FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INFORMATICS INSTITUTE
BACHELOR OF COMPUTER SCIENCE

PEDRO MARTINS DUSSO

A Monitoring System for WattDB: An Energy-Proportional Database Cluster

Graduation Thesis

Prof. Dr. Cláudio Fernando Resin Geyer
Advisor

Dipl.-Inf. Daniel Schall
Coadvisor

Porto Alegre, July 2012

CIP – CATALOGING-IN-PUBLICATION

Dusso, Pedro Martins

A Monitoring System for WattDB: An Energy-Proportional Database Cluster / Pedro Martins Dusso. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2012.

68 f.: il.

Graduation Thesis – Federal University of Rio Grande do Sul. Bachelor of Computer Science, Porto Alegre, BR–RS, 2012. Advisor: Cláudio Fernando Resin Geyer; Coadvisor: Daniel Schall.

1. WattDB. 2. Database. 3. Distributed-monitoring. 4. Database-monitoring. 5. Energy-proportionality. I. Geyer, Cláudio Fernando Resin. II. Schall, Daniel. III. Título.

"Study the past if you would define the future"

— CONFUCIUS, 553 BC

Para Carolina Ferret de Oliveira

pelo chimarrão, pipoca e companhia
nas longas tarde de estudo de domingo

ACKNOWLEDGEMENTS

I would like to thanks my parents Eduardo Vani Dusso and Cláudia Martins Dusso, which invested hard in education, not only financially but also with their time and love. Thank you mother for always having left me immersed in knowledge. Thank you father for always made clear the next step to take and never let me get out from the road, I really believe to be carrying the baton further than where I got it.

I would like to thanks also to Federal University of Rio Grande do Sul (UFRGS) and all people which keep fighting for it. It's not easy being one of the best universities in Brazil and still be completely free. Thanks for all employee, professors and students which make it a university with excellence and international prestige. I would like to acknowledge specially Prof. Dr. Raul Weber and Profa. Dra. Taisy Weber, for being parents to more than five hundred students inside the Informatics Institute. I want to thanks to Prof. Dr. Cláudio Fernando Resin Geyer, who received me as well this work in his research group as soon as I arrived in Brazil. Special thanks to Pedro de Botelho Marcos, that had dedicated so much of his time in order to correct every single detail in the final version of this text.

Thanks for Technical University of Kaiserslautern for its international program, accepting with open arms all Brazilian students. Special thanks to Prof. Dr. Theo Härder, for leading the Database and Information group and adopting us in our attempt to become even better scientists. I would like to thanks also Dipl.-Inf. Daniel Schall, the advisor of this thesis, for the infinite patience and precisely revisions in each part of this work. Thanks to Dipl.-Inf. Caetano Sauer for inspirational conversations and ideas, because sometimes we must leave our mind free to create.

Finally, there are so many important people to say thank you that this section would be bigger than the rest of the text. I would like to express my grateful to others who have directly or indirectly helped me through this five university years; I am pretty sure they will be the best years of my life.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

DB	Database
DBMS	Database Management System
DBA	Database Administrator
SQL	Structured Query Language
SEV	Soft Events
HEV	Hard Events
HDB	Historical Internal Database
MM	Monitoring Manager

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

WattDB is a locally distributed database system that runs on a cluster of lightweight nodes. It aims to balance power consumption proportionally to the system's load by dynamically powering its nodes individually up and down. Monitoring can serve as a basic building block for enabling adaptive and autonomic techniques, taking a special place in the cluster dynamics. This work provides a framework for monitoring and storing system usage counters and events of interest within the database. Monitored information is mapped, collected and consolidated to be saved in a historical database, which can be used to predict future query workloads in the cluster. This framework aims to be accurate, provide relevant and timely data, incurring low overhead in the nodes and in the network.

Keywords: WattDB, database, distributed-monitoring, database-monitoring, energy-proportionality.

RESUMO

WattDB é um sistema de banco de dados localmente distribuído. Seu objetivo é balancear proporcionalmente o consumo de energia com a carga de trabalho do sistema, dinamicamente ligando e desligando seus nodos individualmente. A monitoração do sistema é um bloco básico para possibilitar medidas adaptativas e autônomas, tendo um papel especial dentro das dinâmicas do cluster. Esse trabalho contribui com um framework para monitorar e armazenar estatísticas de uso do sistema e eventos de interesse dentro do escopo do banco de dados. As informações monitoradas são mapeadas, coletadas, consolidadas para então serem salvas num banco de dados histórico, que pode ser usado para prever futuras cargas de trabalho no cluster. Este framework visa ser preciso, fornecer dados relevantes e constantes, sem provocar sobrecarga de trabalho nos nós e na rede.

Palavras-chave: WattDB, database, distributed-monitoring, database-monitoring, energy-proportionality.

1 INTRODUCTION

This chapter provides an introduction for this bachelor's thesis, which is the design and implementation of a monitoring system for a distributed database. It collects and serves information about the database nodes to support the cluster management. Also the motivation behind the work and the project will be discussed. The objectives will be listed and defined. Finally, the organization of the text will be detailed.

1.1 Motivation

Nowadays the whole technology industry — including database systems — dedicates several efforts to become green. Sustainable development is not only driven by government and non-governmental organizations (NGO) pressures anymore; curb the negative impacts of human involvement became a concern for every company. Energy problem is not exclusive from database systems — not even solely of computers. Energy efficiency use tries to reduce the amount of energy required to provide products and services; e.g., building insulating, fluorescent lights and smart fan systems. In computers, three types of energy efficiency can be reached: educational, hardware and software efficiency (Bray 2006). Educational energy efficiency relies in user habit of turning off their computers and monitors when not in use — also called manual power management. It can achieve impressive results, nevertheless only for home and office environments, computers with directly users — it is not a scalable solution for clusters and clouds, scenario where database systems mainly play their role.

For hardware energetic efficiency, the smaller transistor size, the principal reason towards increased performance and reduced costs in computers (as can be seen in Figure 1.1), also results in power use reduction. This explains why the industry has been able to improve computational performance and electrical efficiency at similar rates (Koomey et al. 2009). Within a few years, the conventional CMOS transistors will no longer be able to scale down at a Moore's Law pace. But energy efficiency is not enough (Härder et al. 2011) to achieve the large amount of energy saving aimed.

Figure 1.2 shows computations per kWh grew about as fast as performance for desktop computers starting in 1981. Doubling every 1.5 years, a pace of change in computational efficiency comparable to that from 1946 to the present. Computations per kWh grew even more quickly during the vacuum tube computing era and during the transition from tubes to transistors but more slowly during the era of discrete transistors. As expected, the transition from tubes to transistors shows a large jump in computations per kWh (Koomey et al. 2009).

Ideally, the power consumption of a component (and the system) should be determined by its utilization. This energy proportional behavior, also called energy proportionality,

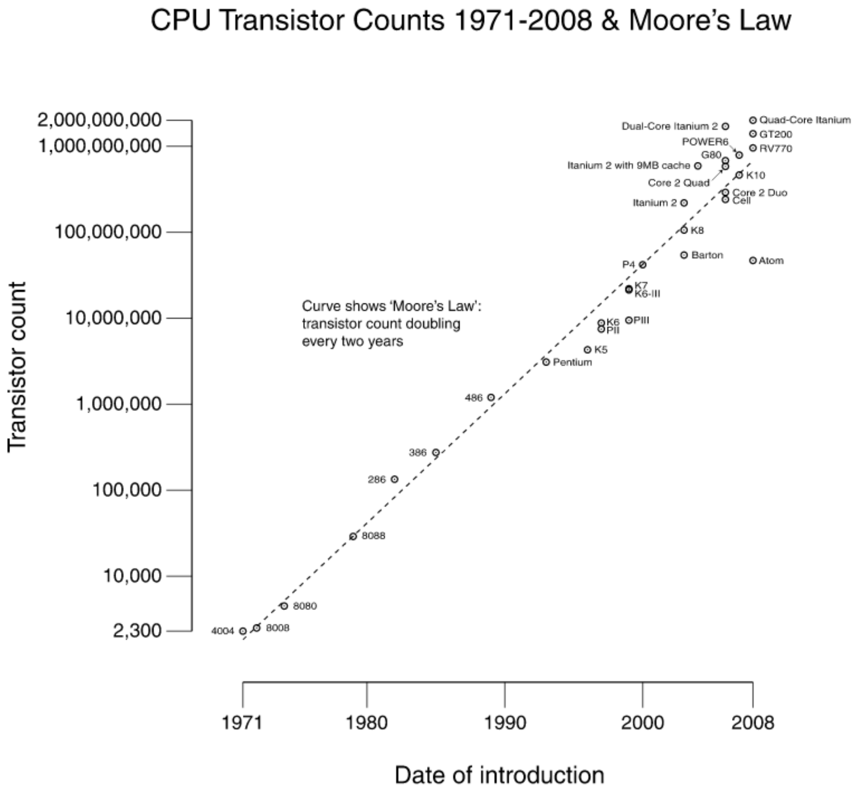

Figure 1.1: CPU transistor counts between 1971 and 2008 and Moore's law. Extracted from (Koomey et al. 2009).

is well developed by CPUs since processors are adopting energy-efficiency techniques more aggressively than other system components (Barroso e Hölzle 2007). Main-memory power usage, however, is more or less independent of system utilization and increases linearly with the memory size, following (Härder et al. 2011). If not all components in a system hardware architecture can provide energy proportionality, how should a server system be designed to address the server' energy efficiency characteristics with respect to server workloads?

