POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

MorPhyne, using a non relational database system for big data management

Supervisor: Prof. Marco D. SANTAMBROGIO

Assistant Supervisor: Alessandro A. NACCI

Thesis of:

Domenico MATTEO

Matricola n. 702473

Anno Accademico 2012–2013

to my family, my sweetheart and my cats too

Acknowledgements

I have a lot of people to thank for...

People I have met and people I have lost.

People who loved me, supported me through this long journey
and stood by my side, even when I was being a prick.

Thank you NECST Lab guys, for the countless hours
we spent together working, coding, studying, playing and joking.

Thank you Santa. Without you none of this would have been possible.
Thanks for all the opportunities, the sacrifices and the efforts you have done
for me and for every single one of your friends.

Thank you beautiful girls (and guys of course) of HOC Lab.
You have been a precious boost to my working and everyday life.

Thank you Mum. Thank you Dad. Thank you sweet sisters. For all the sacrifices
you all have been through to allow me achieve my goals and my fulfilments.

And finally thank YOU, Leti, for all of the above.
Thank you for your patience, your moral support and your love.
You've been the pillar that steadied my steps in this long troubled path.

Thank You All.

Sommario

Al giorno d'oggi, con l'avvento del web 2.0 e delle applicazioni ad uso intensivo di dati in generale, si è reso necessario un approccio più flessibile rispetto al modello relazionale che ha imperato finora. Pertanto questo lavoro di tesi esplora l'utilizzo di database non relazionali focalizzandosi su uno specifico caso d'uso.

Più precisamente MPOWER, una applicazione per dispositivi mobili Android, sviluppata presso NECST Lab, Politecnico di Milano, sarà il caso d'uso in esame. La finalità di MPOWER è quella di raggiungere un guadagno in termini di durata della batteria osservando, registrando ed analizzando il comportamento del dispositivo nel suo normale utilizzo quotidiano, da un punto di vista energetico.
Dal suo lancio MPOWER ha collezionato più di 250 utenti con una quantità di dati di utilizzo superiori ai 250 milioni di record, che vengono poi analizzati per costruire un modello di potenza per uno specifico dispositivo.

L'architettura lato server che si occupa di gestire il collezionamento dei dati era una macchina fisica che però faticava molto a recuperare i dati utili alla generazione dei modelli stessi. Inoltre, una singola macchina fisica non garantiva nessun livello di affidabilità, recupero da guasti o segnalazione degli stessi.

Per affrontare questi problemi, era necessario una intrinseca modularità del sistema, che potesse facilmente adattarsi alle crescenti necessita dell'infrastruttura. La nuova architettura realizzata è composta interamente da macchine virtuali, tre allo stato attuale, ognuna delle quali assolve a compiti differenti. MORPHYNE è l'interfaccia tra l'applicazione MPOWER e le macchine virtuali usate come database. OPIUM e SPEED sono appunto le macchine virtuali dedicate rispettivamente al database relazionale ed a quello non relazionale (nello specifico, MySQL Server e MongoDB)

L'intera infrastruttura è mostrata graficamente in Figura 1.

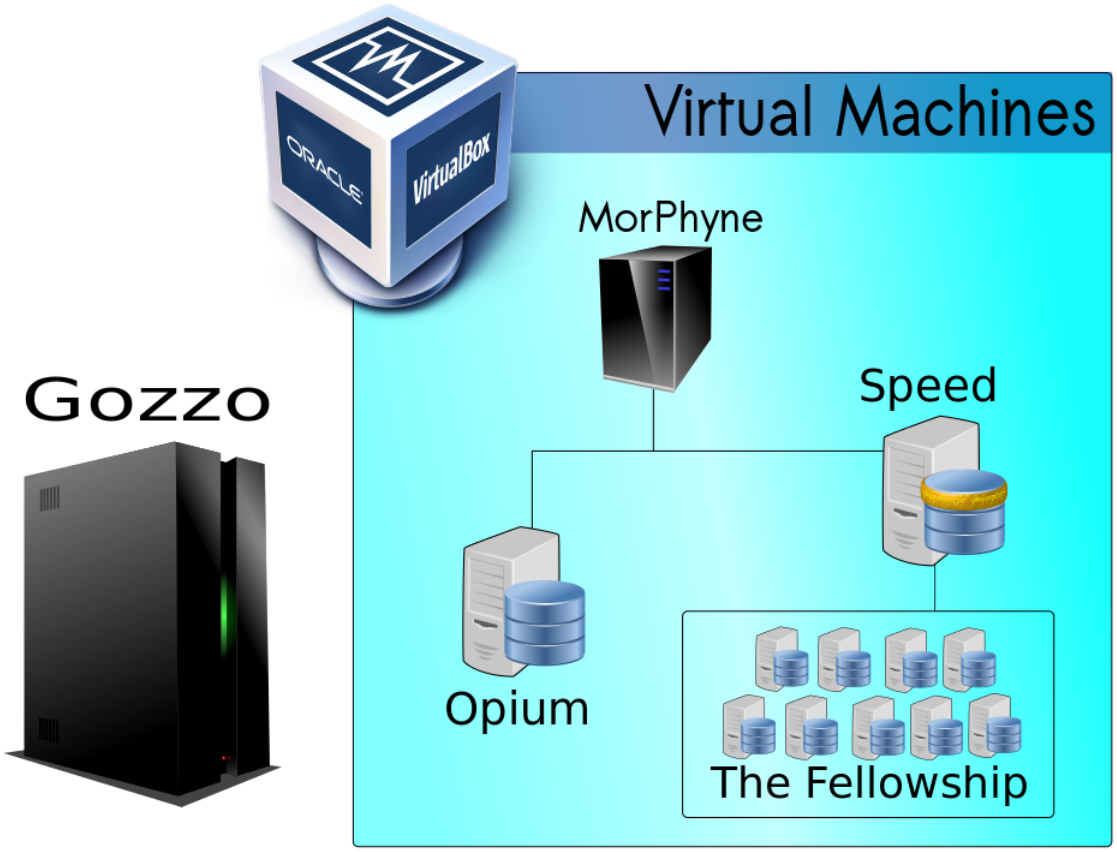

Figure 1: Nuova architettura basata su macchine virtuali

Un altro importante nodo affrontato in questo lavoro è stata la *migrazione* dei file di log generati da MPOWER, precedentemente creati come struttura CSV, ad un non ambiguo formato utilizzando la notazione JSON. Questo si è reso necessario per evitare alcuni problemi sorti con la vecchia infrastruttura, dove la rigida struttura del file CSV prevedeva una corrispondenza netta tra colonne del file e campi del database, creando confusione in caso di architetture mobili differenti.

Dopo una dettagliata analisi dello stato dell'arte ed una ampia esplorazione delle differenti tecnologie NoSQL presenti sul mercato, la scelta è caduta sul database documentale MongoDB, per le sue capacità di amministrare enormi datasets, scalare orizzontalmente e l'affidabilità ottenibile mediante i suoi meccanismi. Oltretutto, il formato predefinito con cui MongoDB registra i suoi documenti è BSON, ovvero Binary JSON. Considerato questo, i log file generati in formato JSON possono essere *trasformati* in BSON con estrema semplicità.

Oltre alle sopra menzionate introduzioni, altre importanti funzionalità sono state aggiunte al sistema, mantenendo una completa retrocompatibilità con le precedenti versioni di MPOWER. Sono stati introdotti un meccanismo di rilevamento delle anomalie nell'elaborazione dei files ed un meccanismo di rimozione dei file corrotti o danneggiati, per evitare le situazioni di stallo che venivano a crearsi nella precedente infrastruttura in casi simili.

Dopo la completa riprogettazione dell'infrastruttura lato server di MPOWER, sono stati svolti diversi test prestazionali sulle due differenti tecnologie.
I risultati ottenuti da MongoDB hanno confermato le assunzioni iniziali che questo tipo di database documentale avrebbe portato vantaggi sia in termini prestazionali, sia in termini di scalabilità, soprattutto considerando l'enorme quantità di dati che ci si aspetta di ricevere in un prossimo futuro.

Comparato alla sua controparte relazionale, MongoDB ha registrato dei buoni risultati, mostrando delle difficoltà iniziali e tirando fuori gli artigli nel momento in cui i meccanismi di caching entravano in gioco.
Prendendo ad esempio una complessa query di selezione che ha estratto 96053 oggetti, in una prima esecuzione della stessa MySQL ha ottenuto tempistiche fino a 2.83 volte migliori rispetto a MongoDB. Nelle query successive però, mantenendo invariato il dataset, MongoDB ha ottenuto tempi fino a 4.15 migliori di MySQL

In conclusione, questo lavoro ha sicuramente portato un grande beneficio verso il futuro sviluppo del progetto MPOWER, in termini di performance, affidabilità e scalabilità, portando nel lungo termine ad un sistema molto più stabile ed efficiente.

Summary

Nowadays, web 2.0 and data intensive application in general often requires a more flexible data model than the classic relational model. Therefore this thesis explores the use of a non relation database instead of a relational one in an use case oriented comparison.

In particular MPOWER, and Android application developed at NECST Lab, Politecnico di Milano, will be our use case. MPOWER's goal is to achieve a better and longer battery lifetime by observing, logging and mining the behavior of the device throughout his normal day life, from an energy focused point of view.
Since its launch MPOWER collected more than 250 users with an approximate amount of activity logs of 250 million of records, that are then analyzed to build the power model tailored for a specific device.

The server architecture that backed the whole data gathering system was a physical workstation that struggled to keep up with the increasing workload and often failed to retrieve the requested data for the building of the power model. Moreover, that single physical machine doesn't provided any kind of reliability, fault tolerance or failover configurations.

To tackle these problems, there was a strong need for modularity and flexibility on the overall system, that could bend and stretch as desired.
The new system architecture is entirely composed by virtual machines, three of them at the present time, which absolve different tasks each.
MORPHYNE is the gateway interface between MPOWER and the database VMs. OPIUM and SPEED are the virtual machines deputed respectively to run the relational and the non relational database (MySQL Server and MongoDB, to be more specific).
The system infrastructure is pictured in Figure 2.

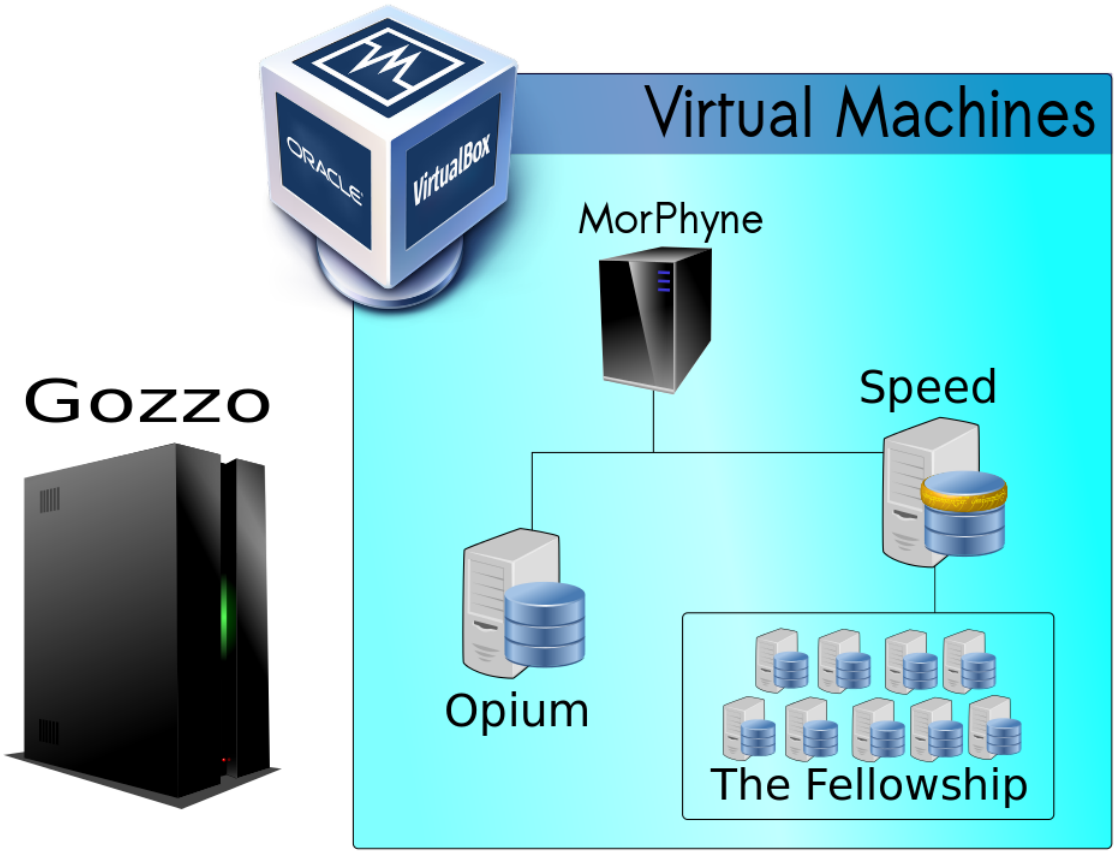

Figure 2: New VM based server infrastructure

Another important issue covered by this work is the *migration* of the log file generated by MPOWER, previously created as CSV, to a non ambiguous JSON notation. This was done to avoid some problems and restriction that a rigid CSV structure brings along, like the strict matching of columns of the CSV against fields of the database.

After a detailed state of the art analysis and a wide exploration of the different NoSQL technologies, the choice fell on the document store MongoDB, for his capabilities of managing huge datasets, horizontal scalability and reliability. Moreover the document type used by MongoDB is the BSON, that stands for Binary JSON. Given that the JSON log files generated by the new version of MPOWER can be very easily mapped into its binary representation.

