the relevant information for the authorization process to take place. Each session references to an account, the one who started the call. But there are not as many sessions as accounts, not everyone will start a call at the same time, except New Year's Day! A reasonable size for the session objects is the number of hundreds of thousands concurrently going on.

Balance Check Task

This task simulate a customer checking his residual balance for the prepaid card, a SIM in the case of the telephone company. First, the customer logins in the website or calls the dedicated number, and then receives all the details on his balance.

Each task, as already explained, corresponds to a transaction composed of different operations. This leads to illustrate how the task work in terms of operations executed:

1. *Read* randomly the MSISDN if the customer makes a call or the webuser if he access to the website.

2. *Read* the account referenced by the MSISDN or webuser, and then

check the residual balance, a simple account's field.

So this task executes a total of two read for every iteration.

Another important parameter to make this task a part of a load test is the amount of transactions per second this task should sustain. The average throughput for check balance task, considering there are millions of accounts in the database, is about ten transactions per second.

Accounts Management Task

The account management task simulates the subscription of new customers and consists pf the creation of an account object and the relative MSISDN and webuser.

The operations involved by this task are:

1. *Write* of a new account: all informations of a customer are inserted in the system by creating a new account object.

2. *Write* of a new MSISDN, containing the telephone number of the new subscriber.

3. *Write* of a new webuser with username and password for the customer.

To sum up, this task executes three writes on the database for every iteration.

To simulate a real scenario, the average throughput is about ten transactions per second.

Services Authorization and Management Task

This task simulates a call started by a customer. After checking the balance, and therefore if the account is allowed to start a call or use a service, a

new session is created and updated every 30 seconds, the unit time. On the average a call lasts about two minutes. When a call ends the session is deleted.

In terms of operations executed on the database, it can be described as follow:

1. *Read* the MSISDN of the customer starting the call.
2. *Read* the account referenced by the MSISDN, and its residual balance, and the other parameters, to check the user's permissions for the service requested.
3. *Write* a new object session and *update* account's balance.
4. *Update* the session every 30 seconds and the account's balance.
5. *Delete* the session after the call is ended.

This is the most complex and important task, because it represents a very frequent task. No wonder if the average throughput is about 2000 transactions per second.

Real Time Prepaid System Summary

To sum up this whole load test we have to define the objects used by the test, the task concurrently executed, and when the test stop.

There are four domain objects involved by this test. Three of these are simple flat objects (objects with few private fields): MSISDN, webuser and session. The forth is a complex flat object (objects with many private fields): account. To test the system in its normal operational scenario we will have

the system preloaded by some millions of objects before starting the actual test.

Three task are executed concurrently. One is the major task, simulating a customer's call, and it runs 2000 times per second. This task is the bottleneck of a real application, and it could be also tested alone in a base test to understand the database's limits in running this task, and therefore to have an idea of the database potentiality. The other two are secondary tasks, in fact the throughput is limited to ten transactions per second.

This test is a load test and therefore we are not interested in stopping it after a certain amount of transactions executed. But what we want is to let the test run for many minutes until to many hours to understand if the database can sustain the load generated by the test.

4.4 Acid Test Case

Base and load test case are used to analyze databases' performance, but, as already said, a database is not made only by performance: there are many other parameters to take into consideration before judging a database, such as ACID properties. To make these tests an all-round test suite, we could add some tests for verifying databases' ACID properties. The above corresponds to the work done by the Transaction Processing Performance Council with their TPC-C benchmark [Council 07]: *performance is not the only benchmark's goal, but ACID properties are tested too.*

Especially the durability property is most interesting to analyze when talking about in-memory databases. Pure IMDB have no durability at all,

while it can be added in different degree, as already described in paragraph 1.5. Therefore would be very useful to know how strong is the durability solution implemented by the in-memory database, although listening to vendors' words their database management systems should offer always strong durability.

Nevertheless, testing ACID properties is not a simple goal to achieve. Particularly the durability is the hardest property to be tested. Also the tests developed by TPC are not intended to be an *exhaustive quality assurance test.*

Chapter 5

Database Benchmark Softwares Overview

The difficulty to analyze objectively databases' performance has already been exapalined and understood in chapter 3. The major problems were found in the differences of test scenarios, implementation and execution platform. In the previous chapter we tried to minimize the first issue by modeling different test scenarios like real application scenarios. This is not a panacea, but in this way results produced by these tests are more valid for the applications they simulate than generic tests.

The focus is now moved on the second and third issues: the tests' implementation and the execution platform, in other words an application benchmark used to run test scenarios. In this chapter we analyze more or less deeply different open source benchmarks trying to find a benchmark for our needs, or, in the case it's not suitable, to take some idea for a future development of a new benchmark application.

5.1 Benchmark Requirements

Before starting the analysis, it's necessary to make clear the requirements a benchmark should have to run our tests. These requirements will give us a set of parameters which can be used to evaluate properly a benchmark. But before stating all the requirements, it's important to point out some general characteristics for a benchmark:

- Not only a single metric: a benchmark should show the results with different metrics in order to provide a better awareness of the results themselves.
- *The more general the benchmark, then less useful it is for anything in particular*: it's impossible to make a benchmark for everything, it's better to put some limitations and produce proper results for a specific task.
- Based on a workload: only the essential attributes of a workload should be used to create a benchmark.

These general characteristics and some of the following requirements are taken by the TPC presentation at the sigmoid conference in 1997 [DeWitt 97], a bit old work, but this council is still active and of course they know how to make a benchmark.

The requirements we are going to analyze are: portability, understandability, flexibility, detailed report, visual report, both for relational and object databases and easy to use. In addition we will illustrate some benefits a benchmark should bring.

5.1.1 Portability

A benchmark is a set of tests runned against a system, a database in this case, to analyze the performance with the main purpose to compare them. The comparison is usually between different systems when you are choosing the best in term of performance for your needs. But when you already have chosen the system it is also very useful to compare the benchmark's results on the same database between different platform. In order to make this possible, both the system to be tested and the benchmark must be able to run on several platforms. While databases usually run on different operating systems, it's not the same for benchmarks. When the benchmark runs on a client's network, and therefore it is for client-server databases, it can be considered portable, because the execution platform doesn't matter. But when we are using embedded databases, the benchmark itself must live togheter with the database system, and therefore able to run on different OS, which means it is portable.

Thus, when testing embedded databases too, a portable benchmark can help us not only in the decision of the best database system given a certain platform, but also in choosing a suitable platform for our database.