Automatically power management relies in software techniques to power down machines or its components when not needed and to turn them on when the workload demands. Simulate by software the proportional behavior found in some hardware components can be the key technique to accomplish energy savings in large, industrial scale while preserving the necessary computational performance. The WattDB project, described in Section 2.4, uses simple hardware due to budget restrictions; nevertheless even huge companies, with unlimited access to state of the art hardware components, still have energy issues — economical or environmental. The solution, today, is beyond hardware technologies. Facebook is deploying an entire new server farm in the Article Circle, using the natural cold climate to handle the grid overheat (Ritter 2011). To attend the desired data throughput for a company as large as Google (which response easily one billion requests every day (Google 2011)) the physical structure needed in terms of servers, switches, routers, cables, buildings and refrigerating is something never seen before.

Successful businesses are not impelled only by mathematical modeling and optimization in production, logistic and consumption models anymore; accurate, relevant and timely information is the key ingredient to effective decision-making in almost any management position in different industry fields — energy, finance, government, health, legal,

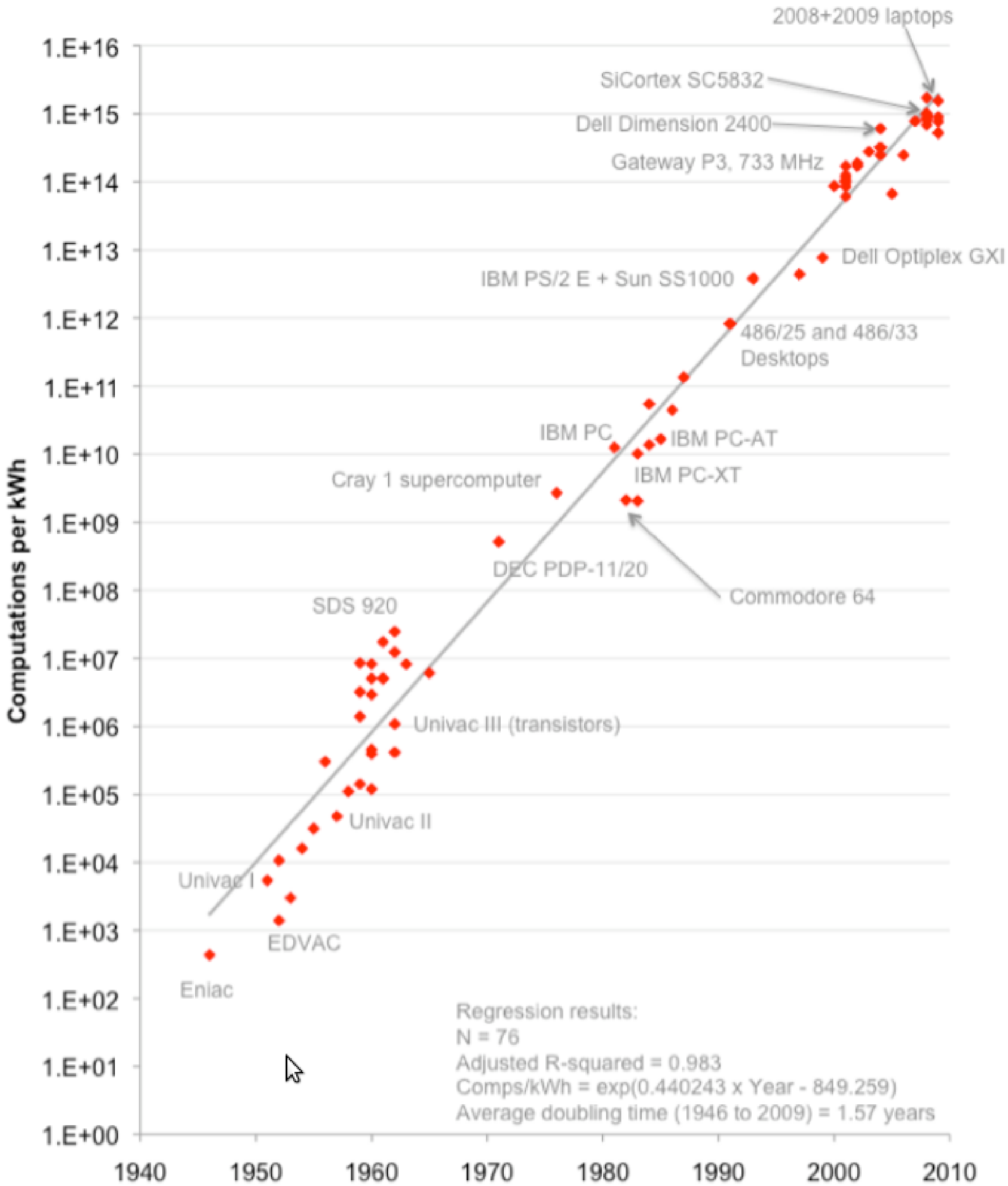

Figure 1.2: Computations per kWh over time. Extracted from (Koomey et al. 2009).

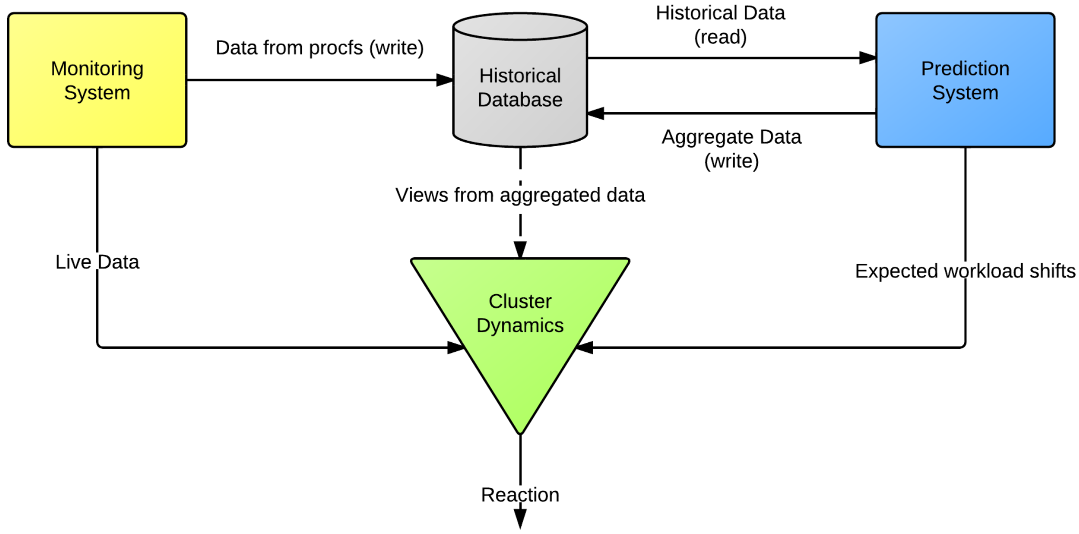

Figure 1.3: Monitoring system provides data to historical database, consumed by prediction system. Both systems feed a decision-making system, responsible for acting in the cluster dynamics.

manufacturing, just to name a few. Learn the value of it, generated not only by customers withal during the whole production process, is fundamental for surviving in any competitive market.

As the importance of information grows, the storage demand for it in an organized, meaningful and accessible way, emphasizing the relevant aspects of reality, grows on the same basis. Database systems are a strategic tool in this field, as businesses rely upon the information stored in it for a myriad of purposes. Together with this development, the energy consumption of database systems evolved from an energy budget problem to a global environmental question, which affects all of us; a typical database server consumes about 60% of its peak energy consumption when idle (Tsirogiannis et al. 2010). The need for a solution addressing significant energy savings while not compromising data access performance and quality is urgent.

1.2 Objective

This work provides a monitoring distributed framework for WattDB. Its main goal is to provide accurate, relevant and timely data, incurring low overhead in the nodes and in the network. Monitoring the cluster can improve the centralized management to execute actions across the cluster. This monitoring system must collect data — system usage statistics and events of interest within the database — from every node (slaves and master), store it in a historical internal database to be aggregated or serving it directly for on time reactions, as can be seen in Figure 1.3. The values measured from these nodes can be used by a decision-making engine to make an informed decision about what actions to take to control the cluster dynamics. Also, the study of past query execution in the database can help determining future workloads, node activation and deactivation polices and supporting energy proportional behavior. New monitored data is used to always check if the previously actions executed by the master had the desired impact on the cluster, or if new actions need to be taken.

1.3 Organization of the Text

The remainder of this work is organized as follows: Chapter 2 discusses the state of the art of the research in distributed monitoring, and also defines the basic concepts used in this work. Chapter 3 discusses what data is relevant to be collected, where to collect it, how to consolidate this data and how to expose it to others components inside the database. Chapter 4 shows how the framework is implemented, the development environment and its modules. Chapter 5 presents the results achieved by the monitoring system with respect to all the tests performed. Concluding remarks and future challenges are outlined in Chapter 6.