Besides the aforementioned introduction, more feature was added to the new system infrastructure while maintaining a full backward compatibility with the holder versione of MPOWER. An anomaly detection on the processing of log files and a mechanism of broken file removal was introduced to avoid the stall caused by damaged or incomplete file log that could be encountered in the previous architecture.

After the complete redesign of the MPOWER server architecture, benchmarks on the two different technologies were performed.
The results obtained from the MongoDB infrastructure confirmed the initial assumption that this document store leads to performance improvement, both in terms of response time on complex queries that in terms of scalability, especially considering that a huge number of data is expected in the near future.

Compared to his relational counterpart, MongoDB performed well, exhibiting slightly worst at first but outperforming MySQL when its caching mechanisms kicked in. For example on a selection query that returned 96053 objects, on the first run MySQL got a timing 2.83 times better than MongoDB. On every following query on the same dataset instead, MongoDB retrieved the data 4.15 times faster than MySQL.

In the end the whole work has been a great step forward towards the future of the MPOWER project, both in terms of performance that in terms of reliability and scalability, leading to a more stable and performing system in the long run.

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

During the last decades relational databases have been widely used for nearly all the applications that needed a storage, regardless of the specific needs. However, nowadays, the relation-based data model is becoming more and more unsuitable for several usages, especially for all those web oriented applications that does not strictly match the relational model.

When E. F. Codd firstly introduced the relational model, in 1970 while working at IBM Research [1], there was a diffuse acceptance from the academic community, while the practitioners remained skeptical regarding the performance to the existing hierarchical and network database [2].
In the early 80s, after the introduction of System R [3], relational DBMS grew very competitive and became the largest player on the market.

M. Indrawan-Santiago [4] suggests that in those years, the general focus of data processing was on:

- *Online Transaction Processing (OLTP)*, mainly focused on simple operations, like plain writing on the database [5]
- Other applications such as Data Warehousing, that required complex operations and read-focus processing

During this time a number of revolution threatened the undisputed dominance of the relational model such as the Object Oriented Movement that gained momentum in the 90s, with the sustenance of the software engineering community.

However, none of these databases really became the threat they claimed to be, nor they made a small dent in the relational dominion.

The reason of this failures could be mainly found in these key points:

- their lack of theoretical foundations and theoretical studies [4] [6]
- the limited performance gain in respect of the relational databases [6]
- the sacrifice of the ACID (Atomicity, Consistency, Isolation, Durability) [7] properties and the possible consequences, in favor of performance and availability

The latest revolution appears to be the NoSQL (Not Only SQL) Movement, which claim themselves to be non-relational, schema free, horizontally scalable, distributed and mainly open source. Their origin can be linked with the introduction by Google of the MapReduce [8] framework first, and their NoSQL database, BigTable [9] later, in 2004 and 2006 respectively.

This thesis work will experimentally make a comparison between a relational approach and a non relational one, applied to real data gathered by a real application from real users... *Things just got real* [10]

1.1 SQL versus NoSQL

A number of non relational databases appeared in the last few years, such as MongoDB [11], Neo4J [12], Riak [13], and much more that sits under the NoSQL umbrella.

There is great ferment around the roles this databases may serve in the present and future years, and even if they will ever live out to their expectations [14]. Meanwhile, the SQL supporters claims that the NoSQL family has a major drawback that can't be ignored, the lack of strict treatment of data integrity [15]

Some factors that have influenced the growth of the NoSQL family are [16]:

- **Big Data**: The Web 2.0 has an increasing need of managing 'big data' for commercial, academic and research applications.
  For some applications that query single rows on many columns, could be practical to group these columns that would otherwise need several *JOIN* to be accessible
- **Mixed-Structured Data**: Facing the challenge to support structured and unstructured data via a less strict schema or schema free models
- **Computing Shift Towards Parallelism**: With the advent of the wide availability of parallel and distributed computation, the focus has been shifted from clock speed per unit to the usage of many processing units, thus needing an adaptive and horizontally scalable model

1.2 Big Data: how much is big?

'Big Data' has become a hot buzzword in the last years, but unfortunately, a poorly defined one.

Wikipedia defines it as "a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications" [17] and even this one is pretty loose.

Since there is not a common definition of Big Data, we could extrapolate what most of them share between each other and build a *custom* definition around it: **variety, velocity and volume**

To me, Big Data, is in the eye of the beholder. As long as the conventional approach to the data set becomes unsatisfying, for the variegation of data, the velocity of their growth or the total volume of it, this could be a big data issue, and should be evaluated as this.

Let us introduce the application I've used to pragmatically make the comparison between our MySQL and MongoDB implementation of the back-end data management

1.3 MPower: a case of study

In nowadays life, mobile phones are becoming cheaper, smaller and with an increasing number of functionalities. Moreover they are always available in our everyday life and they are becoming more popular than laptops. For these reasons, mobile phones are one of the most important interaction points between people and their surrounding environment.

Unfortunately, the resources in such devices are limited, e.g., computational power is reduced and battery life is limited. Nonetheless, since they are continuously used by the users and since they are moving into a fluctuating environment, both internal and external conditions are rapidly changing and this aspect may influence the behavior of the entire system, e.g. switching between network types may cause an unpredictable power consumption or the network instability.

As a consequence, when dealing with power consumption it is impossible to disregard the surrounding environment conditions and the internal state of the device (the context). In fact, with this knowledge of the context of the device, it is possible to optimize the behavior and the functionalities of the mobile phone saving as much power as possible without sacrificing the performance.

To adress this issue, in 2012 we presented **MPOWER** [18], an adaptive power management system for Android-based devices, which ultimately aims to optimize the battery life of a mobile device by analizing his routine behaviour.

The first version of MPOWER was developed to collect all the relevant data that the operating system expose to the developer, in order to later analyze them and build a power model based on them.
The gathering is done every 10 seconds, and this timing has experimentally proved himself to be sufficiently accurate while having a negligible impact on the power consumption.
According to the power consumption statistics introduced in Android 2.3, this consumption remain well under 2% of the global discharge rate.

The idea behind the first version of MPOWER was that, knowing the regular behaviour of a user, we could profile his habits and help him improve the duration of his battery by automatically turning on and off the

Table 1.1: Data gathered from the device

Screen	**Battery**	**CPU(s)**	**Mobile**
is_on	on_charge	max_frequency	state
brightness_mode	temperature	min_frequency	activity
brightness_value	voltage	current_freq	net_type
width	percentage	max_scaling_freq	signal_strenght
height	technology	min_scaling_freq	tx_bytes
refresh_rate	health	governor	rx_bytes
orientation		usage	call_state
		cpu_id	airplane_mode

Wifi	**Audio**	**Bluetooth**	**GPS**
is_on	music_active	is_on	is_on
is_connected	speaker_on	state	status
signal_strenght	music_volume		
link_speed	ring_volume		

components he needed, only when he needed them.

During the course of the year however, the project steered from his starting idea. We now aim at something more general and less obtrusive, profiling the device instead of the user, giving the generated model the power of reuse, without depriving it of his customization.

MPOWER is still an on going work, which evolution is structured into several different stages. As we have briefly mentioned, the current stage of the project focuses on building an accurate mathematical model of the power consumption by analyzing and mining all the data gathered by the application.

1.3.1 Android as a sensing platform

Modern devices have the availability of several hardware sensors, that can be used to sense both the internal and external device conditions.
The Android SDK includes a set of APIs that can be used to access the available sensors and observe the device context.

Information about the battery status is often coarse grained, since most devices do not include a physical measurer about electrical current or more precise hardware sensors. The only information exposed by the system to the developers are the battery percentage, voltage, temperature and an indicator of the battery health.

However, Android has a great variety of sensors that can be used to sense the external environment [19].

Through the SensorManager it is possible to access all the hardware sensors available on the device. Those sensors may include accelerometers, magnetometers, pressure and temperature sensors. Each sensor is identified through a type, a sampling rate and an accuracy. Even the touch screen itself is recognized as a sensor.

The aforementioned sensors are just some of the available ones, but it is clear that from an energetic point of view these sensors have to be used consciously, since most or all of them require an energy consumption. The large amount of information that can be directly retrieved or inferred in an Android device, makes this OS the best choice to work on.

On the other hand, Android does not provide developers instruments to understand how much an application, a component or the whole system is consuming, nor it is possible to know this consumption *a priori*.

Starting from version 2.3, Android has added some statistics for the final user, i.e., a battery usage report that can be accessed through the system

Figure 1.1: Graph of the installation since April 2012

settings menu.

In this screen are listed all the applications and services that have consumed at least the 2% of the battery life, since last full battery recharge.
A graph is is also displayed, showing the battery discharging over time and for each listed application some useful data are reported. However, these data cannot be accessed programmatically by non-built-in- system applications, strongly limiting their usefulness to developers.

1.4 MPOWER server architecture

Since the public release through the Google Play Store, in April 2012, the application collected a huge quantity of data regarding the users usage behaviour, as mentioned in Table 1.1. More than 270 users spread all over the world contributed to this collection, kindly reporting to our *old* server their daily habits data, as show in Figure 1.1.

This data got then analyzed and stored into our SQL database, which is constantly growing, and is now rapidly aiming for the well respectful quota of **250 milions of records**, or **a quarter billion of records**, if you prefer.

1.4.1 Current server architecture

MORPHONE, our current server, that receives all the data, analyze and store them, is a phisical machine, equipped with Debian 6.0 powered by Linux Kernel 2.6.32-5-amd64 .
On the hardware side, it's configuration is:

- Processing Unit: Intel® Core™ i5 650 @3.2 GHz
- Memory: 4 GB DDR2 @667 MHz
- HDD: 500 GB SATA II @7200 RPM

On the software side, the architecture is composed by:

- an Apache Web Server which is the gateway to the external world. It is used to display the website and browse through the personal user data, available on the website throughout Google authentication
- a web application, fully written in PHP, which receives the data from the application and stores them on the local file system, in order to be processed
- a back-end application, also written in PHP, which crunches the data, unzips and decrypts them, and finally store them on the database
- the database, which is a relational MySQL Server instance

1.4.2 Known architectural problems

The presented system architecture, began to show some unwanted behaviour as soon as the real data mining began, in April 2013.
Aimed at the creation of the mathematical models of power consumption, the mining is a data intensive application, and the database was not as responsive as we hoped it will be.

In fact, in order to build an accurate mathematical model of the power consumption given the model specification itself, at the present being, we need all the data ever recorded of the user. The strict dependence of the query complexity from the user seniority is trivial to see.

The selection queries used in the mining process to retrieve the user specific data, proved to be very computational expensive, in the order of hours to days of execution per each. This was probably due by the simultaneously presence of two bad situations:

1. The vast amount of *JOIN* used to retrieve the data from the different tables, filtered by the specific user
2. The sparse recording of data on the database.

Regarding this second aspect, the data on the database are saved in the order of which the files stored on the file system are processed, thus being inserted in a pseudo random order.
Given that, we may assume that in order to retrieve the whole lot of data regarding a single user, the DBMS must search extensively throughout its storage to provide the requested data, and this may cause the big delay we are experiencing.

An experimental proof of this could be found in the export and reimport of the whole database on an even less powerful machine;
after that, a selection query has been run on the test machine, giving results in the order of minutes, where instead on the MORPHONE server, the same query, took more than a day to accomplish.

Needless to say, these time constraints were becoming more and more unacceptable for the generation of the mathematical model and even for its

refinement, since this procedure should be done periodically on hundreds of users.

While aiming at a constantly growing number of users, this problem will only grow bigger and bigger, so we decided to entirely redesign the whole server architecture to actively meet our needs.

A fundamental step to this renewal process is the comparison and the evaluation of a new kind of database that could better meet our needs, MongoDB, one of the most famous non relational document oriented databases.

The main goal of this thesis is to compare the different approaches, and eventually demonstrate that we could have a performance boost by using the right type of database technology.

Chapter 2

State Of the Art Technology

Since this work is focused on the comparison of different database technologies and approaches on a particular use case, this section will be divided into two macro areas:

1. an accurate study and evaluation of the different power aware approaches available on mobile devices, with particular regards of the system level implementations
2. an overview of the different vendor and technologies available in the NoSQL panorama and the technical differences between them

2.1 Power modeling and management

In literature, several attempts to define power/energy models for mobile devices are described, drawing their approach from different fields of computer science.

A large number of works aim at creating power models of the entire device. Data to build system-wide power models can be gathered in two ways: either by attaching external sensors to the device (offline method), or by relying on the device's APIs (online method).

Using the first methodology, an accurate inspection of the mobile device is performed, using predetermined and controlled conditions. Unfortunately, nowadays a great variety of mobile devices exists and it is well known that their behaviour is affected by the OS release version, thus it is almost unfeasible to generate an offline power consumption model for each combination of mobile device and OS version [20].
Moreover, this analysis does not evolve with the running life of the device, while a run-time generated model is able to adapt itself to new software updates or even new devices. On the other hand, an adaptive model generation relying only on software APIs to gather information on the actual battery state cannot be as precise as the one developed using external measurement system.

Recently, some smartphones have been given a wider set of hardware sensors, thus allowing an intermediate approach to be used, with internal sensors providing precise data about the device power consumption. This last approach provides intermediate precision result between the external measurements and the API approach, but it lacks of flexibility, since can be applied only to those devices having internal sensors.

2.1.1 Using external measurements

The models shown in this section make use of external tools to gather the data from the device. This kind of approach provides very precise data, but it is difficult to perform, given its invasiveness on the device.

The goal in [21] is to decompose the energy consumption into independent components. The power consumption is measured by inserting a resistor in series between the battery and its connector on the phone and a

sampling board is used to measure the battery voltage.
Measurements do not require user interaction because are centrally managed by a Power Server, which is responsible for sending test scripts on the device and retrieve the resulting traces.
The tests are specifically designed to stress each component of the device, in order to provide the model for battery consumption of the specific component. The analysis was computed by sending data over a wireless network, using another Android device.
Collected traces allowed then to analyse the energy consumption involved in many different communication phases, e.g., during data sending phase.