5.1.2 Understandability

This feature may seem obvious, but it is not when we are reading thousands of incomprehensible words or graphs with many lines and no legends or captions on the axis. The capability of being understood is essential to make the benchmark useful. Understandability is intended in terms of clear re-

sults and their easy interpretation: as already explained in paragraph 3.1.2 benchmarks' results are easy to be tricked and cheated, and therefore they must be clear to avoid any religion's war and consequently the distortion of the results.

Moreover, it's important to take in mind there are different stakeholders interested in benchmarks' results: engineers working at the database engine, managers selling the system and customers interested in buying a database. Thus a good benchmark must be able to be easily understood by many people and provide a clear view or views of the results.

5.1.3 Flexibility

This is not a requirement for every benchmark, but it is for our purpose. We want a benchmark which can run the test suite described in chapter 4. That is the collection of tests we gathered and we want to run against a database system to understand its performance. Therefore we need a benchmark which with few programming can execute our tests.

Furthermore, recalling the axiom in paragraph 3.1.4, benchmarks' results may depend on too many criteria, and therefore a good benchmark must be flexible in order to simulate the real application scenarios we are interested in, scenarios which we may add in the future. Although we said in this paragraph's introduction a benchmark should not be general because it becomes less useful, it is also true that a good benchmark must be based on a workload, the workload of the application which will use the database system we are going to choose. Therefore flexibility is a key feature for a benchmark

when comes the need to run a own test suite based on the workload of our application.

5.1.4 Detailed Report

Recalling the first general feature for a benchmark in the introduction of this paragraph ("not only a single metric"), results must provide different measures allowing a better understanding of databases' performance. It's not by chance we are talking about understandability. These two characteristics are strictly connected. Of course many measures for every test will make the result harder to understand but will provide more informations to interpret the databases' behavior. On the other hand only one measure is easier to understand but may hide the database behavior and the reasons behind that. Therefore a compromise is needed: in each test we are not interested in all the measures it's possible to take, but only a subset of those can be taken for every test. The reference measurements have been already introduced in paragraph 3.2 and they represent a subset of all possible measures for a database.

In addition more measures the benchmark's application takes, more it alters the database performance, and therefore a compromise in the number of measures is always a good choice, but keeping in mind that only one measure will not tell much about databases performance.

5.1.5 Visual Report

In order to make the results even more understable, and to make the detailed measures more readable, visual reports will play an essential role. Graphs are able to express thousands of numbers in one or few colored lines. Graphs are perfect not only to show the final result of a test, but also the whole trend, from the beginning to the end. Therefore any oscillation in the measures can become evident and easy to read. It will not be hide by an average value reported at the test's end. Of course graph can also be almost useless if they report only the final value. It depends by the implementation. For example graphs such as those from Poleposition benchmark, which will be analyzed in paragraph 5.5, may be very hard to understand and they simply describe an average value.

Furthermore visual reports are usually used with the main purpose to compare system between them, and this is another task they play perfectly. Of course there are many parameters to understand before being able to read properly a graph, but then in a simple graph every line can represent a different database system, allowing an easy comparison between them. This is really appreciated by non technical people, such as managers or customers. For this purpose graphs are a must, and therefore this is a feature we want from a benchmark application.

5.1.6 Both Relational and Object Database

The benchmark we are looking for must be able to work with both relational and object databases. Relational databases are usually able to run as

stand-alone server and accessed via SQL. Instead object databases are used mainly as embedded databases and accessed via a native interface for a specific programming language. Anyway in-memory databases are used mainly as embedded databases and therefore we need a benchmark which can use embedded databases accessed by both SQL (eg: JDBC) and native interface. Although in-memory databases are not a new technology, they became widely used in the last few years, therefore many benchmarks do not provide any support for embedded databases.

When testing both relational and object databases another problem comes out: the benchmark application can test the database systems in different ways, using a relational point of view or an object oriented point of view, or both. This may produce different results in the performance analysis. The knowledge of this feature is important during the results' analysis.

5.1.7 Easy to Use

This is a commonplace for every application, nobody wants something impossible to use. As for other requirements, a compromise is always needed, especially for this feature. The ease of use, in this case, means the possibility to modify several test's parameters, such as the parameters already explained in chapter 4, and also to be able to bring some little variation to the workload simulated, in order to improve the test when the needs change. This ease of use, understood as a variation in parameters, should be granted without any modification in the application's lines of code, but for example in an xml file, or other configuration files, or with some input parameters.

Of course this may seem similar to the flexibility requirement, that we discussed in paragraph 5.1.3, but while flexibility is intended as the capability for engineers to implement completely new test scenarios, ease of use is the capability to execute the workloads implemented with few variations, such as the initial database's state, the exact operations' order, etc. This requirement is needed for all the non technical people who want to analyze the database performance.

5.1.8 Benefits

Discussing about requirements, it's important to note also the benefits a benchmark should bring, the aims it is built around. Referring again to the TPC presentation at the sigmoid conference in 1997 [DeWitt 97], the benefits a good benchmark should bring are:

- Definition of the playing field: a benchmark should clearly point out the database performance and the hypothesis behind these performance.
- Progress's acceleration: a benchmark allow to measure the database's performance and therefore engineers are able to do a great job once an objective is measurable and repeatable.
- Schedule of the performance agenda: this is something also managers and customers can understand. Every release can have clearly declared its goals, such as X increment in the throughput, and it is easy to measure realese to realese progress.

The TPC work illustrate also how good benchmarks have a lifetime, because they firstly drive industry and technology forward, but when all reasonable advances have been made, benchmarks can become counter productive, encouraging artificial optimizations. But this reasoning falls down when the workload, on which the benchmark is based, changes. And the modification of the workload, from little modification to completely new implementation, is a requirement for the benchmark we are looking for.

5.2 The Open Source Database Benchmark

We start now the analysis of several database benchmarks trying to understand if they are suited for our needs, or to take some idea for a future development. Discussing about these benchmarks we will illustrate advantages and disadvantages. Surfing the web and looking for databases benchmark "The Open Source Database Benchmark" is the first who comes to light.

5.2.1 Advantages

First of all this database benchmark is open source. All the benchmarks we will analyze are open source, therefore it sounds a bit strange to point out this feature as an advantage. But this database benchmark has so few advantages that it's good to list them all.

Instead a better feature to point out is the test suite used by this benchmark application: it is based on AS3AP, the ANSI SQL Standard Scalable and Portable benchmark. This is a relational query benchmark divided in two section: single-user tests and multi-user tests. Therefore this is a collec-

tion of SQL script which can be used against a database with a minimum effort in benchmark's programming.

5.2.2 Disadvantages

On the other hand there are several disadvantages. This project started in the beginning of 2001 and its last release is 0.21 dated at January 2006. Therefore both the latest release number and date are bad numbers: this software is outdated and it is still in alpha release. Also the AS3AP benchmark is outdated and very few information are available on the web.