2 THE CONCEPTS OF DISTRIBUTED MONITORING

Despite the huge data centers that nowadays operate all kind of web services, cluster node management was until recently based only in archaic methods like the "ping" command — where a remote node should reply to a received packet, or it would be considered inactive. It is crucial to review the recent frameworks proposed to monitor clusters, as well to define some basic concepts about monitoring. As this work is a component of a distributed database, contextualize the monitoring concepts inside WattDB is fundamental.

2.1 Monitoring

Monitoring is the observation and the regular record of an application activity. It is a permanent data collection service executing over many different aspects of the monitored application or system. Monitoring is to check the progress of the running applications, that is, a systematic observation with well defined goals. As seen in (Sottile e Minnich 2002), monitoring contributes to management. The data read from sensors can be used to make informed decisions about what actions to take — by a human or by a reaction software. After these actions were executed, the sensors are probed again and the data collected can be analyzed to check if the previously actions improved the state of the system or if other actions must be taken.

A monitoring which executes in parallel with the monitored application is an *intrusive monitor*, sharing the computational power which would be used only by the application. This demands the monitor to be lightweight, in order to interfere the minimum as possible in the global system. A monitoring application will keep track of the changes caused by any other running software in the system or even other systems which may interact with the monitored one. This data is then collected and should be served, as a report for administrators or to the main application use, staying aware about the system state. For example, in a medical heart rate controller software would be interesting monitor every action that the controller may take — increasing or decreasing the heart rate. If other medical devices interact with the controller, is necessary to record their actions also, so any change in the heart rate software behavior is registered. If suddenly the heart rate in the patience starts decreasing, a report over the monitored data can help the doctor to understand the situation and correct the problem as soon as possible.

The first perturbation caused by the monitoring process is the execution of its own extra instruction. Nevertheless, a second wave of perturbations can derive from no code optimization between the running application and the monitoring software and operation system's overhead (Malony et al. 1992). How much intrusion will be tolerated depends on the nature of the running application and the desired monitoring precision. To determine this trade-off, it is important to understand that an intrusive monitor will interfere arbitrarily in the execution of the monitored application. This change may:

1. generate incorrect results;
2. created (or disguise) *deadlock* situations, when the *thread's* event order is affected;
3. increase drastically the executing time of the monitored application;
4. make the understanding process of a distributed application even harder.

It is a project challenge to understand the application's characteristics and design a monitoring software that can cooperate with it, helping administrators and other softwares to make the best informed decision.

Monitoring can also execute as a service inside the applications, where it assumes a logging task. In this case, the objective is to collect data generated by the own software — how many times a function executed, how many locks over a specific resource were demanded — and provide this data to an administrator or internal decision-making system, which can make use of it inside the application. In WattDB, the cluster and the database system are being monitored, using a mix of the both approaches.

2.2 Distributed Monitoring

Monitoring a cluster means to execute the monitoring tasks over (possibly) all machines that are part of the cluster. The data collection process can represent the individual state of every single machine (intra-monitoring) or the whole cluster as a single node (inter-monitoring). Intra-monitoring is useful to schedule jobs inside a data center, when inter-monitoring is necessary to address the problem of capacity planning for the provision of cloud data centers (Costa et al. 2010). The monitoring framework presented in this text focus in intra-monitoring, which is the scenario where the WattDB operates.

Distributed and parallel programming are know as more complex then sequential programming. Many independent tasks, communication and synchronization are some reasons for that. This increase in complexity is negatively translated in the quality of distributed applications. Distributed monitoring is normally performed to identify *software* and *hardware* problems that may occur. Software problems, like performance bottlenecks or wrong results in the applications are identified tracing the distributed applications. A *cluster administrator* uses the collected information to analyze and possible fix the problem. It is possible to also have another software acting to resolve unstable situations; this software will consume the monitored data and try to fix the bottleneck — rescheduling tasks or jobs, killing process, etc. In a distributed system, since we have more machines, the chance to have hardware problems (bad memory sectors, bad hard drive disk, etc.) also grows. It is too costly (in terms of money and time) to a human administrate hardware failures in today's data-centers — which can achieved easily more than 10000 machines. The need for a tool that can probe hardware is essential.

There is no silver bullet in terms of distributed monitoring applications. Depending what kind of information is needed, a different tool must be deployed. Different informations are the network linking nodes, execution and data flows, energy, temperature, humidity, etc., for example. *Multiple monitors* increase the system's complexity due the independence of each tool in:

1. *Collecting* the data,

2. *Storing* this data after being collected and

3. *Consume* it later by another software or an administrator.

Normally different monitoring applications have different collecting methods. There is no standard way to gather all sorts of data at a glance. Applications must be traced, infrastructure must be sensored, hardware must be probed. This plurality results in many different monitoring solutions. Store the collected data in binary files or in XML files is useful when another software will consume the data, however humans will need a visualization tool to analyze that data. Analyzing many different information sources naturally increase the complexity of monitoring solutions.

2.3 Database Monitoring

Database monitoring refers to the problem of observing various parameters related to the execution of a query in a database system. It contributes to: detecting performance problems, bottlenecks, or abnormal conditions; auditing usage of system resources and tracking trends; and deciding resource allocation, capacity planning and adaptive query processing techniques, collecting comparable data on a regular timely basis. Monitoring is desirable in a variety of scenarios. When a database system is overloaded, a database administrator (DBA) can use dynamic query monitoring information to decide which queries are consuming resources and which queries are close to completion. In a distributed database, monitoring can collect information like node current state, activity metrics since the node instance started, how the nodes are configured and which of them are currently running.

Provide accurate, relevant and timely data incurring low overhead in the nodes is the main objective of this work. The performance of the monitoring system has two main aspects: the impact on the monitored system (low overhead) and the efficiency of the monitoring (accuracy and relevance). Ideally the monitoring is a tiny fraction on the applications trace; it must be lightweight and imply low impact in the nodes. The second aspect demands the monitor to execute all monitoring tasks in a timely manner, within the desired period and with respect to the wanted components. The monitoring system must scale to handle an arbitrarily large number of nodes, without compromising its fundamental properties.

2.4 The WattDB Project

The WattDB project aims to reproduce the energy-proportional behavior on a database cluster of wimpy (Andersen et al. 2009) shared nothing computing nodes (also called *Amdahl blades* (Szalay et al. 2010)), replacing the one power DB server machine. The cluster is centered in one single node, which can attach further nodes while handling DB processing uninterrupted. In this manner, the cluster can scale up to n nodes, being able to smoothly grow and shrink dynamically. Cluster nodes can be seen as processors cores — increasing system utilization proportional to power consumption as needed. Using a single computing node we would never come close to the ideal characteristics of energy proportionality, as seen in (Härder et al. 2011). Dynamically powering nodes impacts all layers of database software, like storage mapping, query processing and cluster coordination — being the last one the long term objective of this work. The ideal and the observed behaviors are ilustrated in Figure 2.1.

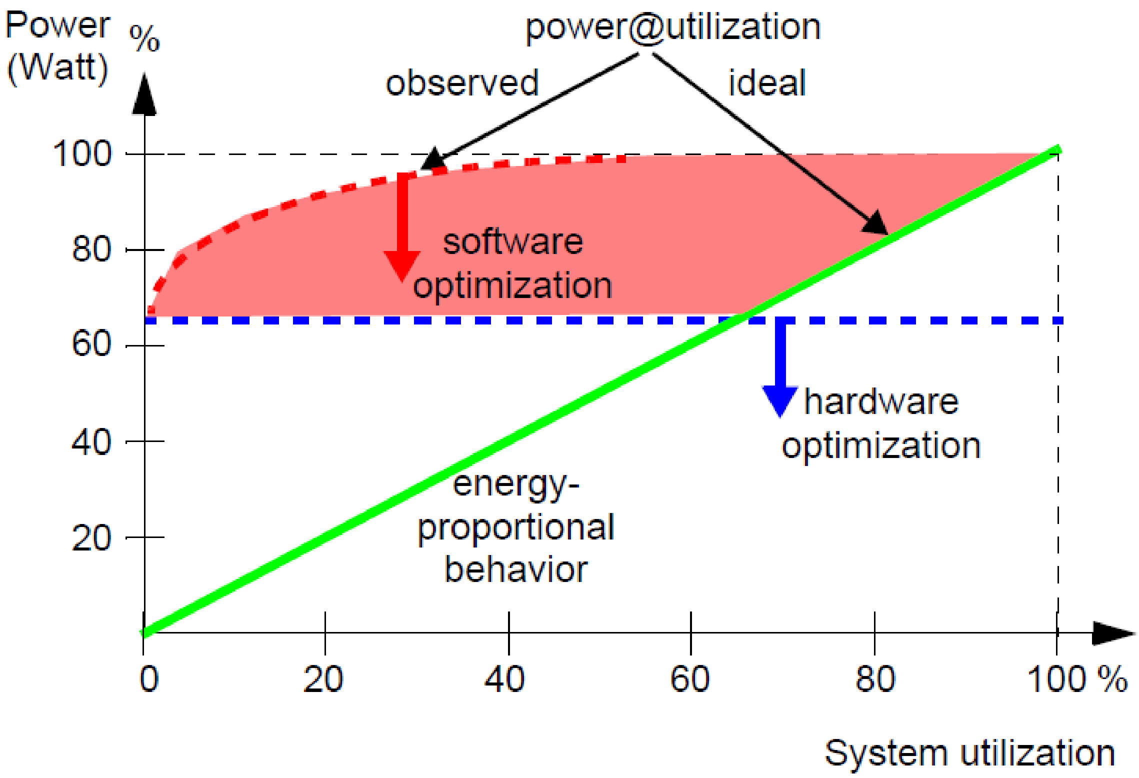

Figure 2.1: Power use over system utilization: single computing node.