In [22], an influential study by University of Michigan, Anand et al. propose a 2-way Operating System-level power management. This system consists in a modified version of Linux, running on a handheld iPAQ. I/O peripherics are fully described using a state enumeration, called power modes, each one associated to a power consumption values expressed in mW. In this system, applications are allowed to query the power mode of I/O peripherics and can even disclose hints about expected device discharge rate in a different power-mode.

To measure the power states, the authors removed the battery from the iPAQ device and sampled current drawn at 50Hz through the external power supply.
The system provides also a decision phase for power saving, relying on heuristics, e.g. using the applications given hints. Experiments are performed focusing on a web browser application and an email reader application.

Stanford University and MIT CSAIL develop in [23] Cinder, an OS based on the HiStar kernel. Cinder provides low-level abstraction for energy man-

agement and it is meant specifically for mobile, energy constrained devices. Measurements are taken through an Agilent Technologies E3644A (i.e. a DC power supply with a current sensor) on an HTC Dream mobile phone.

This research work does not directly focus on the power modeling, however it is interesting that they clearly stated that Cinder needs a solid model in order to work properly.
As stated in [23], the modeling approach suggested was to "build a model from offline-measurements of device power states in controlled setting".

In [24] Carroll and Heiser made a very deep analysis of power consumption on smartphones. Their goal was to understand where and how energy was used, in order to provide basis for understanding and managing mobile-device energy consumption.
They profiled energy consumption taking physical power measurements of supply voltage and current at the component level on a piece of real hardware. They inserted sense resistors on the power supply rails of the relevant components to measure the flowing current.

They also used a National Instruments PCI-6229 DAQ, connected to the sense resistors via twisted-pair wiring, to measure voltages. Using this kind of measurement the authors were able to directly measure the power consumed by the main components of the mobile device.
Furthemore, they measured the total power consumption by inserting a sense resistor between the power supply and the device.
In order to construct their model they ran two different benchmarks: at first, a series of micro-benchmarks designed to independently characterize single components, then a series of macro-benchmarks based on real scenarios.
From those studies emerged that the majority of power consumption can be attributed to the GSM module and the display (including the LCD panel

and touchscreen), the graphics accelerator/driver, and the screen backlight.

In almost all benchmarks, the brightness of the backlight is the most critical factor in determining power consumption. They also showed the impact of the screen image on power consumption. The GSM module consumes a great deal of both static and dynamic power (both maintaining a connection with the network and during a phone call), while the RAM, audio and flash subsystems consistently showed the lowest power consumption.

Using the data collected, Carroll and Heiser built an energy model based on usage patterns, to understand where the majority of the energy was wasted.
In order to do that they defined a set of patterns (suspend: baseline case of a device which is on standby; casual: user who uses the phone for a small amount of voice calls and text messages each day; regular: a commuter with extended time of listening to music or podcasts, combined with more lengthy or frequent phone calls, messaging and a bit of emailing; business: a user that makes talking and email use together with some web browsing) of the smartphone usage and described the daily energy use and battery life under each one of them.
This analysis clearly showed how different kind of usage have very different impact on the power consumption.

Shye et al. described in [25] both a power-modeling approach for Android-based mobile devices and a novel energy-saving policy to manage the screen brightness. Measurements were performed with a Fluke i30 AC/DC current clamp, while the operating voltage is retrieved through Android APIs. The modeling phase consists of a 2 states FSA: stand-by mode and active mode. If in the former state the consumption is accounted as fixed, in this

latter one the power consumption is computed through a linear regression model, given by the R-Tool fed with data collected by stressing an isolated hardware component.

This models can predict the power behavior of each scenario with a median 6.6% relative error. Moreover, the authors used data coming from several users to identify the most consuming component (proved to be the screen). By making the system automatically adjust the screen brightness, they managed to save up to 10% total energy, with minimal impact on user satisfaction.

2.1.2 Using only system APIs

A different approach consists in relying only on the OS APIs to collect the necessary data to build the power models.
OS APIs have some limitations in terms of precision, but they are present on every device, thus allowing these power models to be more adaptable.

In [26] Xiao et al. present a methodology for building a system-level power model, without requiring laboratory measurements. This model relies only on the data provided by the OS, thus it can be exported to any device. They develop a linear regression model with non negative coefficients, describing the aggregate power consumption of the processors, the wireless network interface and the display.

In order to perform the linear regression, they analysed data on the activity levels of each hardware component, i.e. the hardware performance counters (HPCs) for processor, the downlink and uplink data rates for the wireless interfaces, and the brightness level for the display. They discovered a linear regression model is sufficient to model the relation between the variables they choose and the power consumption.

To construct the model they used five types of different workloads: idle

with different brightness levels, audio/video players, audio/video recorders, file download/upload at different network data rates, and data streaming. One of the main features of this work is the possibility to adapt the model to new hardware. In fact, when a new hardware component is installed, it is possible to add regression variables, describing the activity levels of the new hardware component, define new test cases to stress the new components and, finally, fit the new data sets to a regression model.

In the case we are provided of an analytical power model of the new hardware component, it is possible to merge it with the existing system-level model. The power estimation, based on their model, exhibits a median error of 2.62% in real mobile internet services.
Moreover, they provide a model that is independent from usage scenarios and that can be used for runtime estimation with reasonable accuracy.

In [27], a context-aware system to accurately predict the battery lifetime is proposed. The energy consumption of each system component is considered dependent on its operational state and the amount of time the component remains in that state.
As a consequence, the system power consumption is modeled as the sum of the system components.

Data about the discharge rate were collected in several system contexts, where a single system context is a combination of CPU utilization, LCD brightness, WiFi state, IO idle rate and volume of transferred data.
Multiple linear regression techniques were used to build a model able to describe the battery discharge rate, based on the system component states.

The system was tested using an HTC G1 smart phone, running the Android OS. Data were collected using 40 different test scenarios: 16 of those were used to build the model, the remaining to evaluate the generated model. The generated model predicted the battery remaining lifetime with

a relative error of 10%.

In [28], Pathak et al. propose a new power modeling approach based on tracing system calls of the applications, which captures both utilization-based and non-utilization-based power behavior, and it is based on a fine-grained energy estimation.
Their scheme consists of two major components. The first one is a Finite State Machine (FSM) to model the power states and state transitions. Some of the states have constant power consumption, to describe non-utilization power consumption, while other states leverage a linear regression (LR) model (the second major component of the model) to capture the power consumption due to system calls generating workload. Moreover, a testing application is used to systematically uncover the FSM transition rules. Tests were performed on a HTC Touch Pro and a HTC TyTn 2, powered by Windows Mobile 6, and an HTC Magic running Android.

This new approach improves the accuracy of fine-grained energy estimation compared to utilization-based model. Indeed, their model have a 80th percentile error of less than 10% estimating the power consumption over time of a generic application execution that lasted 50ms, while the utilization-based model have an error that varies between 16% and 52%.

The whole application error, with 1 sec granularity, varies between 0.2% and 3.6%, compared to the error 3.5-20.3% given by utilization-based model.

2.1.3 Using custom and internal measurements

Several works have been specifically designed to exploit interfaces already available on specific devices or OS, e.g. the Advanced Configuration and Power Interface (ACPI) or the Nokia energy profiler. The use of these interfaces allows to have precise information about the battery status (e.g., voltage, current and temperature) or even the current power consumption

of the device at a specific time.
These choices lead to the generation of very precise power consumption model. However these models are not portable on devices without specific interfaces: not every Android-powered devices can take advantage of these power models, because they may be unable to provide the required data with the required precision.
An interesting system called Sesame is proposed in [29]. It is a self-modeling approach to build high-rate mobile system models without any need for external measurement systems.
At first, Sesame collects data traces, i.e. system statistics and data provided by the ACPI, building an initial set of predictors; this predictors may even be user defined and strictly depend on the platform in use. Predictors may include, among the others, CPU utilization, cache misses, wifi traffic or LCD backlight level.

Then, the model is built using two iterative techniques, model molding and predictor transformation. The former generates a model using linear regression, while the latter improves the accuracy of a molded model, transforming the original predictors and finding better linear combinations of the original predictors.
The model is composed by a set of sub-models, each corresponding to a different system configuration. Sesame was implemented both for laptops and smartphones, using the Linux kernel. This study highlighted that data collection frequency influences the system accuracy, dropping drastically when the data collection frequency matches the frequency of the register used to store the monitored value.

Accuracy was reported as 86% and 82% at 1 Hz and 100 Hz on the smartphone and 95% and 88% at 1 Hz and 100 Hz on the laptop. In the model generation, Sesame provides energy estimations with error of at most 12% at 1 Hz and 18% at 100 Hz for laptops and 12% at 1 Hz and

22% at 100 Hz for N900.

Another work exploiting the ACPI interface is proposed in [30] the proposed methodology involves a mixed-approach, combining the recent observed device power consumption history with offline benchmark measurements to predict the battery lifetime. The offline measurements are used to offer a baseline to the prediction, while more recent and online data are used to react to varying and unpredictable events.
The offline measurements were computed running on a quiescent system constant-power workloads, i.e. repeated executions of a single program to completely discharge a fully charged battery.

Starting from the offline history curve, several linear regression methods are used predict the battery lifetime taking into consideration also the current system conditions. The online data are retrieved using Smart Battery [31] and the ACPI interface. Their method achieves a 5% maximum error for predicting the power consumption of constant workloads, with the exception of media, whose fluctuations during execution cause the error rate to rise un- til 10%.

In [32], targeting Nokia devices with Symbian OS, they exploits Nokia Energy Profiler tool, which provides precise power measurements with respect to OS APIs.
Here an unsupervised method to model a device energy consumption, based on genetic algorithms, is proposed. This solution uses on-line measurements through system APIs, in order to generate on-demand models, for instance, a new model may be needed when the system configuration changes. Data retrieved through the available APIs include the battery voltage, current and other statistics.

In order to build a model, their method needs a controlled training

dataset, collected exercising independently all the important phone features;
the genetic algorithm is then used to choose the relevant features. The exercised features include WiFi, GSM, CPU and GPS. Experiments presented considered training data mostly collected in predefined conditions, but the system was also tested using a real-location use case scenario. The model can predict power consumption with a 95th percentile error of 0.313 watt. This error increases when the application is released *into the wild* with more frequent and overlapping feature usage.

In [33] an energy estimation system has been presented, able to compute at runtime the power consumption of tasks executed in RAM and the data transfer performed over the WiFi network.
In this work, a battery monitor unit is assumed to be already integrated into the target devices. At first, an initial offline profiler is used to generate a starting model state, using the profiled data. Then, a power estimator component produces the energy estimation on a defined interval, taking into consideration a computation model and a communication model.

Those models are characterized by a set of parameters, computed at runtime using a recursive least square linear regression with exponential decay method. This phase uses the feedbacks from a runtime profiler, which gather information about the battery voltage, temperature and current from the battery monitor.

In [34] Zhang et al. aimed for an online generator of power models.
They started their study analyzing each component separately, using external tools. They concluded the power consumption of major component affects the system independently, so power consumption of the entire system when 3G and WiFi are active is the sum of 3G and WiFi contributions.

After that they evaluated the intra and inter class variance and noticed that different devices have power model pretty far from each other. This justifies the need to develop a power model for each device and, in order to do so, their method was modified to be able to create a power model based on data coming from sensors within each device.

The result indicates that the power model built with PowerBooter is accurate within 4.1% of measured values for 10-second intervals.
However, this model presents two main limitations: the need of a specific discharge curve for every specific device and the need for a mobile phone that allows superuser access in the OS.

2.1.4 Existing Techniques Open Issues

Generally speaking, offline data gathering gives more precise results, but is more difficult to implement and consequently less prone to adaptation. As a consequence, it is seldom used for giving to the users an estimation of lifetime battery.
Moreover, it lacks of generality since the gathered data are specific for a given device. On the other hand, it is possible to perform an online data gathering and to build power model at runtime. These models, despite their degradation in precision, have a great adaptability and are most suited to cope with the fast growing market of smart devices. Among these, we can separate models taking into account the user interaction [35] [36] from those which does not [27] [37].

User-based models provide a better estimation of lifetime battery, since they model the behavior of every single user. However, modeling the user alongside the device makes a number of assumptions; first of all, each device have to be used only by one user.
This fact is commonly true for smartphones, but not for tablets, commonly

used by a whole family or by a working team. Also, more important, in modeling the user we implicitly make the assumption the user will use his phone always in the same way, which does not holds, for instance if he/she goes into vacation or he/she changes his job.

Some of the methodologies analyzed try to cope with this fact making the model forget data over time, even if the user behavior is not changing. On the other hand, state-of-the-art component-based models are built using benchmarks applications: since these applications are built to stress one component at a time, the models built on top of them can rarely predict if any interaction arise among different components from a standard usage of the phone.

2.2 NoSQL Overview

As has already been mentioned in Chapter 1, NoSQL databases mostly differ from their relational counterparts for their data model, for the more relaxed application of the ACID properties and for their less strict or even non existent schema.

The CAP Theorem, the BASE theorem and the Eventual Consistency theorem are the foundations on which all of them resides [38].

CAP Theorem

Eric Brewer's CAP Theorem, that is *Consistency, Availability and tolerance of network Partition*, was first proposed in 2000 [39].
It states that a distributed system can only meet two of the aforementioned characteristics at the same time, as shown in Figure 2.1.

