Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea

In-Memory database benchmark

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

In-Memory Database

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

Competitor Landscape

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 analyze the most popular IMDBs, 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 The Analysis' Structure

- point of view - come strutturata l'analisi (vantaggi/svantaggi/project info/esempi) - vantaggi e svantaggi che in linea di massima condividono tutti gli IMDBs - le sorgenti sono sourceforge e olho

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 programmerfriendly 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-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 for further details 2.2.4 a pag 18.

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

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

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.

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

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 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 probably will be 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

2.3.2 Disadvantages

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

[3] HSQLDB stands for Hypersonic SQL DataBase.

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 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 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 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 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 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 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 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 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 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 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 info info

2.11 ExtremeDB

intorDB intorDB

2.12 Polyhedra

intorDB intorDB

2.13 TimesTen

intorDB in-

torDB intorDB intorDB intorDB

2.14 Csql

intorDB intorDB

2.15 SolidDB

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 in-

torDB intorDB intorDB intorDB

2.21 Pico4v2

intorDB intorDB

Part II

Database Performance Analysis

Chapter 3

Test Suite

3.1 Base Test Case

3.2 Real Time Prepaid System Test Case

3.2.1 Domain Object

3.2.2 Check Balance Test

3.2.3 Write New Account Test

3.2.4 Manage Call Test

3.3 Acid Test Case

Chapter 4

Database Benchmark Softwares Overview

4.1 Our Requisites

4.2 The Open Source Database Benchmark

4.3 Transaction Processing Performance Council

4.4 Apache JMeter

4.5 Poleposition

Chapter 5

The New In-Memory Database Benchmark Application

Chapter 6

Databases Comparison

Appendix A

The first appendix

questa e un appendice

Bibliography

[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.*

[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.*