The centered, single node (also called master node) is responsible for all coordination tasks (Schall e Hudlet 2011). Beyond housekeeping polices, some tasks need more than centralized control to operate — they need information. Cluster dynamics is the redistribution of responsibilities as an implication two main factors: the first is the number of currently running nodes — or cluster size. If you have an operation over a huge table, it is faster to parallel compute it in as many nodes as possible. If the database does not have enough nodes on, it must activate them. However, there is a second main factor to consider — the activation and deactivation time spans. How long takes for a node become fully available after being reactivated to handle an overload situation? How long should the master wait for a low-utilized node be deactivated and disconnected from the cluster? In some situations (e.g., a batch report execution during the night) the total time of the processing is not important as long as the report is ready in the morning. The master can then focus in energy saving measures, using as few nodes as possible to accomplish the task in time. On the other hand, e-commerce applications need an almost real-time response; the cluster must assure a very low answer to clients' request. Cluster dynamics control requires cluster events observation as well usage data to be collect, analyzed and served. This analysis provides expected workload shifts. With this information the master can take the necessary decisions to support the coordinate measures which maintain the whole system operating in an energetic proportionality way.

2.5 Related Work

While many network administrators rely in primitive measures like the previously cited "ping" command to control and manage the clusters under their supervision, some works have been proposed to deal with this important task in a sophisticated approach.

They aim to evolve the cluster monitoring from an undisciplined set of actions executed only in reaction to some problem to a science which can help avoiding failures in the nodes.

ClusterProbe (Liang et al. 1999) is a Java-based tool. It implements a multiple-protocol communication interface, allowing it to connect to a myriad of external sources. The tool scale up easily due its cascading hierarchical architecture. It returns the currently resource utilization of the nodes, information which can be used to guide the scheduling of new parallel applications or re-organization of the ones currently executing. An interesting feature is the capability to download a new monitoring interface (new code from the server), enabling to change the resources which are being monitored by the client (e.g., if the nodes suffer a hardware update). One main feature of the *ClusterProbe* is the open environment communication, which allow other subsystems and other application to consume its monitored data. This is not needed in WattDB, because it is designed to run totally alone in the cluster. The cascading hierarchy of monitoring domains results in monitoring proxies, each one being considered a special agent that can manage other agents within its domain, filtering monitored data or deploying new monitoring interfaces.

Supermon system (Sottile e Minnich 2002) defends the idea that for Tera-scale and beyond clusters basic tools (as ping) cannot handle the job in terms of the number of machines and its workloads. *Supermon* aims periodic monitoring of cluster nodes, making better use of the available data, a strategy similar to our own. A *mon* server on every node serves monitoring data on a TCP port. A *Supermon* server collects this data by serially connecting to each *mon* server. The system does not keep history metrics, making time-series measurement (as CPU load) difficult. This is a fundamental requirements for WattDB system, as many workloads in a database system repeat during a specific time frame (weekly, monthly, etc.) and is impossible to forecast future workloads without studying the past ones. The *mon* agents are lightweight and support extremely rapid polling rates, due the implementation of a kernel module in Linux which inserts the monitored data into the */proc/sys* tree. As described in the next chapter, this work also uses the */proc* file system to collect the desired monitored data, nevertheless it parses the already existing virtual files created by Linux kernel. More than that, we think that a parser for the well-formated */proc* is simple to develop, deploy and maintain than a new kernel module.

Another monitoring tool existent is *Ganglia* (Massie et al. 2003), which is a scalable distributed monitoring system designed for high performance computing systems such as widely distributed grids. It uses uses a multicast protocol to achieve efficient real-time monitoring. This work is focused on the wide-area clusters, differently from WattDB today, which does not have this as main feature. *Ganglia* is able to monitor several computers inside the cluster and, for each of them, the level of CPU usage, memory and disk I/O. However, it cannot collect specific application monitoring traces.

A monitoring system for WattDB must attend to the following requisites:

1. *One single process*: each node in the database cluster must run only WattDB process, besides Linux kernel;

2. *Easy deploy*: WattDB currently runs on Linux, but it is designed to be deployed in any Unix distribution with few or no modifications at all. So must be its monitoring system;

3. *Hybrid monitoring*: hardware utilization and database tracing both must be collected, storage and served.

3 DESIGNING A MONITORING SYSTEM FOR WATTDB

This chapter discusses the design of the monitoring system, splited in four topics: data's relevance, data's collection, data's consolidation and information exposure. Architecture issues will be presented and discussed.

3.1 Design Overview

Within WattDB, monitoring can serve as a basic building block for enabling adaptive and autonomic techniques. The master node needs information to control the cluster dynamics — the resizing of the cluster and the time span to do it. Storage can focus in energy efficiency requirements (consolidating data to as few disks as possible, turning off the unused ones) or high-performance needs (distributing data for faster parallel access across the nodes); to achieve that, centralized coordination of storage mapping must be aware of data distribution. Query processing also needs information to determine where to send some specific query plan, based on the current hardware components or software characteristics of the nodes. For example, when the workload is too heavy for the current running nodes process or when the nodes have finished their tasks and then starting to become idle, the master node need to initiate counter measures keep the response times low or save energy, depending on the case. If a table has a high-traffic area, it can be split into several smaller partitions. The monitoring system is a vigilant tool which collects data from the nodes, consolidate them and make them available to different components of the system. Problems detection, usage auditing and precise decision making can be easily achieved. Still, is important define what data is relevant, how it should be collected and consolidate. Is also fundamental define an interface which the monitoring can use to serve the monitored data for other components.

3.2 Relevant Data

In this text, the term “data" is used to describe raw, direct values probed from cluster. Data can be collected from different sources inside the cluster. These sources are classified in: measurable system statistics (counters) and observable events. “Information” is any relevant data that can assist a decision-making process, aggregated or not. Reacting tasks require the ability to filter dynamically counters or to access aggregate maintained values of them when some event or threshold is met. Usually, live statistics inform how the system is behaving in a specifically time frame, while aggregate counters express system's historical performance. If an unexpected workload needs to be processed, counter's data can be used to react accordingly. Over a period of time, fine-grained information can be obtained, stored for analysis of resources bottlenecks and further tuning of the system. Hence, some counters can be interpreted directly as information — relevant data for reacting purposes.

A node must attend some requirements to execute a database operation. WattDB is distributed, having a client-server hierarchy inside the cluster. The communication between the single master (server) to the slaves (clients) is done through an internal network. While the request processing is being carried by the CPU the data received stay in main memory. Since the nodes are shared-nothing, is interesting to have a hard-disk not to save processing results yet to be able to swap data during the execution — if necessary. The node sends the results after completed the operation. Hence, monitoring these components during the execution of operations provides valuable metrics about system's performance. Monitoring tasks require observation of a large number of events in the cluster, however, actions to be taken as a consequence of monitored data are often based on a smaller volume of information that is typically filtered and/or aggregated. Delegating a memory-bound operation to a node can rely just on the free space available, but the memory probing will also measure buffer and cache size, occupancy indexes, etc.

3.2.1 Counters

Basic counters are associated with system usage. These counters include CPU, memory, network and disk usage statistics — amount of free memory space, number of packages over the network, CPU load level, etc. These counters provide information about the current load of node. Monitoring the number of active connections with the server and the number of currently executing queries can also have a straightforward influence in the cluster behavior. Differently from usage statistics, which must be probed from every node and sent to the master before it can react based on that data, active connections and total running queries are values know by the master. It can start adaptive measures before detecting overhead parameters in the nodes — if the number of active connection meets a defined threshold, master may know in advance that the current running nodes will not support that workload. However, in a scenario with few but heavy queries, the master can only react based on the incoming monitoring data from the nodes. Monitor master and slaves in combination is necessary for a full understanding of actual system state.

3.2.2 Events

Monitoring events are also interesting for the system, beyond the counters. They play an important part in decision-making, since they trigger counter measures necessary to keep the cluster shifting performance up and down conform as needed. For example, after some specified time, idle a node can fire an idle node event, reporting to the master its vacant state; the master then decides if the node must remain activate (there is an incoming workload for the node) or not. Soft events (SEV) are related to query execution, such as SQL statement execution begin and end, locks acquire and release by queries over resources, query commit, etc. Hard events (HEV) are cluster events, like no space in hard-disks for swapping, connection lost between master and slave, counters thresholds (e.g., CPU load over 80%, for a single node or in average). SEV usually happen in the master node, due to the centralized query processing. HEV can be trigger in every instance of the cluster — including the master. As in the relation between slave and master counters, monitoring both kinds of events is necessary to provide relevant timely data for cluster central control. They provide different important information. If some kind of maintenance policy needs to be fired and cannot be attached to a semantic event, is possible to create a timer object, artificially triggering it in a regular timely basis.