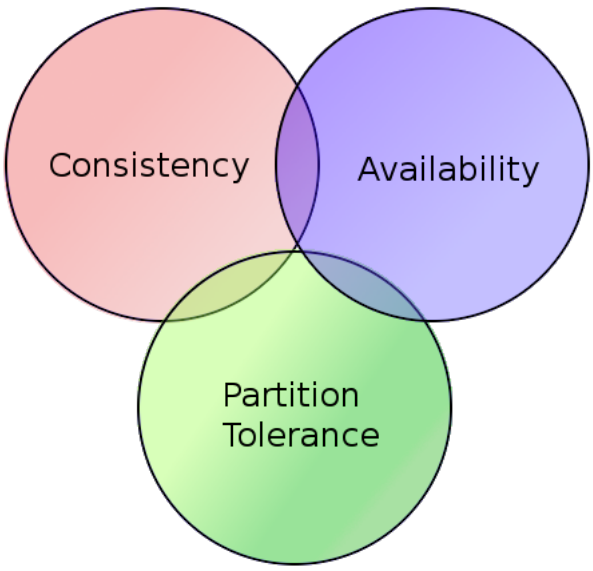

Figure 2.1: The CAP Theorem

When designing Web 2.0 systems, the only orientations available are CA and AP, because for this specific case, Availability and Partitioning tolerance are far more important than Consistency.
As far as these systems are concerned, Eventual Consistency is more than enough.

BASE Theorem

When CAP Theorem collide with practice, the BASE model shows up. BASE model is very different from the more strict ACID model, and it stands for *Basically Available, Soft-state and Eventually Consistent*.

- **Basically Available** means that a partition failure could be supported
- **Soft-state** means that at a given time the state of the system could be non-synchronous
- **Eventually Consistent** means that at last, the data should be consistent

To summarize it, BASE model focus on the high availability, leaving to some application level mechanism the burden of resolving any problem that could have been created by optimistic choices that in the end turns out to have violated the consistency.

Eventual Consistency

ACID compliance may not always be a priority, e.g. a search engine could easily display slightly different search result to two different users without causing any hassle.
In these kind of applications, the focus is all on speed and performance, while the consistency get trampled in the way.

However in a banking application, for instance, two users using the same account, necessarily need to to same balance on it. This is a classic case when the ACID properties cannot be relaxed, and when the Eventual Consistency could wreck havoc.

2.2.1 Data Model in NoSQL

In the past SQL approaches were used even when the relational model doesn't matched the targeted data model.

Moreover, even when a problem could be easily covered by relational blanket, more often than not, the huge feature offered by the SQL databases was used only in very limited part.
A meaningful example of this waste of energy are the logging applications, like the one we are studying, MPOWER.

Given these constraints gravitating around the relational model, several companies developed their own database technology, matching their very own needs, which later fell back in the NoSQL bucket, like for instance FlockDB for Twitter, BigTable for Google or Dynamo for Amazon.
Since every store was developed with unique characteristics in mind, it's difficult to select the right technology for the right application, since there is no *panacea*.

In order to evaluate and compare the different NoSQL technologies on the market, we first need to categorize them based on their affinity and common features. Then we will proceed to evaluate their querying possibilities and how they impact on the overall system.

We will now dig a little more into the main NoSQL families, that are key value, document, column and graph databases, to discover their characteristics and their differences.

Key Value Stores

Key Value stores are basically hash maps, or dictionaries, that simply match a key to it's value.
Keys are the only way to retrieve information on them, since the values are completely opaque to the database. There are no relations among the values, that are independently stored with their unique key.

For this very reason, key value stores are completely schema free, since no sort of complex structure is ever implemented on them, and must be totally managed on the application level.

Key value stores are often used to speed up operations, like the loading of an already visited user specific webpage. Another use of this systems is the caching in support of complex SQL queries, since most key value stores hold their dataset in memory.

The most notable key values stores are Dynamo [40], Project Voldemort [41], Redis [42] and Couchbase (former Membase) [43]

Document Stores

Document Stores structure key values pairs in JSON like documents [44], and within them, keys are unique. Every document has a unique special ID, that identifies it in a collections of documents, thus identifying it explicitly.
Given their nature, JSON like documents does not have a defined schema,

giving this kind of databases a great flexibility as new attributes can be easily added at runtime.

Unlike key value stores, values are transparent to the system, and can be directly queried. Storing JSON like documents have the additional advantage of conveniently supporting data types.

Most popular use cases are real time analytics, logging and the storage layer of small blogs and web 2.0 applications.

The more famous representative of this family are MongoDB [11], CouchDB [45] and Riak [13].

Column Family Stores

Column oriented databases are strongly inspired by Google BigTable [9] which is a "distributed storage system for managing structured data that is designed to scale to a very large size".
The data model, as described by the authors, is "sparse, distributed, persistent multidimensional sorted map" [46].

The only natively supported type of data is the string, leaving to the applications all the weight of the type management, when needed.
To support versioning, partitioning and achieve better performance, multiple versions of a value are stored in a chronological order.
Columns can be grouped in families, mainly used for the partitioning over different machines or nodes of a cluster.

There exists some open source implementations of BigTable, that are HBase [47] and HyperTable [48]. On the other hand Cassandra [49] slightly dif-

fer from the previous ones for the implementation of another dimension, called super column, used to contain columns and column families to handle more complex structures.

Graph Databases

Graph databases plays in the domain of the heavily linked data, with many relationship, since cost intensive operations like recursive *JOIN* can be efficiently replaced by traversals.

Neo4j [12] and GraphDB [50] are based on directed and multi relational graphs, while Sesame [51] and BigData [52] are focused on querying and analysing subject-predicate-object statements.
FlockDB [53] is used by Twitter to efficiently manage their follower-following one directional relations, and it is optimized for very large adjacency lists.

2.2.2 Querying Possibilities

The great variety of stores available on the market that was presented in Chapter 2.2.1 differ from each other not only for their data model, but even for the richness and complexity of their query language, which is different from one store to another.

Several research attempts have been made to standardize the query language for some of the NoSQL families [54] [55], but up to now, developers still have to write in the *chosen* store query language.

Due to their intrinsecal semplicity, key value store only provide key based put, get and delete, because any other complex query will only be a hassle to the system and his performance. If the developers feels he need a much more expressive way to query the store, then key value stores are not the right solution. As a side note, Couchbase is the only key value store that offers REST APIs.
REST (REpresentational State Transfer) is a stateless paradigm usually run over HTTP.

REST involves reading a designated Web page that contains an XML file which describes the desired content, include it, and usually offers methods to manipulate it, like *GET, POST, PUT* and *DELETE*

Document stores instead offers a much more complex query language, due to their ability to query keys, values and ranges.
Moreover, MongoDB adds count and distinct to the lot, and his queries can be extended using regular expressions.
The UnQL project [55] is working on a common language for document stores, recreating an SQL-like syntax to query JSON documents. Document stores usually exposes REST APIs as well.

Column stores provides a somewhat restricted language, populated mainly by range queries , operations like *IN* and *AND/OR* and regular expressions. Every family offers an SQL-like language, but their effectiveness is limited by the fact that only row key and indexed values can be used in *WHERE* clauses.

Both column and document stores provides MapReduce frameworks, in order to process huge amount of data and enable parallelized calculation over large datasets distributed on multiple machines.
Platform like Pig [56] and Hive [57] can be used to bridge the gap between the very low level of abstraction offered by MapReduce jobs and the query languages commonly used in the stores.

Graph databases offers a pattern matching to *extract* parts of the original graph that match the defined pattern, or can be used to traverse from a chosen node to a defined destination.
Most graph databases offer REST APIs to acces data using one of the described strategies.
A popular declarative query language is SPARQL [58] that with a very simple syntax provides graph pattern matching, and is supported by most graph stores, including Sesame, BigData and Neo4j.
On the other hand, Gremlin [59], is used to perform graph traversal based on XPATH.
FlockDB instead doesn't offer a query language, since its use is limited to 1-hop-neighbor relationship.

Chapter 3

MorPhyne, the New Approach

As stated in Section 1.4.2 we encountered some problems during the data collection phase that will be tackled using a whole new server architecture based on a non relational approach.
In Section 1.4 the current server architecture was presented and both hardware and software specification was summarized, but the gory details was not pointed out yet.
In Section 1.4.2 the problems we encountered after using it for a while was introduced, and as we proceed in this chapter they will be explained in more detail.

3.1 Brand new server architecture

Designed and developed with scalability and availability in mind, or in CAP terms, targeting the PA partition (Figure 2.1), the new system architecture will now be presented.

3.1.1 Hardware

GOZZO, is the host of the virtual machines army by Linux Kernel 2.6.32-5-amd64 .

On the hardware side, it's configuration is:

- Processing Unit: Intel® Core™ i5 650 @3.2 GHz
- Memory: 4 GB DDR2 @800 MHz
- HDD: 500 GB SATA II @7200 RPM

Instead of using physical machines, a more *virtual* approach was preferred, using several small virtual machines administered by Virtual Box [60] and kept alive as a system service, all of which powered by Ubuntu 12.04.02 Server amd64. The whole system is illustrated in Figure 3.1

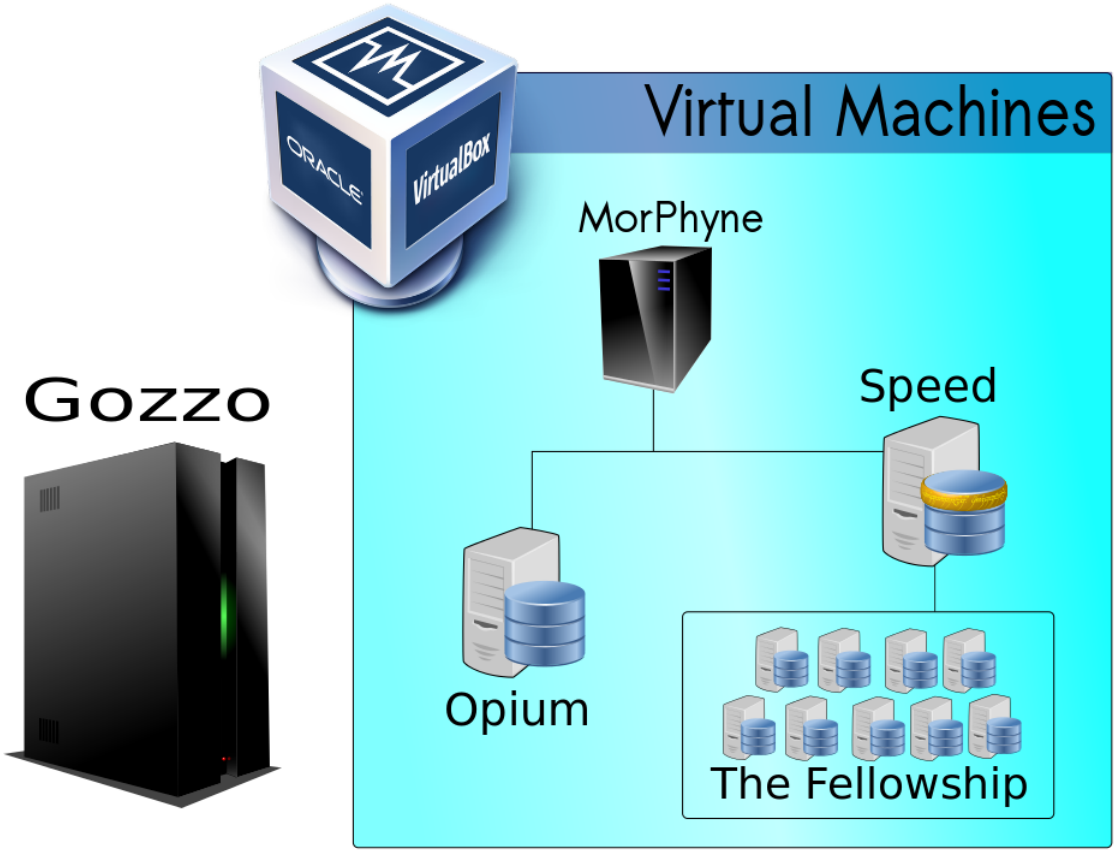

Figure 3.1: New VM based server infrastructure

MORPHYNE is the server that interface with the external world, including the application and the databases. It runs an instance of *Gunicorn* as a WSGI HTTP Server [61], backed by *nginx* as a reverse proxy to handle and demultiplex the HTTP requests [62]. *morphyne* is also the name of the web application, built from scratch in Python using *Flask* microframework [63], and it's the heart of the whole server architecture. For the sake of confrontation, two more VMs are in the field, **OPIUM** and **SPEED**.

OPIUM is our relational database server. It runs a standard installation of MySQL Server version 5.5.31. The server has been created to simulate the old SQL environment on the MORPHONE server, with the same schema and the same data model.

SPEED is the non relational kid. It's the new member of the MorPhyne family and will be of help in discerning whether or not we should chose MongoDB as the new database technology for MPower analytics. It runs MongoDb 2.4.5 with no replication, sharding or indexes enabled yet. Just a standard single machine implementation.

3.1.2 Virtualization Environment

In this first prototype, Virtual Box has been choosen as our virtualization manager over other more "mature" solutions for the following reasons:

- It is easy to setup and easy to use, with a full-flaged command line utility, *VBoxManage*, that makes possible to the sys admin every type of customization and runtime management he will need on the VMs.
- Due to it's simplicity, it is very fast to deploy and run VMs, thus ensuring that no time was wasted in ugly implementation details, while still prototyping the system.

- It poses no limitation on the migrations to other, more flexible and scriptable, virtualization manager, like KVM / QEMU

3.2 Chosen DB Motivation

As already mentioned, MongoDB has been chosen as our new non relational database. The choice fell on a document store, and in particular on MongoDB for several reasons:

- **BSON documents:** *"short for Binary JSON, is a binary-encoded serialization of JSON-like documents. Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays"* [64].