From a relational point of view, this benchmark is portable, because it runs on a network client, but it is written in C language, and it is not portable from an embedded database point of view. When porting a program written in C from a platform to another, there is almost always some code's lines which need to be changed, especially when using sockets or other system calls.

In addition this benchmark is able to run only SQL script against relational database server, thus it needs a lot of programming for the procedures' implementation to test embedded/object databases. What's more we want primarily test embedded/in-memory databases and not relational database servers.

At last, another disadvantage is the lack of detailed visual reports. This benchmark works only by the command line interface and no graphs are produced. Moreover there is no direct comparison between different database systems: in order to compare two databases, manually the benchmark must

be runned against them and then compare the throughput reported. There is also no comparison in the throughput's trend during the execution of the test.

5.2.3 Conclusion

In conclusion it is clear how this database benchmark doesn't fit to our needs. Not even one requirement is completely satisfied and some of them are taken in no consideration. The reports are neither detailed or visual, and embedded in-memory databases are not supported. With a lot of programming this benchmark can also become a good one, but it's more convenient develop a completely new benchmark than extend the open source database benchmark to fit our requirements.

What we have learned from this application is very few: only the AS3AP is a nice discovery. It is an old standard SQL test suite, which can be used to understand and to learn some new kind of test, new interesting workload. But AS3AP is very old and outdated and therefore useless.

5.3 Transaction Processing Performance Council

In the past years, without a standards body to supervise the testing and publishing, vendors begin to publish extraordinary marketing claims running specific benchmarks. The Transaction Processing Performance Council, TPC, is a non-profit corporation founded in 1988 by eight leading software and hard-

ware companies in response to benchmarketing and benchmark wars. The TPC currently has 20 full members: AMD, BEA, Bull, Dell, EnterpriseDB, Fujitsu, Fujitsu Siemens, HP, Hitachi, IBM, INGRES, Intel, Microsoft, NEC, Netezza, Oracle, Sun Microsystems, Sybase, Teradata and Unisys.

The main aim of TPC was the provision of uniform benchmark tests. Therefore they, recnognizing that different types of applications have different workload, created several benchmarks. Besides the outdated benchmark which are TPC-A, B, D, R and W, there are the followings:

- TPC-App: an application server and web services benchmark. It simulates the activities of a business-to-business transactional application server operating in a 24x7 environment.
- TPC-C: an on-line transaction processing (OLTP) benchmark created in 1992; a complete computing environment where a population of users executes transactions against a database.
- TPC-E: a new on-line transaction processing workload which simulates a brokerage firm.
- TPC-H: a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications.

5.3.1 Advantages

Of course there are many advantages when discussing about TPC benchmarks, and above all there is the experience this consortium has accumulated in the last twenty years. Since 1988 TPC is working on benchmarks

expressing how benchmark results depend on the application's workload, on the system design and implementation, including the execution platform, and on the application requirements, such as high availability or strict durability. This experience, and the big names of the members, contributed to impose TPC benchmarks as a standard, especially for OLTP performance.

Talking about standards, the most famous TPC benchmark is TPC-C, the on-line transaction processing benchmark. The scenario simulated by TPC-C is very common real application scenario, and therefore all its success. This is a moderately complex OLTP modeling a wholesale supplier managing orders, whose workload consists of five transaction types:

1. New order: enter a new order from a customer.
2. Payment: update customer balance to reflect a payment.
3. Delivery: deliver orders (done as a batch transaction).
4. Order status: retrieve status of customer's most recent order.
5. Strock level: monitor warehouse inventory.

The benchmark take a measure of the throughput, that in TPC terms is a measure of maximum sustained system performance. It is based on the number of "new order" transactions per minute while the system is executing all the other transactions types. In addition, while running the benchmark, 90% of each type of transaction must have a user response time less than 5 seconds, except "stock level" which is 20 seconds.

But throughput is not the only metric used by TPC, there is also a price/performance metric. The price is simply divided by the throughput

and tell you how much is the cost for a transaction. It's important to note that this price is not the cost of the hardware, or another component. It include all cost dimension of the system environment: terminals, backup storage, servers, software, and three year maintenance.

TPC-C requires also transactions to be ACID (atomicity, consistency, isolation and durability) and therefore TPC included different tests to demonstrate that the ACID properties are supported and enabled during the execution of the benchmark. These tests are not intended to be exhaustive quality assurance tests. Therefore these test are a necessary condition, but not a sufficient, to demonstrate the ACID properties. In the TPC-C specification document [Council 07] there is the description of different kind of scenarios which can be used to test the acidity of the database system.

Lastly TPC doesn't work only for creating good benchmarks, but also a good process for reviewing and monitoring those benchmarks. A nice metaphor written by Kim Shanley in 1998, chief operating officer at TPC, says good benchmarks are like good laws: as Aristotle said, if the laws are not obeyed, do not constitute good government. And this is the meaning for the process reviewing and monitoring. Although this last concept about process monitoring is really important, this is not an interesting feature for the benchmark application we are looking for, simply because it doesn't regards the application itself.

5.3.2 Disadvantages

The transaction processing performance council offer many advantages, but its benchmarks are not exempt from disadvantages. First of all TPC benchmarks are only specification and there is no implementation available: each vendor must implement its own benchmark and then the council will review and monitor the process to check if the specification are kept. Although it is possible to find some free TPC-C implementation in internet, they are all unofficial benchmarks.

In addition all the available implementations, as for the official private implementations, are for relation database only and run SQL script. Not by chance all the TPC-C specification are expressed in terms of relational data model with conventional locking scheme. But in the introduction of TPC-C specification [Council 07] it is clearly stated that it's possible to use any commercial database management system, server or file system that provides a functionally equivalent implementation. The relation terms, such as "table", "row" or "column", are used only as an example of a logical data strucuteres.

Moreover all these benchmarks have been used for high end system and for vendors who produce both hardware and database management system. Although it's possible to use a proper implementation of these benchmarks for every kind of DBMS the comparison between the official results will be almost always of few interest: because the official results regards only high end systems and because the results obtained by the proper implementation will not get any credit by the TPC consortium.

5.3.3 Conclusion

To sum up the whole discussion about TPC benchmarks, especially TPC-C, it's evident how this is not suitable for our needs, but instead all this work can be used as an inspiration. There are different reasons why this doesn't fit the requirements previously exaplined. The council itself wrote in the TPC-C specification [Council 07] a sentence that clearly state why this is not what we are looking for:

"Benchmark results are highly dependent upon workload, specific application requirements, and systems design and implementation. Relative system performance will vary as a result of these and other factors. Therefore, TPC-C should not be used as a substitute for a specific customer application benchmarking when critical capacity planning and/or product evaluation decisions are contemplated".