There are some counters associated to events as well. Query execution start and end time can be useful to design a behavioral patterns of the database regime along the day — which can be extended to weeks, months, and so on. These patterns (aggregated counter values) are used by the master to forecast the incoming workloads — e.g., every last Wednesday of the month between 10h and 18h run employee time table verification. In such case, the employee tables are spread across free nodes (employee from A to D in node one, from E to H in node two, and so on) and the desired script run in parallel in each one of them. The duration of some events can also be relevant to detect overhead in the components. Master node requests (send) a query plan for a node executes, waiting for a response (answer). When the time span between send and receive is too long, the cluster may be experiencing a network or node over-utilization.

3.3 Collecting Data

Events can be directly observed during program execution. Event handlers are in close integration between monitoring and server code, attempting to incur low overhead to the system. However, to get system usage statistics we must go beyond the database code. WattDB is currently running exclusively on Linux platform, which implements the useful */proc/* directory — also called *proc* file system (procfs). This file system contains a hierarchy of special files which represent the current state of the system — allowing applications and users to peek into the kernel's view of the system. Inside */proc/* directory one can find a wealth of data detailing the system hardware and any process currently running (RedHat 2007).

3.3.1 The Proc File System

Proc file system is often referred to as a virtual file system. In Linux, all data is stored as files — which, normally, are text or binary files. But the */proc/* directory contains another type of file — virtual files. An effort has been made to keep the information formatted in a consistent and compatible way over time. The files content are constantly updated, fact reflected by the current time and data of them. In addition, files containing similar information are grouped into virtual directories and sub-directories, allowing easy and organized way to look for relevant files. Files containing specific process data are inside a folder */proc/[pid]/*, where [pid] is the current ID of the process.

Collecting data from procfs is fine while WattDB focus only in Linux. Many modern UNIX systems only allow privileged processes to read data about running process. In those operation systems we need a more generic way. LibGTop is a standalone library to get this data, like CPU and memory usage, on those systems where special privileges are needed (Baulig e Poó-Caamaño). GTop library biggest advantage relay on portability across any UNIX distribution; it is the most portable option to acquire system usage data for monitoring purposes in multi-platform applications.

WattDB monitoring system uses procfs instead of GTop library as project decision. The files in *proc* are special and will not be memory cached, so all reads are real and reflect current information from the kernel. Today WattDB does not have the ambition to be portable across all UNIX -distributions, as a proof of concept without commercial intentions in the short run. Thus, using the library would only add an unnecessary layer in the data collecting flow, since in Linux, GTop must parse the virtual file system as well to obtain the data.

```
8       0 sda 76390 38637 4132784 1968910 437711 711810
8       1 sda1 35544 30285 2194578 1325560 59186 105071
8       2 sda2 2 0 4 70 0 0 0 0 0 70 70
8       5 sda5 40795 8352 1937810 639170 341263 606739
8      16 sdb 125 28 1218 280 7 1 64 200 0 480 480
8      17 sdb1 75 25 794 190 7 1 64 200 0 390 390
```

Figure 3.1: Non human-readable procfs file (*/proc/diskstats*).

```
MemTotal:        4055736 kB
MemFree:          306240 kB
Buffers:          155716 kB
Cached:           571276 kB
SwapCached:            0 kB
Active:          2805456 kB
```

Figure 3.2: Human readable procfs file (*/proc/meminfo*).

3.3.2 Procfs Files Parsed

The parser developed attends specifically WattDB monitoring needs. As seen in section 3.2, not all system hardware needs to be probed; hence just some files are relevant for data collection. The main files used are:

- */proc/[pid]/statm*: status of the memory in use by the process WattDB. Some statistics reported by this file are total program size, number of pages shared, of code and of data/stack.

- */proc/[pid]/stat*: contains a variety of status information about the process, like state (running, sleeping, waiting), parent pid, virtual memory size, start time and more.

- */proc/stat*: contains a variety of different statistics about the system since it was last restarted. Mainly used for obtaining CPU processing times.

- */proc/cpuinfo*: identifies the type of processor used by the system, with fields like vendor, model, MHz. It is being used to determine the number of cores (siblings) of the CPU in use.

- */proc/meminfo*: contain a large amount of information about the system RAM usage, not specifically to any process. Total memory installed, free, buffers and cached sizes are some examples.

- */proc/net/dev*: lists the various network devices configured on the system, complete with transmit and receive statistics. Contain the number of inbound and outbound packets and more.

- */proc/diskstats*: extensive disk statistics to help measure disk activity. Field examples are number of complete reads and writes, number of sectors written, number of I/O currently in progress.

3.3.3 Procfs Parsing Process

Procfs are in many different formats. Some of them are human readable (like */proc/meminfo*), some of them are not (like */proc/stat*). The parsing process must handle these characteristics. Basically, the virtual file is open and loaded to a stream reader

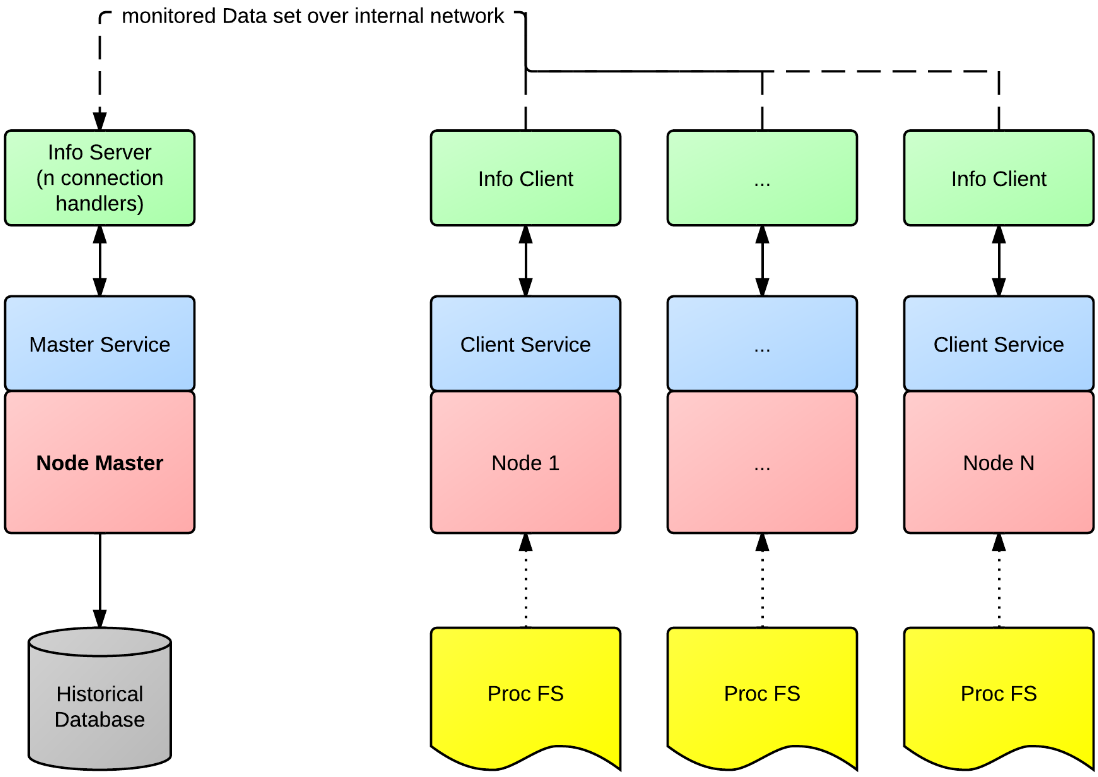

Figure 3.3: Client server WattDB communication architecture.

— see Appendix C.1. Overloads of this method return the first or the n-th line of the file being read for efficiency purposes, as in most of the cases not all file data is needed. The files non-readable for humans normally contain all statistics with respect to some counter grouped in one line — the diskstats file, for example, have all the disk usage metrics for a determined partition per line (Figure 3.1); human readable files have the "*counter: value*" structure, and are normally one per line written in the virtual files (Figure 3.2). In any case, the lines read have to be split in several singles values. A split method (Appendix C.2) receives one string (one line of the file), one delimiter to use as limit for each sub-string and a vector to return the sub-strings computed. E.g., with a delimiter ";" the split of the string "a;b;c" would result in sub-strings "a", "b" and "c". Some data cleaning is executed, removing white spaces and colon punctuation marks.

3.3.4 Communication Architecture

Once the correct files are located and their data is collected, nodes need to send it to master, responsible for collecting the whole cluster statistics, use and store them. The monitoring service for WattDB runs in every node, because the database system must be homogeneously monitored. But just the master can take advantage from the monitored data, since the centralized control approach. Figure 3.3 shows how the communication is architected inside WattDB. Following the client-server model, the cluster has n nodes connected to a single master. Every slave node runs a client service, responsible for executing query physical plans over set of database data received from the master. The client service has an interface called Info Client for sending and receiving data (not exclusive to monitoring, query logical/physical plans, tables, and other database structures too) which communicates with the master node. The master has a similar interface to handle the data exchange, running over the master service. To avoid producer-consumer problems, the Info Server handle each client on a separate thread, parallel handling requests and responses over the network. The slave collects the usage statistics from its procfs and then serializes this data, which is asynchronously sent through the master. The client node keeps sending monitored data to the server without being requested; if for some reason this data cannot be handled in the server, the packages are discarded.