  Since log files have been moved from CSV to JSON (see Section 3.3 for more), using BSON gets even much more convenient.

- **Auto-Sharding:** *"Sharding is MongoDB's approach to scaling out. Sharding partitions a collection and stores the different portions on different machines. When a database's collections become too large for existing storage, you need only add a new machine. Sharding automatically distributes collection data to the new server and automatically balances data and load across machines[...] "* [65].

  With thousands of users in mind, one could only imagine what our database will became in a year or two, so it is way better to have an easy way to scale-out, instead of expensively scale-up.

- **Indexes:** *"Indexes provide high performance read operations for frequently used queries. Indexes are particularly useful where the total size of the documents exceeds the amount of available RAM."* [66]

Indexes could be used to strongly improve the performance of the queries used in the model generation and maintenance.
Also, they could be used to give a more responsive output to our data visualization environment (see Chapter **??** for more)

- **Replication:** *"A replica set in MongoDB is a group of mongod processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets"* [67]

  Even if data processing is an asynchronous operation (see Chapter 3.6) for more), we could not give up the pursuit of the availability that was start in Chapter 2.2

3.3 Log File to be processed

In the previous section, the log file was mentioned, referring to the transition from a CSV file to a JSON file. In this chapter we will see why this transition was mandatory, what was the former structure of the log file, and what is the newer structure.

3.3.1 CSV Log File and its Structure

The structure introduced in the first version of MPOWER was designed to be lightweight, fast and easy to read and write. Unfortunately, as the project grew and the users increased, some problems start to show up in the processing of the files, mainly due to the heterogeneity of the devices used. The file, structured as a collection of Comma Separated Value, once sent to the server, was then processed following a positional logic. Each

value was placed in the corresponding column of the corresponding table on the database, and everything worked like a charm.

```
1 1352975344409;0;33.0;3580;1;Li-ion;1600000;600000;600000;200000;ondemand...
2 1352975405669;0;33.0;3563;1;Li-ion;1600000;200000;600000;200000;ondemand...
3 1352975457944;0;33.0;3580;1;Li-ion;1600000;200000;600000;200000;ondemand...
4 1352975504830;0;33.0;3560;1;Li-ion;1600000;1000000;1200000;1000000;HYPER...
5 1352975515847;0;31.0;3568;1;Li-ion;1600000;200000;1200000;200000;HYPER;9...
```

Code 3.1: An example of the CSV log file

Then, one day, dual core smartphones kicked in and slowly eat an increasingly bigger piece of the cake, since they finally appeared on MPOWER too. The mismatch caused by the adjunct CPU, not envisioned in the first draft of the log file, caused every value after the second CPU to shift away, and to be written (or tried to) in the wrong column of the database.
Developers quickly worked on a bug fix, anticipating up to 4 cores.
After all, who needs more than 4 cores anyway?

But that was before Samsung announced the EXYNOS5 OCTA mobile processor during CES keynote, in January 2013 [68], with his brand new shining and rumbling 8 cores.
That made it finally clear that this sort of file structure was of no use to our heterogeneous logging requirements and we needed a completely new type of file structure.

3.3.2 JSON Structure

For this reason, during the design phase of the new server architecture, the log file got a full redesign and became a collection of adjacent JSON file, in which the keys points out the tables on the first level of depth and the fields on the second level. The structure is explained more clearly in Listing 3.2. However, to keep the log file as lightweight as possibile, we decided to

skim it of all the useless blank spaces and newlines, and to have a single log for every line. This means that for every log on the app, cadenced every 10 seconds, we will have a line on the log file, with a different timestamp of course. This structure is shown in Listing 3.3

```
{
        "timestamp": 1357601640,
        "screen":
                {
                        "height":1184,
                        "width":720
                },
        "wifi":
                {
                        "tx_bytes":1927104,
                        "connected":1,
                        "wifi_rx_bytes":3545954},
        "bluetooth":
                {
                        "bluetooth_on":0,
                        "bluetooth_state":10
                }
}
```

Code 3.2: A small extract of a JSON log file

```
{"timestamp":1357601640,"wifi":{"tx_bytes":1927104,"wifi_rx_bytes":3514...
{"timestamp":1357601650,"wifi":{"tx_bytes":2017124,"wifi_rx_bytes":3514...
{"timestamp":1357601660,"wifi":{"tx_bytes":2028804,"wifi_rx_bytes":3514...
{"timestamp":1357601670,"wifi":{"tx_bytes":2037104,"wifi_rx_bytes":3514...
{"timestamp":1357601680,"wifi":{"tx_bytes":2059904,"wifi_rx_bytes":3515...
```

Code 3.3: An example of the real "one-liner" log file

3.4 Communication Protocol

To safely complete the handshake procedure before accepting files sent over the internet, we devised a communication protocol directly evolved from its predecessor, actually used on MPOWER against MORPHONE. The protocol is visually illustrated in Figure 3.2, and will be explained in more details in the paragraphs below:

1. - **MPOWER:** sends a *#reset* message, to clear any other eventually pending sessions.

2. - **MPOWER:** sends a *#send* message, pointing out that he wants to begin an handshaking phase to send some log files.
   - **MORPHYNE:** reply with another *#send* message, acknowledging the request.

3. - **MPOWER:** sends the version of the protocol, *ver#2.0* in our case.
   - **MORPHYNE:** reply with the same protocol version, if it's a know protocol.

4. - **MPOWER:** then sends a message containing his *device_id*, claiming to be the device identified by it.
   - **MORPHYNE:** reply with the same *device_id*, adding a random 8 byte alphanumeric salt to strengthen the next request.

5. - **MPOWER:** encrypts his own device_id followed by the salt received from MORPHYNE with AES-256 (CBC) using his secret token as a symmetric key.
   - **MORPHYNE:** decrypt the received message, extract the salt and compare with his own, then extract the *device_id* and compare it with the one stored in the database. If both checks are passed, reply with an *auth#ok* message

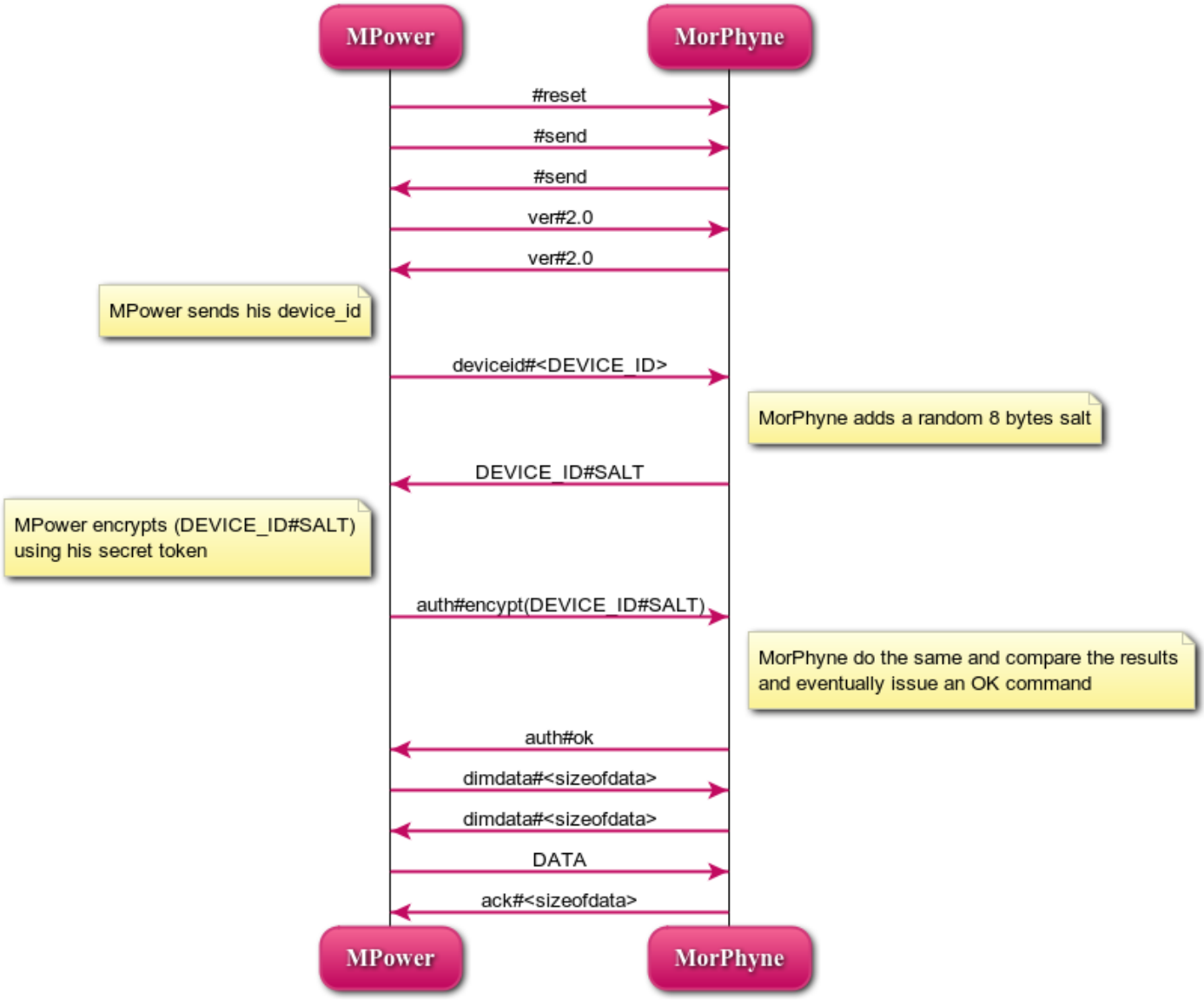

Figure 3.2: Worflow of the Communication Protocol version 2.0

3.5 System Behavior

After explaining how the data are under the hood, in this Chapter the system behavior and functionalities will be explained, both from the Android application and the server side.

3.5.1 Authentication

First of all, the device must be uniquely identified, in order to prevent abuse of the data sending.
We used *Pseudo-authentication* [69] for this purpose, that differs from standard authentication methods because the web application directly requests a limited access token from Google OAuth on behalf of the user, and the user grants us the permission to access his/hers information.
We then use the granted access to discover the personal information of the user, so we do not directly authenticate the user, but instead we lean against a third party to do this for us, and we get the response from them.

We decided to implement the pseudo-authentication using Google OAuth2 Login only to verify that the email used by the user is a legitimate Google email address. The whole mechanism is illustrated in Figure 3.3.

Let's be a little more fine grained about the authentication process:

1. We prompt the user through the application with a link on MORPHYNE that embeds some information for Google Authorization Server, and then let the user click on it.

2. Google then authenticate the user, showing him/her all the information we requested (name and email, in our case), and at last replies to MORPHYNE with a unique temporary token, which is the key to access the user information.

3. This token is then embedded in another request that MORPHYNE will send to Google, but to this time the *User Info API* is our destination. Contextually the token is both SHA256 hashed to be stored on our database, and wrote in a cookie for the app to read it, hash it as well and store in his local settings. This token will later be used in the

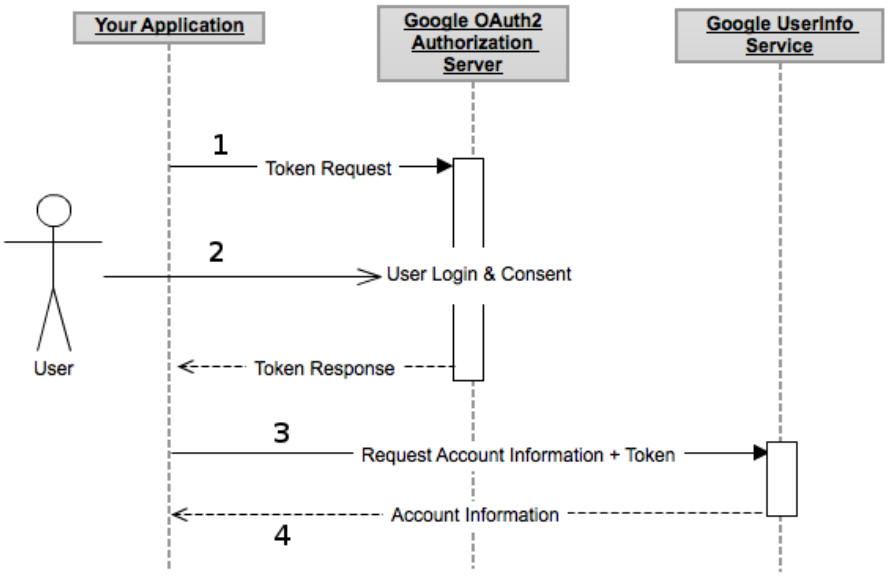

Figure 3.3: Google OAuth2 Login Flow

`https://developers.google.com/accounts/docs/OAuth2`

symmetric encryption/decryption of the log file (see Chapter 3.5.3 for more).

4. In the end Google replies to MORPHYNE with the requested user information, that in our case are only the basic personal information, such as Name and Surname, and its verified email.

5. With the information retrieved all along, we can create a new user on the database, identified by his email, some basic personal information, the hashed token and the device ID. This ID can be the IMEI of the phone or the MAC address of the network interface, in case the device lacks GSM module and consequently doesn't have an associated IMEI. Note that some extra columns are included but left blank, for legacy reasons.

Table 3.1: A couple of test users from the database

user_id	email	name	surname	country	GMT	privacy
10	test@gmail.com	Morphy	Tester		0	0
11	morp@gmail.com	Camal	Melonte		0	0

Table 3.2: A couple of devices from the database bound to the users by the ID

device_id	token	user_id	model	arch	kernel_version
0123456789	dcm87d78d23...	10	gt i9100	armv7l	3.0.16
e3:f3:57:c2	oinhf873hf8...	11	Hero	armv6l	2.6.29

6. The device is now ready to collect and send data to MORPHYNE without further need any other authentication or user intervention.

