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

Dedicato a te

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

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

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 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 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

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 seem faster or slower: the test scenarios used to run a benchmark and to analyze database's performance. 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 major part of benchmarks does, particularly Poleposition [Grehan 05].

Every test can execute different read/write operations on the database and all these operations are inside a loop. The word *transaction*, used extensively in the following paragraphs, refers to an execution of the loop, and therefore all the operations inside it.

The key features of these kind of test are:

- A fixed number of transaction before the test stop: so tests are config-

ured 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 transactions per second.

- Different kind of object: 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 avoid concurrent test, 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 database's limits.

In order to create a test scenario we have to define the object/s involved by 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 execution 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 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 to 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 transactions 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, whose meaning has already been exaplined in paragraph 3.2. 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 transactions.

4.3 Load Test Case

Restrictions introduced by base tests are very evident although they can be useful in certain situation and for simple and fast approximate results. These restrictions are due to the impossibility to simulate real complex application scenario. Substantially the main restrictions are two:

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 scnerios most realistic as possible.

2. The second restriction result from 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 may lower the performance of other threads.

Base test case offer very important and useful results, but when the need 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, differently by base tests, are:

- Multi task: there will be not only a combination of read/write/update/delete in one transaction executed synchronizedly, but in more transactions running concurrently against the database.

- A bond on the throughput: for every transaction it is possible to specify the maximum throughput, if the database can reach it, in order to simulate real application usage.

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 involving complex transactions: check balance, accounts handling, and manage calls and services.

We start describing the domain objects used by this test, and the we pass to analyze the three different task 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 is having many private field but no hierarchies. In a real application there are millions of accounts instantiated in the database, one for every customer. This dimesion 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 subscription is 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 doesn't is deleted. A session contain the start time of the call represented, and the time of the last unit. Each

session references to an account, which started the call. But there are not as many sessions as accounts, not everyone will start a call in the same time, except New Year's Day!

Check Balance 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 read/listen his balance.

Each task, as already explained, corresponds to a transaction composed by 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 transaction.

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 Handling Task

The account handling task simulates the creation of a new subscription by a customer, and so the creation of the account object, and after 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 subscription.

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

To sum up, this task executes three write on the database for every transaction.

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

Manage Calls and Services 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. After this time 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: msisdn, webuser and session. The forth is a complex flat object: account. The number of objects already initialized in the database, before the test start, is on the order of millions.

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 minor task, 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. Then the stop condition will be a specific amount of time.

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, except the vendor's words, how strong is the durability solution implemented by the in-memory database.

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 specific benchmark.

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 a benchmark should have:

- 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 anythng in particular*: it's impossible to make a benchmark for everything, it's better if it has some limitations but works better in some specific tasks.
- 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 choosen 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 sytems, 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 inteded as clear results 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 the consequently 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 cutomers 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 requirements 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 test 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' behaviour. On the other hand only one measure is easier to understand but may hide the database behaviour 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 measures have been already introduced in paragraph 3.2 and they represent a subset of all possible measure 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 choiche, 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 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 may 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 represents 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 standalone 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 databse 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

5.1.8 Benefits

define the playing field accelerate progress for engineers set the perfomance agenda for managers

5.2 The Open Source Database Benchmark

Surfing the web and looking for an open source benchmark, this is the first benchmark it's possible to find...

5.3 Transaction Processing Performance Council

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*** [Council 07].

5.4 Apache JMeter

5.5 Poleposition

Poleposition fa schifo! [Grehan 05]!!!

5.6 Overview's Results

Tabella riassuntiva, con le caratteristiche di ogni benchmark che ci hanno convinto Nessun benchmark va bene. Si potrebbe pensare di prendere alcuni benchmark ed estenderli implementando i casi di nostro interesse. Oppure fare una nuova applicazione di benchmark...segue nel prossimo capitolo.

Chapter 6

The New In-Memory Database Benchmark Application

6.1 Functional View

6.2 Development View

6.3 Plug-In Architecture

Chapter 7

Results' Analysis

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.