This is an emblematic sentence and it can be considered as an extension of the axiom enunciated in paragraph 3.1.4 where we said that performance depend on a huge number of criteria. The meaning of this sentence is that TPC-C is a good benchmark used to compare different systems for a generic purpose OLTP, without critical performance requirements. In this case, you need a benchmark which simulates the workload of your application, or the application itself. While TPC benchmarks are not flexible, they can't be used to run our test scenarios. In fact TPC-C benchmark is simply a complex application scenario, a benchmark specification, and not an application benchmark, which we could eventually extend to run our test suite.

But this benchmark and this analysis is not useless. This is a source of

inspiration for an extension of our test suite: from the implementation of a new load test case similar to TPC-C (it is a standard de facto for generic OLTP); to the implementation of some, not strict, ACID test. Moreover the TPC work faced many problems in the last year in benchmark's definition, accumulating a lot of experience, and this is a part of our work.

5.4 Apache JMeter

Apache JMeter is an Apache Jakarta project, a set of open source Java solutions and part of the Apache Software Foundation, which encourages a collaborative, consensus-based development process under an open software license. Apache Jmeter is designed to load test functional behavior and measure performance. Originally developed with the main purpose to test web applications, it has expanded to other resources, both static or dynamic: files, servlets, perl script, java objects, databases, FTP servers and more.

5.4.1 Advantages

Apache JMeter is a 100% pure Java desktop application, and therefore allowing complete portability. In addition JMeter doesn't need any installation, it's possible to download the binaries and simply execute them. This is perfectly suited for our need to test in-memory database on different platforms. Also most of the database systems we will test are written in Java e mainly for Java. In addition JMeter offers a powerful GUI which let you set your test in an easy way, and therefore ease of use is accomplished.

A key feature of JMeter is the capability to load test: there are not only

the classic stress tests but load test too, which can simulate a group of user and different load types. Tests in JMeter are represented by a test plan that is a collection of user's behavior. Each user usually executes different actions, such as a SQL query or a HTTP request, simulating therefore a certain behavior. For each user is also possible to specify several parameters in order to simulate a particular load. Figure 5.1 is an example of a JMeter configuration test for a database via JDBC. It shows some of the features previously described such as the thread group, which is a set of users with identical behavior.

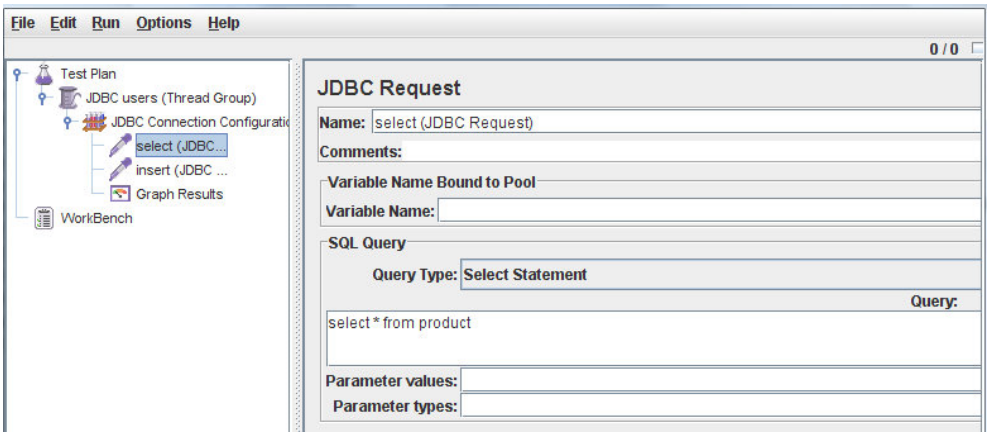

Figure 5.1: JMeter GUI

Moreover these load tests can also be executed concurrently by JMeter and in two different ways:

- Each thread group can simulate a collection of users with the same behavior. Each user is a thread, and the amount of thread is a parameter of the thread group. Therefore it's possible to simulate different users with the same behavior.

- It is also possible to implement different thread groups and so different behavior.

The ability to execute load test and concurrent test give us all the flexibility we need to run our test suite, particularly load test case such as the real time prepaid system explained at paragraph 4.3.

Another important feature in favor of JMeter is the capability to load test also databases via JDBC. As already said the figure 5.1 is an example of a JMeter configuration test for a database via JDBC. It shows how to simulate a group of user with the same behavior: they firstly execute a select and then an insert. The whole database configuration is setted up by the JDBC Connection Configuration where we need simply to specify the driver to be used and the usual connection parameters, such as the database URL, username and password.

In figure 5.1 there is another element which we have not still explained: the Graph Result. JMeter is also capable to represent the results in a graphical form. In figure 5.2 there is an example of a very simple graph obtained during a database load test (this image is taken from the Apache JMeter tutorial). This Graph Result is only a specific Listener which is possible to configure in the test plan, but there are many other listeners such as Table Result or Monitor Result and so on. It is also possible to execute JMeter from the command line interface, saving precious resources while running the test, logging all the results in a file, and then use a specific application to represent them in better ways.

Last but not least, JMeter is based on a plugin architecture and therefore it is highly extensible. Every element can be extended. For example a new