3.5.2 Data Collection

As been already mentioned in Chapter 1.3, MPOWER collects a lot of data regarding the user behavior, and store them locally on the file system, divided in 4 files per day. This is done to avoid losing an entire day of recording in case a file gets corrupted.

3.5.3 Encryption and Compression

At the end of the day, when the phone is on charge and the user is *much likely* sleeping, MPOWER compress the log file using *gunzip* compression [70] and then encrypts the compressed file with AES-256(CBC) symmetric algorithm using the previously exchanged token as the common secret key and an IV (Inizialitation Vector) randomly generated for every file, to en-

sure the best level of security.

In the older version, the cypher was done before the compression, thus wasting much of the compression potential that comes with very repetitive files. In MORPHYNE instead, we have flipped the phases in order to gain the maximum compression ratio before the encryption.
This is much more important in the new log file than in his predecessor, since the JSON structure is much more verbose than it's CSV counterpart (see Chapter 3.3 for more).

Gunzip, as reported in their website "*finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length)*". [71]

3.5.4 File delivery

After the data are prepared by means of encryption and compression, they are sent to MORPHYNE for the processing and storage. This step is divided in two consequential phases:

The first id the **Handshaking**, where the application starts a custom defined handshaking phase with the server in order to ensure that the data really comes from the alleged device. Several steps are performed in order to verify and acknowledge the identity of the sender, following our custom communication protocol (see Chapter 3.4 for more).

While the second, the Data sending, is the effective transfer of the encrypted log file from the application to the server. Several checks are made on the legitimacy of filename and dimension that must match the size declared in the handshaking. This checks are done order to provide a sufficient level of security for the system.

3.5.5 Backward Compatibility

Great effort was spent to maintain the highest level of backward compatibility towards the old users. We identified 3 types of users of MPOWER that we will encounter during the *transition* phase from the old version to the new one:

1. **The Rookie:** is the brand new user, that have never been registered on MORPHONE, our old server. These users will have no sort of problems, since they will use the latest version of MPOWER against the latest revision of MORPHYNE.

2. **The Migrator:** is the *aficionados* of MPOWER, always up-to-date with the latest release and registered in MPower since the crack of dawn. These users are really precious to us, and must be taken care of.

   The problem resides in the creation of the token. Due to the loss of previous implementation details and the lack of documentation on this topic, the creation of the token used in the symmetric encryption phase remained a black box since now. Given that, we came up with an alternate solution that doesn't require another registration from the users nor the loss of the user itself.

   We will use our new cryptographic infrastructure, with new JSON files and cryptographic algorithms, but instead of creating a new type of token we will keep using their old secret token, thus ensuring that we can continue to understand each other.
   As simple as it seems, this solution cost us several hours of brainstorming to migrate from a *throw everything away* attitude to a little bit less revolutionary *let's save the ship, at least* attitude.

3. **The Relentless:** installed an older version of MPOWER and never

updated it. Our typical relentless guy it's 5 to 10 releases behind the latest version and doesn't really care to update and have much likely forgotten about MPOWER. We will keep collecting their data and periodically migrate them to MORPHYNE and the new format, until one day the old MORPHONE server will be dismissed. By then they will have finally upgraded the app, uninstalled it or bought an iPhone... either way it's a *non critical* user that we can afford to lose.

3.6 Data Processing

When the data are finally stored on MORPHYNE filesystem, an asynchronous job is scheduled to process them.

The processing of files is well illustrated in Figure 3.4, and can described in details in the next paragraphs:

Decompression and decryption

The files gets decompressed and inflated to their original form, while they are still encrypted. After that, MORPHYNE extracts the secret token from the database using the device id stored in the filename as a key:

```
DEVICEID_2013-07-22_1.json
```

Basically what we have is an array delimited by "_" characters, in which the first element is the device id, the second is clearly the date of the log and the last one is the part of the daily log (as mentioned in Chapter 3.5.2).
Now that we have the secret key, we can decrypt the log file using the random IV stored in the first 16 bytes of the file and the token we just extracted from the DB. The file is now plain text ready to be parsed and stored.

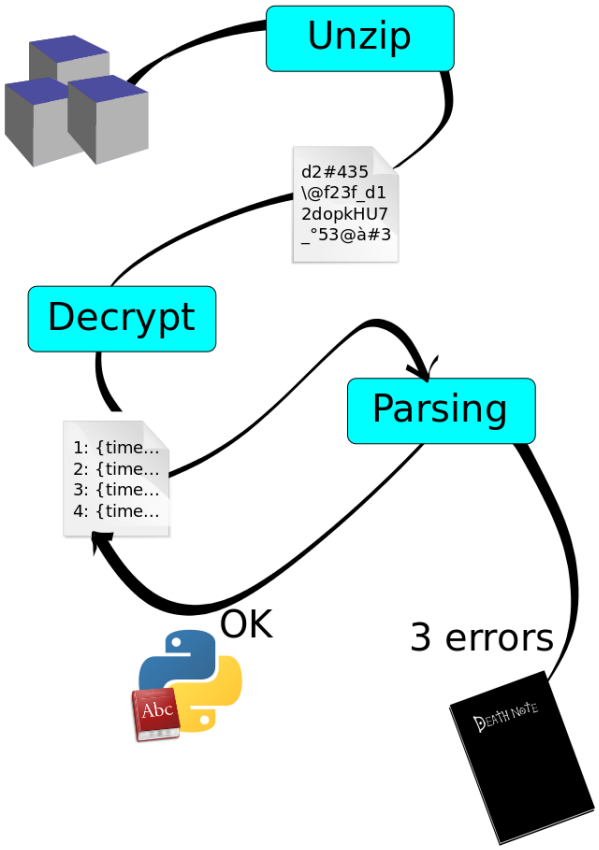

Figure 3.4: Workflow of the data processing

Parsing, and storing

Python, like many other high-level languages, offers a full fledged and intuitive interface to access and manipulate JSON files, a module called with no big surprises `json`.

The log file then gets read line by line from Penelope, the parser module, that reads throughout the file, translating each JSON line into a dictionary

structure which Python can process.
In the meantime, after each translated line, the output dictionary is given to the process that in turn disentangle the knots and then call the functions to store the data on the database.

In order to ensure a complete decouple between MORPHYNE, OPIUM and SPEED, I've also developed a simple DAL module (Database Abstraction Layer) that exposes some functions to whom it may needs them, without disclosing the implementation under the hood.
This decoupling grants not only an easy migration from the relational OPIUM to the non relational SPEED, but also ensures that only the function I've exposed can be executed, on both databases.

Anomaly detection and Fault Tolerance

In the first version of the data processing code no mechanism of fault detection was implemented. The result was that, when something caused the processing of a file to abruptly fail, the parser simply quit the job, trying again on the next scheduled one. However, when the failure was not recoverable, the parser got stuck in a *failing loop* from which the only exit was to manually delete the guilty file, and finally release the bolt on the file queue.

To avoid this unpleasant situation, whenever Penelope encounters an error processing a JSON line, she skips the entire file and write down the error code alongside the faulty file name, giving it 2 more chances to redeem.
If every one of the 3 attempts goes black, she has no other options but to inscribe the corrupted file in the *Death Note* structure. [72].
After having processed the other legitimate log files, Penelope execute the deletion of those whose name appears on the *Death Note*, while keeping a record of the fallen in an internal log file for bug track down purposes.

Chapter 4

Results and Comparisons

In this chapter the results of the SQL versus NoSQL comparison are offered. Let's keep in mind that this is a strongly use case oriented confrontation, where no optimization were done on both sides of the wall, to keep the quarrellers as square as possible. This means that the standard settings for both the architecture have been used, to compare their *out-of-the-box* behavior. The only exception is the addition of an index to MongoDB as will be presented in Chapter 4.2 to pair with the indexes that MySQL adds by default to its primary keys.

In order to make a fair comparison, the dataset of both technologies needed to be of the same order of magnitude. Given that, two different migrations were needed to achieve this setup:

1. Full SQL export of database *mpower_v0* from MORPHONE and import to *mpower_v0* of OPIUM. The user database was imported too, but its dimension is irrelevant, especially compared with *mpower_v0*.

   The export from MORPHONE took about **1 hour and 17 minutes**, while the import in OPIUM took **1 hour and 54 minutes** to accomplish.

2. Migration from SQL to NoSQL structure, that is, from OPIUM to SPEED. Since this was a critical and far from trivial task, it will be discussed in details in Section 4.1.

4.1 Data Migration

To efficiently convert the table structure of the relational database into a document structure, the whole migration had to be designed with this goal in mind.
The structure needed to be migrated from the table layout (like the one shown in Table 1.1) to a whole new document layout, like the one presented in Listing 4.1:

```
{
        "_id" : {
                "timestamp" : NumberLong("1363618083088"),
                "device_id" : "012529001822588"
        },
        "audio" : {
                "used_memory" : NumberLong(390592),
                "music_volume" : NumberLong(10),
                "audio_mode" : NumberLong(0),
                "speaker_on" : NumberLong(0),
                "ring_volume" : NumberLong(7),
                "music_active" : NumberLong(0),
                "voice_volume" : NumberLong(5)
        },
        "battery" : {
                "temperature" : 1,
                "health" : "2",
                "voltage" : NumberLong(4),
                "percentage" : NumberLong(100),
                "technology" : "Li-ion",
                "on_charge" : 1
        "bluetooth" : {
                "state" : NumberLong(12),
```

```
24                  "is_on" : 1
25          },
26          "cpu" : {
27                  "current_freq" : NumberLong(320000),
28                  "governor" : "ondemand",
29                  "cpu_id" : NumberLong(0),
30                  "info_max_freq" : NumberLong(600000),
31                  "info_min_freq" : NumberLong(122880),
32                  "usage" : NumberLong(69),
33                  "max_scaling_freq" : NumberLong(600000),
34                  "min_scaling_freq" : NumberLong(122880)
35          },
36          "gps" : {
37                  "status" : NumberLong(0),
38                  "is_on" : 0
39          }
40  }
```

Code 4.1: An extract of a real BSON document in SPEED

Moreover, the migration from the relational structure of OPIUM to the non relational structure of SPEED had to address some I/O and memory issue:

1. *"One does not simply walk into Mordor"* [73]: it was simply unreasonable to make a *"SELECT *"* statement for every table and then parse the returned structures. A Python dictionary containing more or less 250 millions of record will weight tens of GBs, all stored in RAM.
   A ridiculous scenario, of course, in which MORPHYNE will surely have severely crashed before even start the parsing, due to the missing great amount of physical memory, and the consequently high number of memory swapping.

2. The migration proved to be strongly memory bound, but the limited

resources that GOZZO could provide, wouldn't suffice for all the virtual machines as it would have been recommended.
With those premises, several configurations have been tried in order to achieve the best performance and stability on the overall process.

On the virtual hardware side, the best available configuration was:

- **MORPHYNE**: Single CPU, 2 GB RAM, 20 GB Dynamic Size HD.
  Since MORPHYNE it's doing the dirty work of retrieving the data from one side and storing to the other, 2 GB of physical memory was the minimum to manage the dataset without collapse (see Chapter 4.1.1 for more).

- **OPIUM**: Single CPU, 1 GB RAM, 50 GB Dynamic Size HD.
  OPIUM it's merely used to retrieve a relatively small amount of data per round, so there was no need to *over-power* it.
  MySQL database weighed around 30 GB, and that is the reason behind the 50 GB hard disk.

- **SPEED**: Single CPU, 2 GB RAM, 200 GB Dynamic Size HD.
  As was said before, the migration proved to be strongly memory bound, and SPEED would have need much more physical memory than 2 GB, but unfortunately it wasn't available.
  The migration have still been completed, but with at least 4 GB of memory it would have been much more fast.
  Given its more complicated structure, it was guessed that MongoDB structure will have weighed more than its MySQL counterpart, thus the sizing of 200 GB.
  In retrospect this have been a wise choice, since mpower database alone weighed a bit less than 80 GB.

However, to handle the software issue instead, a strong memory saving solution was needed combined with the best possible configuration of virtual hardware, thus leading to a time consuming migration.

4.1.1 The Humongous Dataset

As a matter of fact, there was two very different approach to reach the goal of migration while fighting with the huge dataset issue.

1. Make several *JOIN* as the one used in the model generation and then retrieve the *merged* data as a set of big single rows
2. Use a more *Divide and Conquer* approach, analyzing one table at a time, retrieving a subset of the dataset and writing down the whole table, one step at a time, and then proceeds to the next table, adding the new content to the already existing documents.

The second approach has been preferred on the first for its relaxed schema and for the more controlled and modular migration.

The pseudo-code in Listing 4.2 shows how the migration workflow was handled table by table, step by step. Another tuning in this migration was the dimension of the dataset returned for every loop iteration.

Since this would have dramatically impacted on the memory consumption of MORPHYNE, several configurations had been tried, eventually landing on **1 Million** records per iteration on every table.

With this configuration, the Python script steadily occupies **500 MB** of *phisical memory*, with peaks up to **1400 MB** during the fetching operations on OPIUM, and each iteration took about **15 minutes** to complete, with an overall extimated time to accomplish of **62 hours**.

```
1
2   opium = SQL_DB_CONNECTION
3   speed = MONGO_DB_CONNECTION
4
5   startRow = 0
6   blockSize = 1000000
7
8   tables = [ 'audio', 'battery', 'bluetooth', 'cpu', 'gps', 'mobile',
        'screen', 'wifi' ]
9
10  for tab in tables:
11
12        # return an iterable containing the rows from the query
13        sql_data = opium->query("SELECT * FROM mpower_v0.tab
14                                 LIMIT startRow, blockSize")
15
16        for data_row in sql_data:
17
18                # the 'parse' function simply converts the row in
19                # the json structure show in Listing 4.1
20                jsonStruct[ tab ] = parse(data_row)
21
22        # updates the collection 'data' on speed
23        # the upsert flag is ON
24        speed->data.update(jsonStruct, upsert = True)
25
26        startRow += blockSize
```

Code 4.2: Pseudo-code of the migration

From the Listing 4.2 one may have already noticed that instead of using a more canonical *insert* in the collection, a series of *update* was done. More specifically this updates are all performed with the *upsert flag* set to *TRUE*.