Data must be collected on a regular timely basis. If the gathering interval is too big — performed infrequently — monitoring can lose valuable information about the system usage. If data is gathered too frequently, the monitoring service can impose significant overhead on the server — in terms of processing and storage. Today monitoring system peeks procfs every 10s; the ideal resolution within WattDB stills an open question, to be solved in future work.

3.4 Consolidating Data

Generally, collected data is not ready to be consumed — directly procfs statistics are not always useful information. Data consolidation is a process that takes various data sources and compiles them into another data structure, with more semantic value than the original. If from a frame of system active metrics the master node can take immediate actions, it would be better to rely in a solid information background to make better decisions. Auditing requires consolidation over the collected data; the system needs some statistical study to be able find points outside the curve, unusual performance metrics.

3.4.1 Historical Internal Database

The ability to keep state or history information about probes is provided thought a historical internal database (HDB). Collections of monitored objects are stored to keep history of system state. This storage could be done in CSV or XML files, to be analyzed later by some specific tool. But that would add unnecessary work to WattDB, since we can use its own relational engine to store and process all collected counters. The basic storage architecture between the monitoring system, the prediction system and the historical internal database can be seen in Figure 3.4. While not fully operational, WattDB will take advantage of SQLite (Hipp 2011) to support the historical database. SQLite is self-contained (require minimal support from external libraries or operating system), server less (the process that wants to access the database reads and writes directly from the database files on disk, there is no separate server process), zero-configuration (there is no server to be started, stopped or configured), transactional (all changes and queries appear to be ACID [1]) database engine. These properties make it easy to use inside the WattDB code, because there is no need for access drivers and no server process to concur for system resources. The historical database has five main tables: a Time table, which holds the time stamps of each measurement. It has seconds' precision. The table Node lists the current nodes in the cluster, using the same node ID used inside the application. Usage tables hold the counter values, measured from the relevant system components — as CPU, memory, disk and network. Each of these tables has a foreign key to Time and Node tables. For a specific node at a specific time there must be only one row of usage values. Figure 3.5 shows the relaton between the tables; more details about the structure of each table are described in Appendix B.

[1] Atomicity, consistency, isolation, durability.

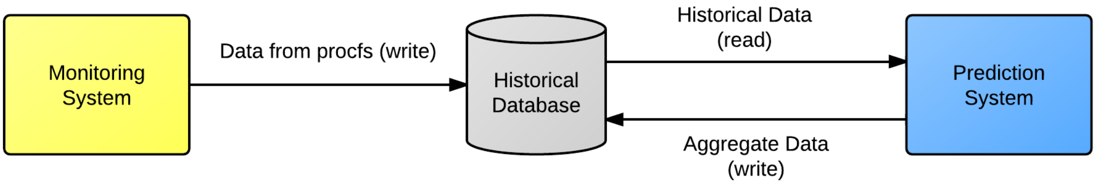

Figure 3.4: Monitoring saves data in a historical database, used by the prediction system to consolidate usage counters and forecast future workloads.

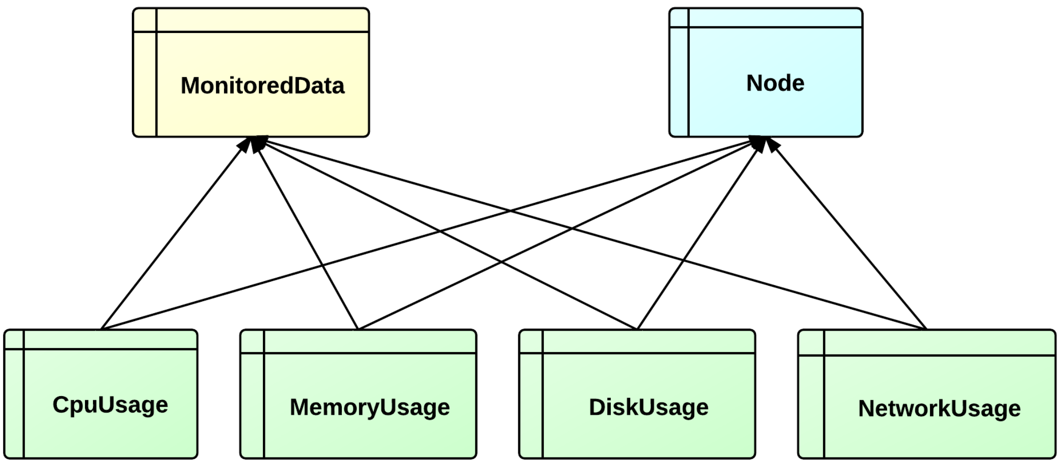

Figure 3.5: Measurements are time-stamp and node-value unique (e.g., usage values for node x at time y).

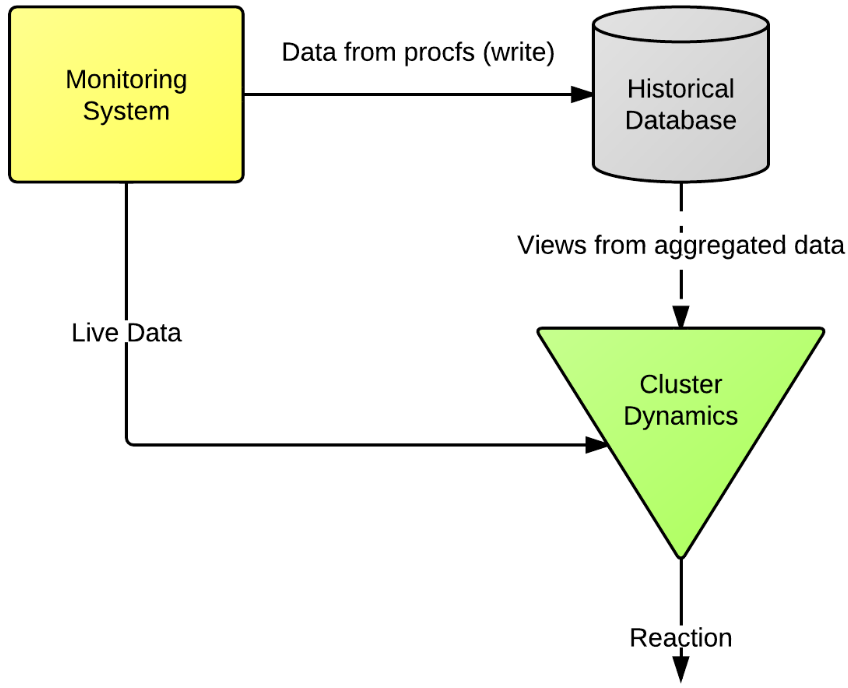

Figure 3.6: Cluster dynamics can consume live data directly from monitoring system and aggregated data from historical database.

3.4.2 The Aggregation Price

Consolidating data can be an expensive operation for the database, depending on the amount of data to process. The precision of any statistic study regarding the activity metrics of the system increase with the volume of data monitored. The system cannot delegate resources for data aggregation too frequently; otherwise query processing would be affected. Data consolidation must run in a non-priority schedule, possibly at night or weekends, and must reuse previously results, which are also saved in the historical database. This way, clients of the database (e.g. query processing or cluster dynamics control) can query it without imposing too much overhead — just a selection operation in the already processed information. The most important consumer of the historical database is a prediction system, responsible for aggregating the data stored. The prediction system in detail is beyond the scope of this text.

3.5 Exposing Information

The distributed architecture of WattDB implies a more complicated database administration, fact aggravated by the dynamically powering of the nodes. The main objective of the monitoring system is to provide relevant and accurate data to support the cluster control. To achieve that, the data collected and consolidated must be available to be consumed. As seen in Section 3.3.1, information can be supplied directly or indirectly (aggregated) from the monitored data. Query processing can use monitored information to decide to which node sends some query plan to be executed or which queries are near of completion. Cluster coordination can use threshold information or idle times to decide if more nodes should be turned on or if there is any node that can be deactivated.

The use of monitoring information happens in the master node. The monitoring service in the master is responsible for not only storing the data collected in the HDB, but also to make it available. The monitoring manager (MM) holds objects for each monitored component. These components are updated in the same frequency as the monitored data is produced. Every time new packages arrive through the Info Server; CPU, memory, disk and network usage statistics are constantly refreshed in the monitoring manager. Components of the system read information about the desired node from the MM. This way, live data about the cluster usage is always available in the master node (Figure 3.6).

It can be useful to compare the current cluster situation to a specific previously situation or to the average of some of them. To do so, the master service will need the historical information in the HDB, which keeps state of the collected probes. Views can be defined over main attributes, facilitating the query for information. Provide access to relevant data from the HDB still an open problem, to be solved as future work.

4 IMPLEMENTATION OF THE MONITORING MANAGER

This chapter discusses the implementation of the monitoring system. An architecture diagram will be presented, and the classes and objects will be detailed.