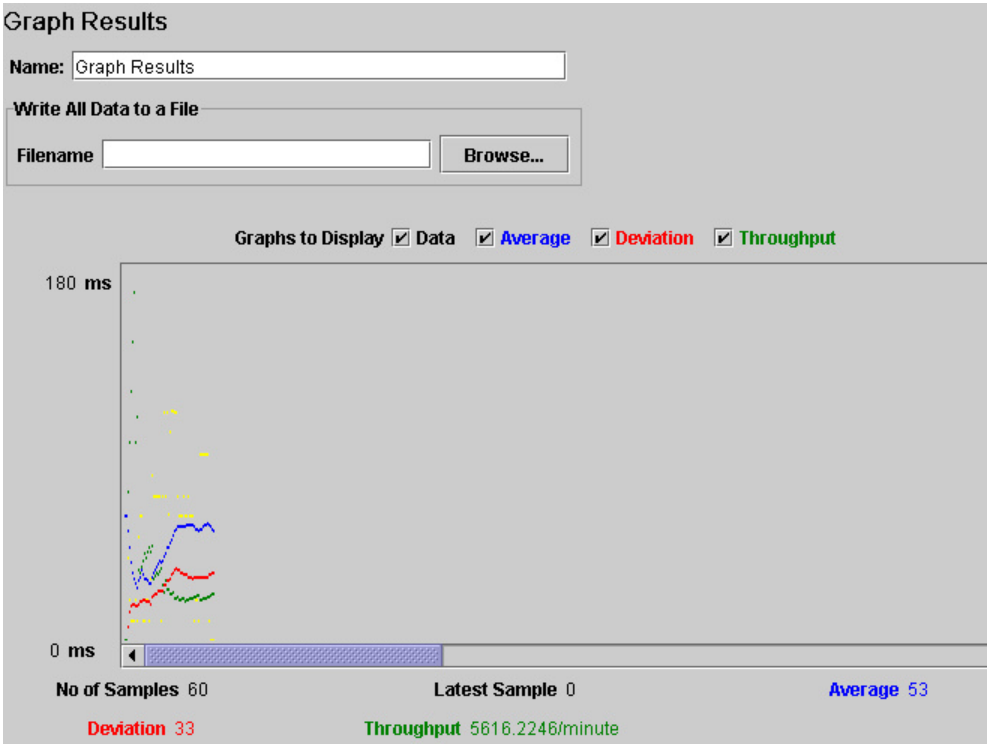

Figure 5.2: JMeter Graph

Listener with a particular graph can be added, and also new Timers. This feature makes JMeter really flexible, and so another requirement is accomplished.

5.4.2 Disadvantages

To make this analysis complete, we have also to evaluate the disadvantages of JMeter. And, as first thing, it's evident how JMeter doens't allow load testing of in-memory databases or other kind of databases which don't have a JDBC interface. It only acts with databases through SQL, and it is quite obvious: at the moment there is no a standard for database native interface. Therefore, although JMeter works with relational databases, it will not work with all the in-memory databases without a proper extension, an implementation of a new plugin, because there is no plugin available for our needs.

Moreover JMeter doesn't compare different systems between them, but instead it measures the performance of one system at time. Therefore there is no direct comparison, while the comparison is our main purpose. In fact JMeter was originally designed to test web applications in order to find the best tuning for the web server and the relative application. This means that JMeter is not used to find the best web server with the same application, which in other words it can be said that it is not used to find the best database for our application scenario. Nonetheless it can be also used to compare different systems, but not with much ease of use.

Also the visual representation of the result, the Graph Result, is not that great, and of course JMeter is not famous for graph results. It is not very

fluid, and it has some bugs, such as when the test time is too much the graph goes in overflow.

5.4.3 Conclusion

Now, after the analysis of advantages and disadvantages, we can conclude this investigation about Apache JMeter. It seems a very interesting application benchmark with a lot of features which meet most of our requirements such as:

- portability, because it is written in Java and it is a desktop application;
- flexibility, which comes from the plugin architecture;
- ease of use, tanks to the GUI;
- visual and detailed report.

Although all these advantages, there is still something which is not perfect. Firstly JMeter can't test both object and relational databases, and this is an important limitation considering our main purpose is to test in-memory databases. Nonetheless JMeter can test java objects, and therefore it can work with java native interface, and however it is possible to write a new plugin to add the functionalities we need. But this solution will also add a considerable amount of work in programming the new plugin, losing partially or completely the usefulness of an already existing application benchmark.

Secondly also the graph listeners already implemented in the application are not very useful. They can only be used for a fast and approximate analysis. In order to create a good graph with the resulting data stored

in a file, there are two possibilities: implement a new graph listener or use another application to represent the result. In both cases there is again the necessity to write new code's lines.

In addition JMeter doesn't allow an easy comparison between different results, and therefore, with the purpose to compare several database performance, there is no more ease of use.

Finally, JMeter is a great application and the extension with new plugins and other elements is very attractive, especially considering this is an open source application widely used. But not only the amount of code to write may be equal or more than the code needed to write a specific application for our needs; in this way we are also forced to move in a more complex application, although JMeter is easily extensible. However, taking in mind JMeter is not the best application to compare different performance, this is not a proper way to work. Nevertheless the extension of JMeter is still very interesting.

5.5 Poleposition

Poleposition is an open source database benchmark, developed to compare database engines and object-relational mapping technology. Poleposition is built to be a framework to help developers in evaluating databases performance, through a simple implementation with only few lines of code. In fact the impetus behind Poleposition came from the observation that developers evaluating candidate databases for future applications often resorted to constructing ad hoc benchmarks rather than using "canned" benchmark tests (or

relying on vendor-provided data). This is entirely understandable: to properly evaluate a database for a specific project, you would want to exercise that database in ways that correspond to the application's use of it [Grehan 05]. This is the same concept we have already explained in paragraph 3.1.3 and paragraph 3.1.4.

Poleposition use the metaphor of a race, how it is clear by its name, to help the developers in understanding the framework structure.

5.5.1 Advantages

This is another really interesting database benchmark, full of advantages. Above all, again, it is a Java desktop application, without the need of any installation, except for eventual database servers. Therefore the portability specified as first of our requirements is granted, allowing the test of database systems on different platforms.

In addition also the flexibility is provided by Poleposition: in fact it is a framework, for database benchmarking, allowing the implementation of new tests and of new database systems. The framework is also simplified by the metaphor used to describe the whole application,and which is evident by the name itself. This application benchmark is configured like a championship car racing, where:

- A *circuit* is a set of timed test cases that work against the same data.
- A *lap* is a single (timed) test.
- A *team* is a specific database category or engine that requires specific source code.

- A *car* is a specialized implementation of a team, and therefore every database system can use several implementations.
- A *driver* is an implementation of a circuit for a team .

In favor of Poleposition there is also the reporting tool: results are available both in HTML format and in a PDF file, providing a fast and clear idea of the results obtained. The PDF file and the HTML result are a collection of circuits, which actually are:

- Melbourne: writes, reads and deletes unstructured flat objects of one kind in bulk mode.
- Sepang: writes, reads and then deletes an object tree.
- Bahrain: write, query, update and delete simple flat objects individually
- Imola: retrieves objects by native id.
- Barcelona: writes, reads, queries and deletes objects with a 5 level inheritance structure.

These circuits are a collection of laps: a write, a read, a delete etc. And each lap show the numeric results with both a table (table 5.1) and a graph (figure 5.3).

The table 5.1 shows the time the "write" lap takes during the melbourne circuit. For each car/team is shown the time in milliseconds they take to execute a certain number of writes: 3000, 10000, 30000 and 100000. Therefore it's easy to compare them and understand who is the fastest, the winner of

t [time in ms]	objects:3000	objects:10000	objects:30000	objects:100000
db4o/6.4.14.8058	379	7365	1398	4820
Hibernate/hsqldb	716	799	2617	14916
Hibernate/mysql	1442	2948	9437	35853
JDBC/MySQL	2881	1872	5470	18422
JDBC/Mckoi	1202	2842	9371	33530
JDBC/JavaDB	970	907	5454	8676
JDBC/HSQLDB	276	57	199	931
JDBC/SQLite	20381	46545	138990	483594

Table 5.1: Poleposition time table: melborune circuit and write lap

the race. The same result is also represented by a graph, figure 5.3, which clearly shows which database is above the others. To note how on the y axis the time is in a logarithmic scale and therefore higher is the value and faster is the database system. On the HTML result there is also another kind of chart, which measures the memory consumption, but on the PDF file this is not present.