That is an ***upsertion***, and it is MongoDB way of saying:

```
1
2       # Starring:
3       # jsonID as a small dictionary for 'device_id' and 'timestamp'
4       # jsonData as a JSON formatted dictionary
5       # data as a MongoDB collection
6
7       if jsonData is already in data:
8               data.update({ '_id': jsonID },{ $set: jsonData })
9       else:
10              data.insert({ '_id': jsonID, jsonData })
```

Code 4.3: Upsertion exploded

Even though insertions in MongoDB are blazing fast, *upsertions* have been used since, at last, they can mimic the final structure of the JSON files received from the MPOWER application, while keeping some modularity on the migration code. The choice surely had a big impact on the overall performance of the migration, especially on the duration of it, but it seemed a very good way of simulating the real environmental condition that MORPHYNE will have to deal in the near future.

4.2 Power Model Selection Query

As have already been explained in Section 1.3, in order to generate the power model for a specific device, it is mandatory to retrieve the entire history of the device itself from the database.

Doing this has proved to be very difficult on MORPHONE. In fact, the *standard* retrieval query has never been completed on it.
Instead, as already explaind, the strange fact was the it runs smooth and clean on a laptop test machine with a standard MySQL configuration and an almost identically powerful hardware than MORPHONE (see Section 1.4.1):

- Processing Unit: Intel® Core™ i5 @1.7 GHz
- Memory: 4 GB DDR3 @1333 MHz
- HDD: 1000 GB USB External Drive @7200 RPM

The selection query, written in his SQL form is reported in Listing 4.4, while his MongoDB equivalent is written in Listing 4.5. One can notice that the queries differ from each other mainly for *JOIN* form, while it's trivial to see that they aim at retrieving the same fields.

```
1   SELECT c.device_id,
2           b.on_charge, c.cpu_id,
3           m.airplane_mode, m.data_state,
4           w.is_on, blue.is_on,
5           g.is_on, s.is_on,
6           s.brightness_value, c.timestamp,
7           b.percentage, m.signal_strenght,
8           m.tx_bytes, m.rx_bytes,
9           m.call_state, m.net_type,
10          w.is_connected, w.tx_bytes,
11          w.rx_bytes, c.usage,
12          b.voltage
13  FROM data_cpu c
14          INNER JOIN data_mobile m
15                  ON (c.timestamp = m.timestamp and c.device_id =
                        m.device_id)
16          INNER JOIN data_screen s
17                  ON (c.timestamp = s.timestamp and c.device_id =
                        s.device_id)
18          INNER JOIN data_blue blue
19                  ON (c.timestamp = blue.timestamp and
                        c.device_id = blue.device_id)
20          INNER JOIN data_gps g
21                  ON (c.timestamp = g.timestamp and c.device_id =
                        g.device_id)
22          INNER JOIN data_wifi w
23                  ON (c.timestamp = w.timestamp and c.device_id =
                        w.device_id)
24          INNER JOIN data_battery b
25                  ON (c.timestamp = b.timestamp and c.device_id =
                        b.device_id)
26  WHERE c.device_id = '359118041296606';
```

Code 4.4: MySQL version of the power model selection query

```
1
2       db.data.find({'_id.device_id' : '359118041296606',
3                           'wifi': { $exists : true },
4                           'bluetooth': { $exists : true },
5                           'gps': { $exists : true },
6                           'screen': { $exists : true },
7                           'battery': { $exists : true },
8                           'cpu': { $exists : true },
9                           'mobile': { $exists : true },
10                      }, {
11                          'wifi.is_on' : 1,
12                          'bluetooth.is_on' : 1,
13                          'gps.is_on' : 1,
14                          'screen.is_on' : 1,
15                          'battery.on_charge' : 1,
16                          'battery.voltage' : 1,
17                          'battery.percentage' : 1,
18                          'cpu.cpu_id' : 1,
19                          'cpu.usage' : 1,
20                          'mobile.airplane_mode' : 1,
21                          'mobile.data_state' : 1,
22                          'mobile.call_state' : 1,
23                          'mobile.net_type' : 1,
24                          'mobile.tx_bytes' : 1,
25                          'mobile.rx_bytes' : 1,
26                          'mobile.signal_strenght' : 1,
27                          'wifi.is_connected' : 1,
28                          'wifi.tx_bytes' : 1,
29                          'wifi.rx_bytes' : 1,
30                          'screen.brightness_value' : 1
31                      })
```

Code 4.5: MongoDB version of the power model selection query

4.2.1 Benchmarks and Comparison

Let us look at both queries from a performance point of view to see if an actual improvement in terms of performance has been achieved, with respect to the old database infrastructure.
In order to do this, the selection query has been run several times with different *parameters* and customization, as will be illustrated in this chapter.

The queries were run searching for 3 different *device_id*, which returned 0, 242989 and 96053 document objects, with different performance results. In particular the 0 results query was done for evaluation purposes only, to show how the caching mechanism could improve even the queries that doesn't actually return any document.

First of all, the query shown in Listing 4.5 was run against the data collection using the *out-of-the-box* MongoDB installation, without adding any optimization on it.

Then again, the same queries, but this time adding the Index shown in Listing 4.6 to the collection, which *covered* the wannabe retrieved fields and ensured the access to MongoDB caching mechanisms.

```
1   db.data.ensureIndex({
2                   '_id.deviceid' : 1,
3                   'wifi.is_on' : 1,
4                   'bluetooth.is_on' : 1,
5                   'gps.is_on' : 1,
6                   'screen.is_on' : 1,
7                   'battery.on_charge' : 1,
8                   'battery.voltage' : 1,
9                   'battery.percentage' : 1,
10                  'cpu.cpu_id' : 1,
11                  'cpu.usage' : 1,
12                  'mobile.airplane_mode' : 1,
13                  'mobile.data_state' : 1,
14                  'mobile.call_state' : 1,
15                  'mobile.net_type' : 1,
16                  'mobile.tx_bytes' : 1,
17                  'mobile.rx_bytes' : 1,
18                  'mobile.signal_strenght' : 1,
19                  'wifi.is_connected' : 1,
20                  'wifi.tx_bytes' : 1,
21                  'wifi.rx_bytes' : 1,
22                  'screen.brightness_value' : 1
23          },{
24                  'name' = 'modelIndex'
25          })
```

Code 4.6: Code of the Index built against the 'data' collection

The collection `mpower` on MongoDB was widely tested with and without the aforementioned index, to evaluate its efficiency on the selection query used to generate the power model.

Not surprisingly, the queries got very different performance results, as shown in Figure 4.1.
As soon as the caching mechanism kicks in, the queries gets greatly speed up, reaching more than acceptable execution timings. With the database collection now indexed, it was interesting to evaluate the effect of it on reiterated queries and after certain *system events* that could actually occur in a production environment, like `mongod` service restart, system reboot and system shutdown.

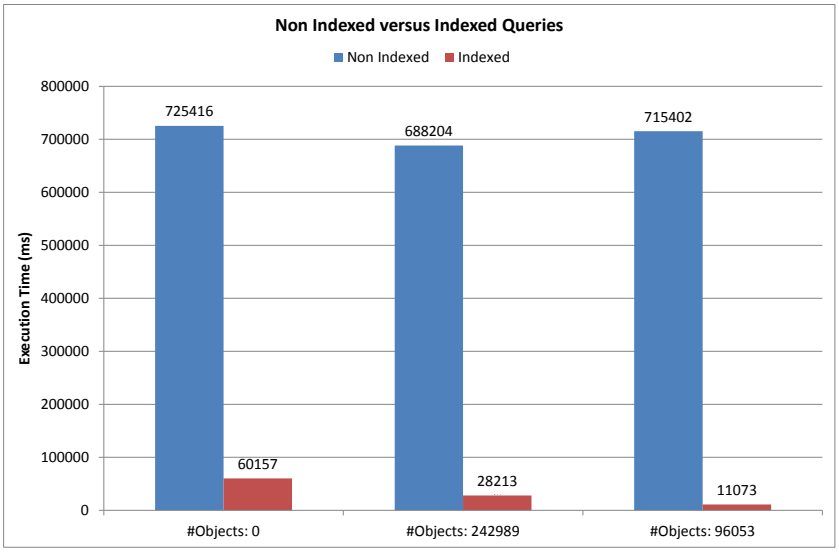

Figure 4.1: Comparison of Indexed versus Non Indexed Queries

Figure 4.2 shows the effects of data caching upon repetitive queries.
The improvement against the first run is clearly visible, ranging **from 16 to 28 times faster** than its non cached counterpart.

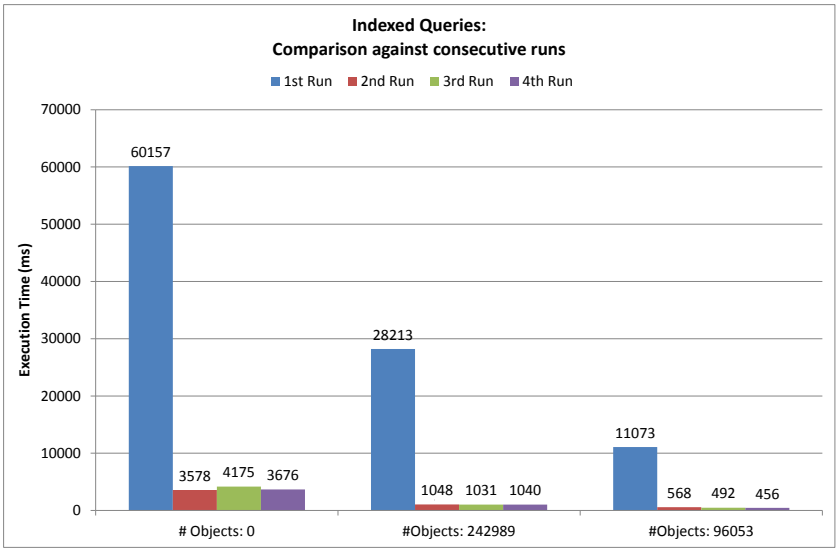

Figure 4.2: Caching mechanism cut the cost of querying over time on MongoDB

To experimentally measure the caching potential of MongoDB, the system was tested starting from a standard boot followed in close sequence by a system reboot, a `mongod` service restart, 2 more reboots and finally a shutdown and a fresh boot again.
Every step was done to evaluate the potential wiping of the cache by the `mongod` service.
Between every step, the same query shown in Listing 4.5 was run consequentially 4 times on the system to experimentally measure the caching potential after these *cleaning* events, and the results are shown in Figure 4.3.

It's fascinating to see how MongoDB eventually figured out that this dataset was retrieved over and over again, and from that point on, the data got cached in a more permanent way, thus ensuring stable and fast querying over that particular dataset of the collection.

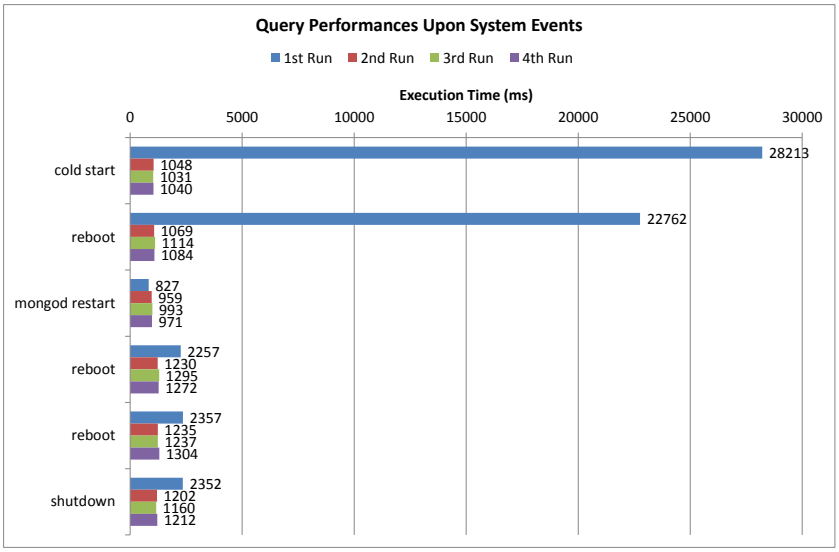

Figure 4.3: Indexed versus Non Indexed Queries on MongoDB

Moreover, it was surprising discover that while going back to the first *hard cached* query, after having done tens of other queries, haven't wiped the cached data, withholding the same great performance as before.

Another interesting result is shown in Figure 4.4, where the query was done using a *device_id* that did not match any criteria, thus resulting in an empty set. The non indexed query obviously scan all the data set before realizing that the requested document cannot be found, that is the whole 25770269 documents collection.
The query on the indexed collection instead, only scanned 881913 documents, speeding up the querying from 12 minutes to one minute straight.