4.1 Implementation Overview

Monitoring system is architected to work as a service in the nodes — master and slaves. It runs in parallel to other services, like query processing, storage mapping and housekeeping activities in the master and parallel to the execution of operation in the slave nodes. The main class, Monitoring Manager, is responsible for encapsulating the data gathering and data cleaning. The manager holds four different kinds of objects to execute those tasks — hardware and software system information and a node information gathering. A historical database manager is responsible for building the queries to insert data in the HDB. An overview of WattDB architecture can be seen in Figure 4.1.

4.2 Gathering the Data

NodeInfoGather will gather the data and load it into process and system usage objects. It gathers this data implementing any library for system usage statistic collecting compatible with the operating system which WattDB is currently running on. Today, it implements it's own parser (as described in Section 3.3.2), *ProcInfoParser*, which will meet the parsing needs of the monitoring system while the project is running on Linux. If the database has to migrate to other UNIX distribution which does not implement procfs, another library for obtaining system usage statistics, replacing the current *ProcInfoParser*, has to be used. This change is transparent for the monitoring system, since the details about the implementation of the parsing process are encapsulated by the *NodeInfoGather* object — no matter how its collecting process works underneath. *ProcInfoParser* returns a simple string vector containing the values read from the virtual files. These values are populated in the appropriate objects. The master node has to handle n slave nodes. The slave nodes must send the monitored counters already parsed into useful data types (integer or float, in most of the cases) to save master's processing time. *NodeInfoGather* does that, converting the values distributed across the nodes.

4.3 Hardware System Information

The *HardwareSystemInfo* class is not currently in use, as WattDB is a hardware-homogeneous cluster — every single node has the same hardware class, maker and model.

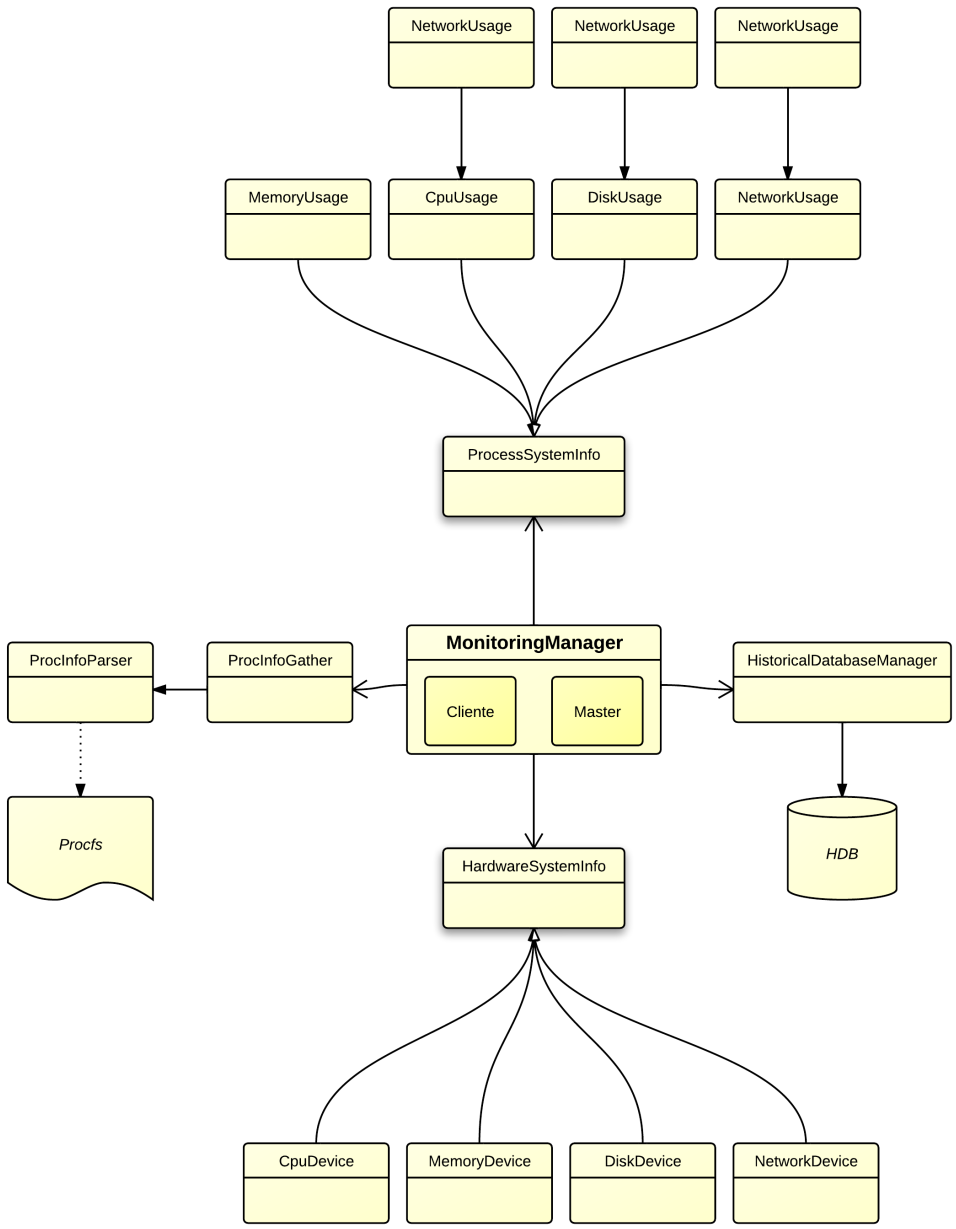

Figure 4.1: The monitoring system architecture.

The class purpose, however, is to give even more detailed information for the decision-making entities — not only in the cluster dynamics, but also queries processing. In a scenario with heterogeneous devices (nodes with different hardware configurations), the decision to which node send some specifically query plan can benefit, not only from usage statistics, but also from the available hardware information. E.g., send CPU bound operations to a node with a faster processor or send memory bound operation to machines with more installed memory can reduce response time. High parallelizable operations can benefit from GPUs, as seen in (Reus 2012). Still, since not all operations are GPU bound, it is not necessary to add graphic processing hardware in every node. As a future work, the implementation of a node value index (NVI) can summarize the machine's hardware characteristics, with higher values for nodes with faster CPU, larger memory and smaller I/O access times or containing singular components as GPUs or SSD drives. This index is useful in a heterogeneous cluster, even though have no meaning when the computers have the same components.

4.4 Process System Information

The word "system" in this context refers to Linux kernel and WattDB process running together in a node. The *ProcessSystemInfo* class consolidates the usage objects. It holds one instance of each component monitored: CPU, memory, disk, network. WattDB is the only process running, besides the kernel. Thus, is no problem for some of these components reflect global system statistics — like disk (reads and writes completed), network (number of packages received and sent) and CPU (the amount of time that the system spent in user mode or idle).

4.4.1 Memory

Show the status of the memory in use by the process. *MemoryUsage* object specifies WattDB process measures, but also include system information. This is useful because the difference between the database and operating system buffer policies. The first eight members refer specifically to WattDB process; members from nine to twelve refer to whole system, not only WattDB process.

1. *Size*: total number of pages of memory (total program size in kilobytes).
2. *Rss*: resident set size.
3. *Share*: number of pages of shared memory
4. *Text*: number of pages that are code.
5. *Data*: number of pages of data plus stack.
6. *Vsize*: virtual memory size in bytes.
7. *Rss*: resident set number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.
8. *Rsslim*: current limit (in bytes) of the rss of the process; usually 2,147,483,647.
9. *MemTotal*: total amount of memory available in the system.

10. *MemFree*: contains the amount of memory available on the heap. This field is used as a reference for allocation amount.

11. *MemUsed*: the difference between MemTotal and MemFree.

12. *MemBuffers*: the amount of memory used by system buffers.

13. *MemCached*: the amount of memory being cached.

4.4.2 Disk

The disk is being measured through its physical *Partitions* today. This means that first the monitoring system maps the current *Partitions* of the installed node hard disk (from */proc/*Partitions*2*), and then start gathering the disk usage statistics for those *Partitions* — the *DiskUsage* class is composed by *Partitions*. This is monitoring is just focused in hardware use metrics; in future, the monitoring system should collect partition storing information with respect to storage mapping. All fields are cumulative since boot, except “I/O currently in progress”.

1. *Major*: the major number of the divide with this partition.

2. *Minor*: the minor number of the device with this partition. This serves to separate the *Partitions* into different physical devices and relates to the number at the end of the name of the partition.

3. *Blocks*: lists the number of physical disk blocks contained in a particular partition.

4. *Name*: the name of the partition.

5. *ReadsCompleted*: this is the total number of reads completed successfully.

6. *ReadsMerged*: reads and writes which are adjacent to each other may be merged for efficiency. Thus, two 4K reads may become one 8K read before it is ultimately handed to the disk, and so it will be counted (and queued).

7. *WritesMerged*: see ReadsMerged.

8. *SectorsRead*: the total number of sectors read successfully.

9. *MillisecondsReading*: this is the total number of milliseconds spent by all reads[1].

10. *WritesCompleted*: the total number of writes completed successfully.