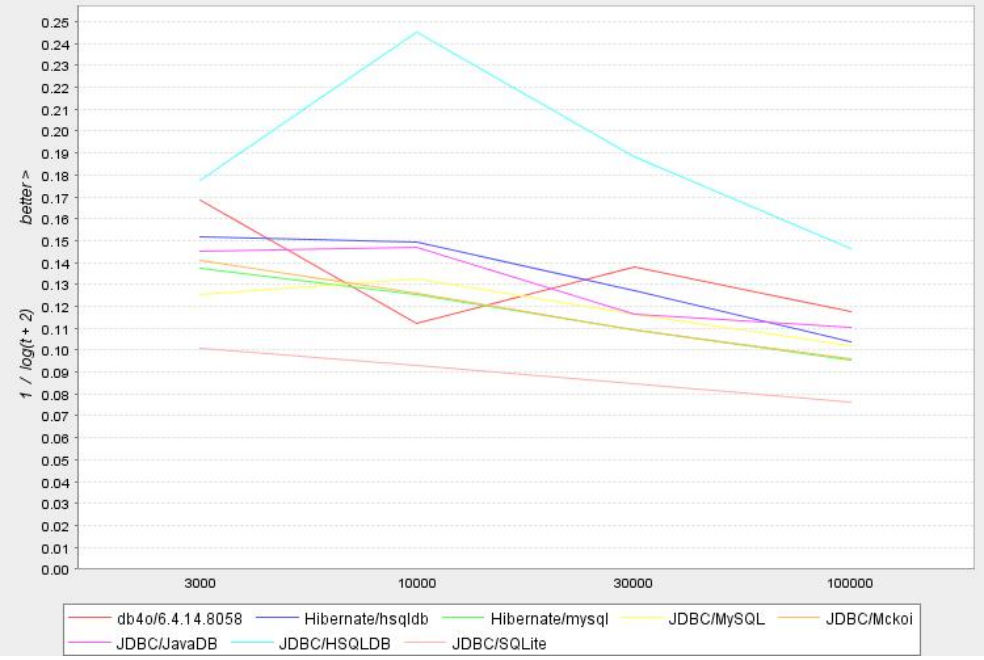

Figure 5.3: Poleposition time graph: melborune circuit and write lap

Poleposition can also be used to test both object and relational database systems. This is possible because it was developed with the aim to compare database engines and object-relational mapping technology. Hence this makes poleposition a really interesting database benchmark for our needs.

Finally, as last advantage, there is something that is not properly a feature of the application, instead it is an idea the programmers took in mind during the development of the application, and therefore it reflected on the quality of the application. The idea we are talking about is the following: *"Objectivity must be the most important goal of the benchmark tests. The results show very clearly that there can not be one single best product. Depending on your application, some circuit results may be very important for you and some may be completely irrelevant."* This statement perfectly accords to the axiom described in paragraph 3.1.4, and on which is based our work.

5.5.2 Disadvantages

The metaphor may help to understand the initial problem of the benchmark, but on the other hand, it becomes a trap, a limit for the benchmark itself: if they continue with the car race they will soon run out of gas [Grehan 05]. Any further extension will be forced to follow the rail imposed by the metaphor. Moreover this metaphor makes the benchmark a simple race, and there is no load testing.

Although poleposition's output is very catchy, it is not easily readable and it lacks of detail: time and memory are the only measures taken in the HTML result and in the PDF file only the time is represented. In addition

every test and graph/table lack of an explanation of what it is testing and representing, making the comprehension both of the test and the result very opaque.

Moreover the results are always too much summarizing. For example in figure 5.3, and thus also in table 5.1, only the total time of the test is represented, and so we can calculate the throughput dividing the number of objects inserted with the total time. But it's not possible to know if this throughput was constant or growing or oscillating too, there is no representation of the throughput's trend during the execution of the test. This contributes to make the test even less clear, hiding the real database systems' behavior. For example, looking at figure 5.3, we can't understand why HSQLDB and Db4o are so much oscillating. Of course it can be the real database performance, but it could also be a mistake in the test implementation or in the database configuration. Certainly a more detailed report of the result could be helpful.

Furthermore this framework only allow single-user testing, because, as the authors said in [Grehan 05], *"some sort of multi-threaded test to simulate multiple user would be exceedingly difficult without extensive modifications to the framework. But if that could be done, PolePosition could provide enterprise-level testing"*. Nevertheless they hope in the future to introduce this new functionality, but at the moment it is still not working.

Finally this database benchmark is quite recent: it started in 2005 and actually there is still only one release available for download, posted in June 2005. Looking through the SVN repository on sourceforge, it's possible to notice some new changes and activity in the last years, but there is still no new release available for download. In addition the code of poleposition is

not that clear and object oriented as they said in [Grehan 05].

5.5.3 Conclusion

As first impact Poleposition seems to be exactly what we are looking for:

- the portability is granted by the Java language;
- new test can be implemented allowing flexibility;
- there are very catchy graph report;
- and it can works both with relational and object databases.

But a deep analysis of this feature showed that it is not suited for our needs. Particularly the results are very poor, and the graphs don't show any database behavior during the execution of the test: in fact these graphs shouldn't be line charts, but it would be better if they were bar charts, because they don't show any trend.

But a very important disadvantage, which exclude Poleposition from our choice, is the lack of load testing, also due to the incapability to execute concurrent tests. This is a major disadvantages, especially when we want to simulate real application scenarios, which is a consequence of the axiom on which is based our work.

In addition this benchmark is not easy to use, although it may seem easy. It's impossible to understand the meaning of the results without looking at the code: *"if you consider basing your choice of database on this benchmark, bring along lots of time to look at what every test does. You will not be able to judge the results without looking at the source code that produces them."*

And therefore this benchmark is not for non technical people who don't know Java language, but also for engineers is not so easy, because they need a lot of time to understand properly the benchmark results, looking through the code's lines.

It could be possible to extend this framework with the features we need and the authors themselves exhort this possibility. On top of that there is a substantial amount of code to write and the framework code is not simple and easily extensible: for example there are methods with hundreds of line of code and with many many nested conditional instructions, which is not object oriented programming, losing all the maintainability such a programming paradigm offers.