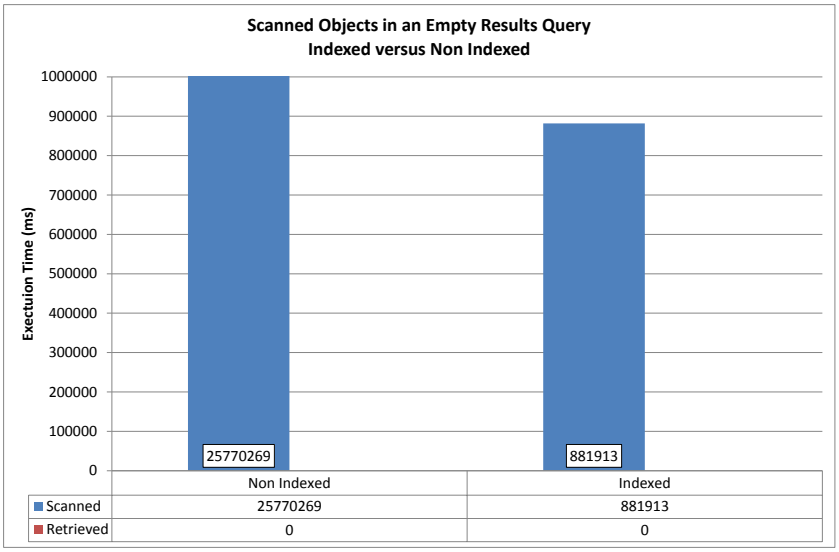

Figure 4.4: Scanned Objects in Empty Sets on MongoDB

4.2.2 MySQL versus MongoDB direct comparison

Now that we have deeply analyzed the behavior of MongoDB queries, it is time to make some comparisons with his relational opponent, MySQL. By default MySQL index all the primary keys in his tables, that for our database schema are `device_id` and `timestamp`, present on every table of the schema.

Given that, every query with the `WHERE device_id = 'xxxxxxxxxxxxxx'` statement is covered by an index, thus ensuring way faster results.
In Figure 4.5 we could see the results of our little battle for database dominance, where the contenders bravely fought for the trophy, but none of them can declare itself the clear winner.

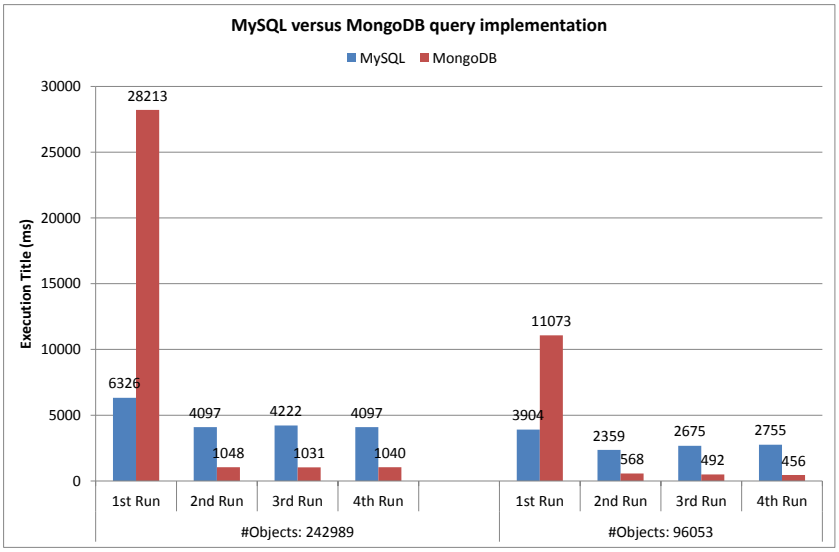

Figure 4.5: MySQL against MongoDB direct confrontation

Coming back to the data, one could observe that MySQL queries gives steadier results in every run, while MongoDB starts a bit rusted, but immediately gains a huge advantage both on its previous run and MySQL successive runs.

This behavior is clearly imputable to MongoDB caching mechanism which stores in RAM memory the most recently used data, in indexed, serving the queries directly from there. [74]

Chapter 5

Conclusions and Future Works

The main purpose of this thesis work was to compare and evaluate two different database technologies in a specific use case.
On one hand the classic SQL technologies, *the past*, was tested, while on the other hand there were NoSQL databases, nowadays often referred to as *the future of databases*.

This strong categorization may lead to awkward usage of these NoSQL technologies, even when they don't really fit the problem, just because they are *trendy* and commonly named *the future*.
This could easily lead to ill-choices in designing a system architecture, as this thesis have experimentally proved that even in a dataset that may seems huge, like the one that was used, the performance gain wasn't really that big.

However, performances aren't the only marker that should be taken into account while designing an infrastructure that could support thousand od users with potentially terabytes of data. MongoDB offered a relatively simple way to manage these kind of datasets, while offering a simple path to horizontal scalability, by means of his *sharding* technology.

That said, the introduction of MongoDB in our system could be a great advantage both in terms of scalability and performance over time, since the dataset is intended to grow exponentially, not mentioning the reliability that could be obtained through replication.

5.1 Future Works

The aftermath of the redesign work done on the infrastructure has indeed proved some good results, as shown in Chapter 4, like the read performance over indexed data or the modularity and fault tolerance of the virtual machines environment.

Nevertheless, a lot of work can still be done to further improve this work, and some hints will be given in the next Sections of this Chapter.

5.1.1 About using MongoDB on MorPhyne

Good results have been obtained, as already said and shown in Chapter 4, in terms of performance and flexibility and through the adoption of a *schema-free* technology, and much more could be obtained *fine tuning* MongoDB.

For example, the replication could lead to better read performance, and the sharding could ensure stability and fault tolerance, as well as horizontal scalability.
The indexes can still be improved, trying for example a different type of `'_id'` that the one we actually use, shown in Listing 4.1. It can be argued that having a *double* `'_id'` can be harmful in terms of indexing performances, thus changing this to an unique `'_id'`, like the *Object_ID* used as a default by MongoDB could improve its caching potential.

Another important aspect that could be investigated is the persistence of the cached data in memory, and how this data are prioritized against each other. A better understanding of this mechanism could lead to a better query planning, always aimed at achieving the best read performance possible.

5.1.2 About the System Infrastructure

The new system architecture is more reliable and fault tolerant than the old one, mainly thanks to the *virtual approach*, i.e. the use of several virtual machines, that has substituted the physical one.
Thanks to the use of virtual appliances, in fact, the system gained modularity on its components, as shown in Section 3.1, such as that whenever a virtual machine fails or get broken, it can be removed and replaced by another one without causing much distress on the overall system.

A cloud environment has also been envisioned, where the virtual machines could be replicated and populated with little to no effort.

However, some work can be done in this direction too, experimenting for example different virtualization engines capable of a better dynamic resource allocation or with deploy automation tools like Vagrant [75].
Vagrant can create reproducible and portable work environment on top of the biggest virtualization engines on the market and, with the integration of provisioning tools like Chef [76] or Puppet [77], can be used to automatically install and configure software on the machines.

Bibliography

[1] E. F. Codd, "A relational model of data for large shared data banks," *ACM*, vol. 13 Issue 6, pp. 377–387, 1970.

[2] A. Silberschatz, H. Korth, and S. Sudarshan, "Database System Concepts," *5th ed. McGraw Hill*, 2006.

[3] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V.Watson, "System R: relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, pp. 97–137, 1976.

[4] M. Indrawan-Santiago, "Database research: Are we at a crossroad? Reflection on NoSQL," *Network-Based Information Systems (NBiS)*, vol. 15th International Conference, pp. 45–51, 2012.

[5] M. Stonebreaker and r. Cattell, "10 rules for scalable performance in 'simpleoperations' datastores," *ACM 54*, vol. 6, pp. 72–80, 2011.

[6] S. Bagui, "Achivements and Weaknesses of Object-Oriented Databases," *Journal of Object Technology n.4*, vol. 2, pp. 29–41, 2003.

[7] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys (CSUR)*, vol. 15 Issue 4, pp. 287–317, 1983.

[8] http://research.google.com/archive/mapreduce.html.

[9] http://research.google.com/archive/bigtable.html.

[10] http://www.youtube.com/watch?v=Br2_22e2RoU.

[11] http://www.mongodb.org/.

[12] http://neo4j.org/.

[13] http://wiki.basho.com/.

[14] N. Levitt, "Will NoSQL Databases Live Up to Their Promise?," *Computer, no 2*, vol. 43, pp. 12–14, 2010.

[15] http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext.

[16] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. Brewer, M. Carey, and S. Chaudhuri, "The Claremont report on database research," *SIGMOD Rec. 37*, vol. 3, pp. 9–19, 2008.

[17] http://en.wikipedia.org/wiki/Big_data.

[18] A. Bonetto, M. Ferroni, D. Matteo, A. Nacci, M. Mazzucchelli, D. Sciuto, and M. Santambrogio, "MPower: Towards an adaptive power management system for mobile devices," *Computational Science and Engineering (CSE)*, pp. 318–325, 2012.

[19] N. Lane, E. Miluzzo, P. Lu, H. T. D. Choudhury, and A. Campbell, "A survey of mobile phone sensing," *Ad Hoc and Sensor Networks*, 2010.

[20] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pp. 12–12, 2011.

[21] A. C. Rice and S. Hay, "Decomposing power measurements for mobile devices.," in *PerCom*, pp. 70–78, IEEE Computer Society, 2010.

[22] M. Anand, E. B. Nightingale, and J. Flinn, "Ghosts in the machine: interfaces for better power management," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, (New York, NY, USA), pp. 23–35, ACM, 2004.

[23] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, "Energy management in mobile devices with the cinder operating system," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 139–152, ACM, 2011.

[24] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2010.

[25] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 168–178, ACM, 2009.

[26] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski, "A system-level model for runtime power estimation on mobile devices," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, (Washington, DC, USA), pp. 27–34, IEEE Computer Society, 2010.

[27] X. Zhao, Y. Guo, Q. Feng, and X. Chen, "A system context-aware approach for battery lifetime prediction in smart phones," in *Proceedings*

of the 2011 ACM Symposium on Applied Computing, SAC '11, (New York, NY, USA), pp. 641–646, ACM, 2011.

[28] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 153–168, ACM, 2011.

[29] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, (New York, NY, USA), pp. 335–348, ACM, 2011.

[30] Y. Wen, R. Wolski, and C. Krintz, "Online prediction of battery lifetime for embedded and mobile devices," in *Proceedings of the Third international conference on Power - Aware Computer Systems*, PACS'03, (Berlin, Heidelberg), pp. 57–72, Springer-Verlag, 2004.

[31] Smart Battery System Implementers Forum, "Smart battery data specification(v1.1)," 1998.

[32] M. B. Kjærgaard and H. Blunck, "Unsupervised Power Profiling for Mobile Devices," in *Proceedings of the 8th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2011)*, Springer, 2011.

[33] S. Gurun and C. Krintz, "A run-time, feedback-based energy estimation model for embedded devices," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, (New York, NY, USA), pp. 28–33, ACM, 2006.

[34] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery be-

havior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[35] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice, "Exhausting battery statistics: understanding the energy demands on mobile handsets," in *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, MobiHeld '10, (New York, NY, USA), pp. 9–14, ACM, 2010.

[36] J.-M. Kang, S. seok Seo, and J. W.-K. Hong, "Personalized battery lifetime prediction for mobile devices based on usage patterns," *Journal of Computing Science and Engineering*, vol. 5, no. 4, pp. 338–345, 2011.

[37] Y. Xiao, P. Savolainen, A. Karppanen, M. Siekkinen, and A. Ylä-Jääski, "Practical power modeling of data transmission over 802.11g for wireless applications," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, (New York, NY, USA), pp. 75–84, ACM, 2010.

[38] G. Wang and J. Tang, "The NoSQL principles and basic application of Cassandra model," *Computer Science & Service System (CSSS)*, pp. 1332 – 1335, 2012.

[39] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," *Pervasive Computing and Applications (ICPCA)*, pp. 363–366, 2011.

[40] `http://aws.amazon.com/dynamodb/`.

[41] `http://www.project-voldemort.com/voldemort/`.

[42] `http://redis.io/`.

[43] `http://www.couchbase.com/membase`.

[44] http://www.json.org/.

[45] http://couchdb.apache.org/.

[46] F. C. et al., "Database: A distributed storage system for structured data," *ACM Trans. on Computer Systems*, vol. 26, pp. 1–26, 2008.

[47] http://hbase.apache.org/.

[48] http://hypertable.org/.

[49] http://cassandra.apache.org/.

[50] http://www.sones.com.

[51] http://www.openrdf.org/.

[52] http://www.systap.com/bigdata.htm.

[53] https://blog.twitter.com/2010/introducing-flockdb.

[54] E. Meijer and G. Bierman, "A Co-Relational model of data for large shared data banks," *Communications of the ACM*, vol. 54, pp. 49–58, 2011.

[55] http://www.unqlspec.org/.

[56] http://pig.apache.org/.

[57] http://hive.apache.org/.

[58] http://www.w3.org/TR/rdf-sparql-query/.

[59] https://github.com/tinkerpop/gremlin/wiki.

[60] https://www.virtualbox.org/.

[61] http://gunicorn.org/.

[62] http://nginx.org/en/.

[63] http://flask.pocoo.org/.

[64] http://bsonspec.org/.

[65] http://docs.mongodb.org/manual/core/sharded-clusters/.

[66] http://docs.mongodb.org/manual/indexes/.

[67] http://docs.mongodb.org/manual/replication/.

[68] http://www.samsung.com/global/business/semiconductor/news-events/press-releases/detail?newsId=12521.

[69] P. Murukutla, "Single sign on for cloud," *Computing Sciences (ICCS)*, pp. 176 – 179.

[70] http://www.gzip.org/.

[71] http://www.gzip.org/algorithm.txt.

[72] http://en.wikipedia.org/wiki/Death_Note.

[73] The Lord of The Rings - The Fellowship of The Ring.

[74] http://docs.mongodb.org/manual/faq/fundamentals/#does-mongodb-handle-caching.

[75] http://www.vagrantup.com/.

[76] http://www.opscode.com/chef/.

[77] https://puppetlabs.com/.