However this application is very catchy and it is a great source of inspiration for a future development. Very interesting features, although they are not perfect, are the report in a PDF file, the graphs, and the idea to make a database benchmark framework.

5.6 Benchmark Overview Summary

Coming to a conclusion we sum up the pros and cons of the database benchmark applications analyzed in the table 5.2. It shows only the most interesting negative and positive features we found in these benchmarks, in order to understand what we need and take inspiration for a future development of a new benchmark application.

The starting idea was to find an application with the requirements explained in paragraph 5.1, so that we could be able to run our test suite. But

Benchmark	Pro	Con
The open source database benchmark	Open source AS3AP	SQL only C language No comparison No visual reports Latest release: 0.21 (2006)
TPC-C benchmark	Standard Simulate OLTP systems Price/performance metric Acidity testing For every DBMS	For high end vendor No implementation available
Apache JMeter	Java desktop application Load testing Plug-in architecture JDBC plug-in	No plug-in for IMDB No comparison Bad graphs
Poleposition	Java desktop application Database benchmark framework Both relational and object DB Results comparison Catchy report	The metaphor is a trap Lack of detail in results Only single-user testing Quite recent

Table 5.2: Benchmark summarizing table

the final result is that there is no benchmark which perfectly fits our needs. This means there are only two possibilities to execute our test suite:

- choose one of the benchmarks previously analyzed and then extend it, in order to add the functionalities we are looking for;
- develop a new benchmark starting from our requirements.

The first idea was already taken into consideration during the benchmark analysis and rejected in the conclusion paragraphs. In summary:

- The first benchmark, the open source database benchmark, was completely unsuitable. Not even one requirement was satisfied. The only positive aspect was the use of AS3AP, the ansi SQL standard scalable and portable benchmark (portable only for relational database server), which can be helpful to the creation of new test case.

- The transaction processing performance council, with their TPC-C, is very interesting because it is a standard and it can simulate a generic OLTP system. But there is no implementation available, and anyhow this will not help us to execute are test suite. Instead this can be a good source for inspiration, both for a new load test case and for some ACID test case. So it is a nonsense talking about extension.

- Apache JMeter is a very good general purpose benchmark application and offers a plug-in for testing relational database server through JDBC. Therefore to use this tool for our aim we need to write a new plug-in, which may not worth the effort, especially considering that JMeter allows an efficient measurement of the performance, but not an easy comparison between different system, and moreover the graphs are quite poor, although extensible.

- Poleposition, on the other hand, is suited for both relational and object database systems, but its testing capability are quite poor. The lack of concurrent testing, and therefore of load testing, prevent us to execute our test suite. A possible extension of this application is not the best solution, especially looking at the source code which is not well organized.

Therefore the best solution is to develop a new benchmark application. Although this decision, the analysis done is not useless, in fact we understood which features can be used from other benchmarks, and how to develop them, and what should be avoided. But this will be explored in the next chapter.

Chapter 6

The New Database Benchmark Application

The previous chapter is dedicated to the research of a suitable benchmark application for our needs which is able to execute our test suite. The analysis didn't bring to any available benchmark neither to a benchmark easily extensible in order to implement the features we want. Therefore the conclusion was to develop a new benchmark application, starting from our requirements and treasuring the knowledge acquired during the analysis.

In this chapter we describe the new benchmark application developed for our needs, starting form the specification defined from the requirements and then describing the application from different points of view. Subsequently we will discuss also about the application usage, from the application's configuration to the extension and the implementation of new databases and tests.

6.1 Analysis and Design

This section is dedicated to the analysis and design of the benchmark application developed during this thesis. We start from the analysis and therefore from the specifications' definition, which come from the requirements described in paragraph 5.1. Then the focus is moved on the design, describing and representing the application by different point of view: functional, information, concurrency and development.

6.1.1 Requirements Specification

In order to define the specifications from the requirements on which the benchmark application is based, it's primarly important to understand what is a specification in relation to requirements. A requirement is a singular documented need of what a particular product or service should be or do. On the other hand a specification is an explicit set of requirements, or in other words it is a well-formed requirement, a formal requirement, an implementable requirement.

Now it is clear how the discussion done in chapter 5 is not useless. Through that analysis we obtained a better understanding of the requirements, and a way to implement them. This will help us in the process of requirements specification. The table 6.1 summarize the specifications obtained from each requirements.

Requirements	**Specifications**
Portability	Java
Understandability	clear PDF file with graphs
Flexibility	extensible framework
Detailed Report	several measures
Visual Report	graphs with trend
Both Relational and Object Databases	databases interface written in Java
Ease of Use	xml both for test and report configuration

Table 6.1: Benchmark specifications

Portability

We already recognized the value of portability as a requirement when testing embedded databases, because it allows us also to chose the best platform for the database we are testing and to study the difference in the performance with a platform's change. From the experience gained with the benchmarks' analysis we know that both JMeter and Poleposition grant the portability we need. And this is granted thanks to the Java language. In fact they are both Java desktop application who are able to be executed on several platform because the Java Virtual Machine allows Java to run everywhere (*"write once, run everywhere"*).

Therefore this requirement brings to a simple specification: when you want to grant portability to an application, the Java language is a good solution.

Understandability

When a benchmark is not easily understandable, it is useless. Benchmarks are too easily tricked and cheated, and thus they are meant for different stakeholders. Looking at JMeter and Poleposition we understand how a graph can help to explain the numeric results and to compare several systems. Particularly Poleposition offers a very catchy PDF result, containing all the tests with all the results expressed both in tabular and graphic results. But in this PDF what the test does is not very clear. Then the need of clear results which explain also the test's operations. For this purpose JMeter is a bit better, because with a tree structure it shows exactly the task and all the operation involved by the test.

Therefore what we need to grant this requirement is a PDF result, containing all the information about the tests:

- what the test does, in terms of operations involved on the database;
- graph result which catch the attention of the reader and quickly express the meaning of the numeric result.

This is the specification implicated by understandability.

Flexibility

Benchmark always suffers the limitation introduced by the workload simulated. The performances measured by the benchmark are highly dependant on the workload, and therefore for an application, whose workload is not similar to the benchmark's one, the results are nearly useless. To overcome

this limitation, flexibility is a key feature. In this contest flexibility is meant as the capability of the benchmark application to run complete new tests, and so to simulate new workloads. Furthermore it is also intended as the capability to run the benchmark against new database systems. Also this time JMeter and Poleposition show a possible solution to this problem, but they are not equals.

As regards Poleposition, it works like a framework, in fact the main idea behind this application is to make a framework for databases benchmarking. Therefore Poleposition allow an easy implementation for both new tests and database systems, although it is not so easy and flexible as a framework should be, especially for new tests. In both cases the solution involves the programming of new code's lines.

JMeter, instead, is able to create and execute new tests thanks to a graphic editor which can be used to create very complex scenarios. In regard to the implementation of new database systems, at the moment JMeter can test only SQL databases, but a possible solution may be the development of a new plug-in. In fact JMeter is based on a plug-in architecture, which allow a good evolution of the software.

In conclusion the resultant specification is: create a benchmark application which works as a benchmark, with tests that can be implemented by programming, and with pluggable databases.

Detailed Report

A well detailed report with a lot of interesting measures will tell you much about the databse behavior, while only one single metric will simply make

your trust go away when you read it.

The first who stated this idea was the transaction processing performance council, who, in fact, adopt two different measures: the first is the throughput and the second is a price/throughput measure. This beacuse the TPC-C benchmark doen't compare directly different systems, but every vendor run it on his own, therefore without a price/performance measure, it is very hard to understand if the result depends from more expensive computers. This is not our case, but we agree that one measure doesn't exaplain the real database performance.

JMeter, instead, has several listeners which are able to take different measures, and they can be composed in many ways inside a test plan. So JMeter allows the user to take exactly the measures he wants. And this is applicable also for graph listners.

In this case case JMeter shows a very good solution, and this is exactly what we like: the possibility for the user to chose the measures to display as final result.

Visual Report

When you face complex results with a lot of numbers, a visual representation will make them clearer. Thus when using graphs also a comparison between different systems will be easier, because it becomes a comparison between different colored lines. Furthermore graphs are very helpful if you want to analyze not only the final result of a test, but the whole trend.

Although Poleposition makes use of graphs, these don't represent any test's trend, they simply represent the final value of the test. Nevertheless

Poleposition with its graphs allow a very easy comparison between different database systems and the graphs are very catchy.

On the other hand JMeter provides very stark graphs, without any comparison between different systems. But it is possible to represent each measure on the graphs and in addition they show the performance's trend during the test's execution. Using appropriate listeners, or writing new ones, it is possible to represent almost everything.

To sum up we want graphs who let us compare different database systems and draw different measures, with the capability to analyze the trend during the test's execution, but in a catchy way. Something like Poleposition graphs with in addition the trend representation. Furthermore the user should chose which graphs create, specifying the measure to drawn on the graph's axis.

Both for Relational and Object Databases

Relational and object database system usually doesn't share the same interface. From the Java point of view, relational databases are usually accessed through JDBC, while object databases usually use native interfaces. It's not easy to make a database benchmark work with two different technologies, there is an impedance mismatch.

Poleposition is able to test both relational and object databases, but at the cost of a completely new implementation of the whole test for every database which is to be tested. JMeter instead works only with relational databases, but it provides also the possibility to test Java objects, which is very useful thinking at Java native interfaces to access the databases. Nevertheless this means that an object oriented point of view will be used

for object databases and at the same time a relational point of view will be used for relational databases, therefore the final result will have less meaning. With the purpose of benchmarking and comparing it's better to use a uniform point of view.

Since the benchmark application we want to develop must work with both relational and object database systems and since most of our database are object databases, we will use an object oriented point of view when benchmarking databses and therefore every database must implement a Java native interface for each test. Every database must implement a own interface, a driver, because there isn't yet any standard for object oriented querying.

Ease of use

Easy of use is intented as a variation in the tests'parameters, such as the initial database state and others already explained in chapeter 4, allowing little variations to the workload simulated, in order to improve the test when the needs change.

This ease of use should be granted without any modification in the application's lines of code. For this purpose Poleposition uses several properties files while with JMeter the user can modify the whole test, without any programming, and therefore he can also modify every parameters in the test without touching any code's lines.

Our idea consists in an xml file which contains all the tests' configuration. The xml file should be organized with a tree structure such as the JMeter's representation of the test plan. This xml must contain all these variable parameters. Furthermore we would like to configure also the final report of

the benchmark, specifying which graphs draw and which measures represents for each test.

6.1.2 Functional View

6.1.3 Information View

6.1.4 Concurrency View

6.1.5 Development View

6.2 Development

maven

svn

itext (per il pdf)

jfreechart

pattern utilizzati: indirection, adapter, factory method(driver del db), singleton, composite (per le operation), facade, proxy, monitor

6.3 Application Usage

6.3.1 Configure the XML

Bench-properties.xml

Simple Test

Real Time Prepaid System

6.3.2 Implementation of New Tests

6.3.3 Implementation of New Databases

Chapter 7

Results' Analysis

7.1 Why Pico4 is faster

Appendix A

The first appendix

questa e un appendice

Bibliography

[Bernt 05] Johnsen Bernt. *Database Benchmarks*. `http://weblogs.java.net/blog/bernt/archive/2005/10/database_benchm_1.html`, october 2005.

[Burleson 02] Donald Burleson. *Database benchmark wars: What you need to know*. `http://articles.techrepublic.com.com/5100-10878_11-5031718.html`, august 2002.

[Council 07] Transaction Processing Performance Council. *TPC Benchmark C*. `www.tpc.org/tpcc/spec/tpcc_current.pdf`, june 2007.

[DeWitt 97] Dave DeWitt. *Standard Benchmarks for Database Systems*. `http://www.tpc.org/information/sessions/sigmod/sigmod97.ppt`, 1997.

[Fowler 05] Martin Fowler. *InMemoryTestDatabase*. `http://martinfowler.com/bliki/InMemoryTestDatabase.html`, november 2005.

[Gorine 04] Andrei Gorine. *Building durability into data management for real-time systems*. `http://www.mcobject.com/downloads/bassep04p19.pdf`, september 2004.

[Graves 02] Steve Graves. *In-Memory Database Systems*. `http://www.linuxjournal.com/article/6133`, 2002.

[Gray 93] Jim Gray. *Database and Transaction Processing Performance Handbook*. `http://research.microsoft.com/~Gray/BenchmarkHandbook/chapter1.pdf`, 1993.

[Grehan 05] Rick Grehan. *An Open Source Database Benchmark*. `http://today.java.net/pub/a/today/2005/06/14/poleposition.html`, june 2005.

[Hobbs 03] Darren Hobbs. *Prevayler Pertubations*. `http://darrenhobbs.com/?p=225`, march 2003.

[Miller 03] Charles Miller. *Prevayling Stupidity*. `http://fishbowl.pastiche.org/2003/04/11/prevayling_stupidity/`, april 2003.

[Prevayler 08] Prevayler. *Prevayler Home Page*. `http://www.prevayler.org/wiki/`, march 2008.

[Wuestefeld 01] Klaus Wuestefeld. *Object Prevalence*. `http://www.advogato.org/article/398.html`, december 2001.