11. *SectorsWritten*: the total number of sectors written successfully.

12. *MillisecondsWritting*: the total number of milliseconds spent by all writes.

13. *IosInProgress*: The only field that should go to zero. Incremented as requests are given to appropriate struct request_queue and decremented as they finish.

14. *MillisecondsSpentInIO*: this field increases so long as field IOs currently in progress is nonzero.

[1]Measured from __make_request() to end_that_request_last().

15. *WeightedMillisecondsDoingIO*: this field is incremented at each I/O start, I/O completion, I/O merge, or read of these stats by the number of I/Os in progress (member 13) times the number of milliseconds spent doing I/O since the last update of this field. This can provide an easy measure of both I/O completion time and the backlog that may be accumulating.

4.4.3 CPU

CPU times are being monitored for the whole system. User and nice members, however, represent the WattDB processing times, as it is the unique process running. The statistics are measured for each core of the processor; CpuUsage class maintains a list of *CpuCore* objects. There is one more *CpuCore* object, with the total times for processor (sum of all cores). The fields correspond to the amount of time, measured in USER_HZ[2] that the system spent in:

1. *User*: user mode.
2. *Nice*: user mode with low priority.
3. *Sysmode*: system mode.
4. *Idle*: idle task.
5. *Iowait*: waiting for I/O to complete.
6. *Irq*: time servicing interrupts.
7. *Softirq*: time serving softirqs.
8. *Steal*: stolen time, which is the time spent in other operating systems when running in a virtualized environment.
9. *Guest*: time spent running a virtual CPU for guest operating systems under the control of the Linux kernel.
10. *CoreId*: the number of the core being measure in the cluster.

4.4.4 Network

Network received and transmitted packages statistics are measure from the network interfaces of the current system. *NetworkUsage* is composed by *NetInterfaces*. A network interface object has the name of the interface (local host, Ethernet, infrared or wireless), a *Receive* and *Transmit* objects (which share some common members) which the counters values. *NodeInfoGather* class creates the usage objects to populate the *ProcessSystemInfo* class; the creation of a *NetworkUsage* object as example can be seen in Appendix C.3.

1. *Receive*: the network received statistics.
2. *Transmit*: the network transmitted statistics.
3. *InterfaceName*: the name of the interface.

[2] 1/100ths of a second on most architectures, use sysconf(_SC_CLK_TCK) to obtain the right value

4.5 Monitoring Manager

Once the usage statistics are loaded from the node information gather, they are packed into a serialized object and sent to the master using boost serialization method (Ramey 2004). Monitoring manager running in the slave nodes can send the *ProcessSystemInfo* class, with the usage objects. However, to consolidate the data collected, the monitoring system on the server side must insert these objects data into the historical database. With the purpose of not overhead the master node, these insert statements are built (with the data from *ProcessSystemInfo*) in every slave; the nodes have the responsibility of sending valid insert SQL statements to the master. *MonitoredData* class works as a container for that, having a string member for each necessary table of HDB. This class is then serialized and sent, also using boost serialization methods. Currently, monitoring system does not send *ProcessSystemInfo*, as there is no consumer for the data; *MonitoredData* is being sent to test the historical database. As mentioned in Section 3.3.4, the slaves send data to the master asynchronously. Once the client service is started, the monitoring service also starts to run, collecting data since the node became active. As soon as the master receives its first package of data, it can save it in the historical database.

5 MONITORING SYSTEM RESULTS

This work aimed three main objectives: provide relevant, accurate data and incurring low overhead to the system. In this chapter, the results achieved by the monitoring system will checked against this objectives and discussed.

5.1 Accuracy and Relevance

The accuracy objective was achieved: the data being collect reflect real and current usage metrics from the system. *Procfs* plays a big part on this, providing an easy to use and constantly updated file system for providing the information needed. It has a wide quantity of files with several fields containing information about process, kernel and hardware. We are sure that the monitoring can be severely extent to a variety of counter and configuration info; We believe, however, the most interesting components for current stage of the database were measured. It is hard to define how relevant the data monitored is, because there is no currently working system making use of it. Once the WattDB starts taking advantage of the monitored data to benefit storage mapping, query processing, buffer policies, cluster dynamics and other database functions, the monitoring system can be tuned to attend the future cluster needs.

5.2 Overhead

To validate if the proposed monitoring system incurs low overhead compared to the original WattDB, several tests were executed over both databases – the original one and the monitored one. The test results were analyzed and compared.

WattDB Benchmark Tool Overview

The testing process used the existing benchmark tool for WattDB. The tool executes four basic database pre-defined operations – select, join, insert and delete. It executes these operations over an artificially created database – Cinema. The Cinema database has the following tables:

1. *Studio*: with id and name columns;
2. *Producer*: with id, first name and surname columns;
3. *Actor*: with id, first name and surname columns;
4. *Movie*: with id, studio, producer, title, release and genre columns;

5. *Role*: with id, movie id, actor id and name.

Some not exhaustive examples of the pre-defined operations are:

1.
```
SELECT  Title, Genre, Release
FROM    MOVIE;
```

2.
```
SELECT  s.Name, p.First Name, p.Surname,
        m.Title, m.Genre, m.Release
FROM    MOVIE m, STUDIO s, PRODUCER p
WHERE   s.Id=m.Studio AND m.Producer=p.Id;
```

3.
```
INSERT INTO MOVIE
(Id, Studio, Producer, Title, Release, Genre)
VALUES
(1, Warner Bros, Spielberg, ET, 1982, Adventure);
```

4.
```
DELETE FROM ACTOR WHERE Id=100;
```

In a nutshell, the benchmark tool works like this: each of the basic four operations has an *executing ratio*, which determines the percentage of that operation in relation to the benchmark. This way, we can test just one operation (setting it ratio to 1) as we can define a mix of operations (e.g., 0.5 selects, 0.3 joins, 0.2 inserts and 0 deletes). The benchmark tool uses an extensive list of values generated by a *RandomDataGenerator* to populate the test database and create operations over this data. Also, it chooses randomly from a collection of pre-defined queries which ones to execute. The precise definition of the benchmark tool is beyond the scope of this text.

Experiments Configuration

This section describes the hardware, operation system, compiler and the benchmark configurations used in the experiments.

Hardware

The CPU used is an Intel Xeon CPU 5130@2.00GHz, 4 cores and 4096KB of L2 cache. The machine has 3960MB of memory, and no swap space is defined.

Operation system, IDE and compiler

Ubuntu 10.04 LTS – Lucid Lynx. The monitoring system for WattDB was developed under Eclipse C/C++ Development Tools version 6.0.2 and compiled in CC Ubuntu/Linage 4.6.3-1Ubuntu5.

Benchmark configuration

Both databases were benchmarked with the configurations showed in Table 5.1.

Select, join and insert operations were single executed from Test 1 to Test 9. Delete operations were not executed solo because they are basically inserting operations. Tests 10 to 12 execute all operations, trying to simulate a real database workload, despite the identical ratios. The database system and the benchmark tool run on the same single

	Select	Join	Insert	Delete	Clients	Transactions
Test 1	1	0	0	0	1	10000
Test 2	1	0	0	0	2	10000
Test 3	1	0	0	0	4	10000
Test 4	0	1	0	0	1	10000
Test 5	0	1	0	0	2	10000
Test 6	0	1	0	0	4	10000
Test 7	0	0	1	0	1	5000
Test 8	0	0	1	0	2	5000
Test 9	0	0	1	0	4	5000
Test 10	0.25	0.25	0.25	0.25	1	10000
Test 11	0.25	0.25	0.25	0.25	2	10000
Test 12	0.25	0.25	0.25	0.25	4	10000

Table 5.1: The transactions configurations for testing the monitoring system.

machine. Executions with more than one client were simulated by the benchmark tool, which runs each client in a different thread. All tests were performed with 1, 2 and 4 clients. The benchmark tool executed 10000 transactions in the select, join and in the mixed tests (Test 1 to 6 and 10 to 12).

For the insert experiment, this number must be reduced to 5000. The database storage used for the experimentations has a fixed size, because the database still in development. When a partition in the database storage is full, it throws an *OnPartionFill(partition_number)* exception, when the master node tries to reallocate data from full partitions to any other which still have space. After several insert operations, the database system execution starts to just handle partition full exceptions – until the point that there is absolutely no space anymore. Executing 5000 insert transaction is a reasonable number since even fulfilling some partitions the master node kept handling the incoming operations.

Experiments Results

Here are presented the dataset resulted from the experiments. The test results are grouped by transaction, so we can compare the performance of WattDB with and without the monitoring system. Each configuration was repeated 30 times, and the collected results from each system were paired compared. The confidence interval used to compare the observations was 90%. Here, the term "original database" stands for the WattDB without the monitoring system, and "monitored database" means the WattDB instance running with the monitoring service.

Select

The observations from the tests 1, 2 and 3 show that the configuration with only one client constantly selecting the original database is visually faster than the monitored database, as we can see in Figure 5.1. Select is a cheap operation in terms of query processing; setting up the monitoring manager, the creation of historical database and other basic administrative tasks of the monitor have a greater impact on the system performance. This overhead is not so significant as the number of clients increase, since the own database system have more work to do, handling multiple client connections. Statis-

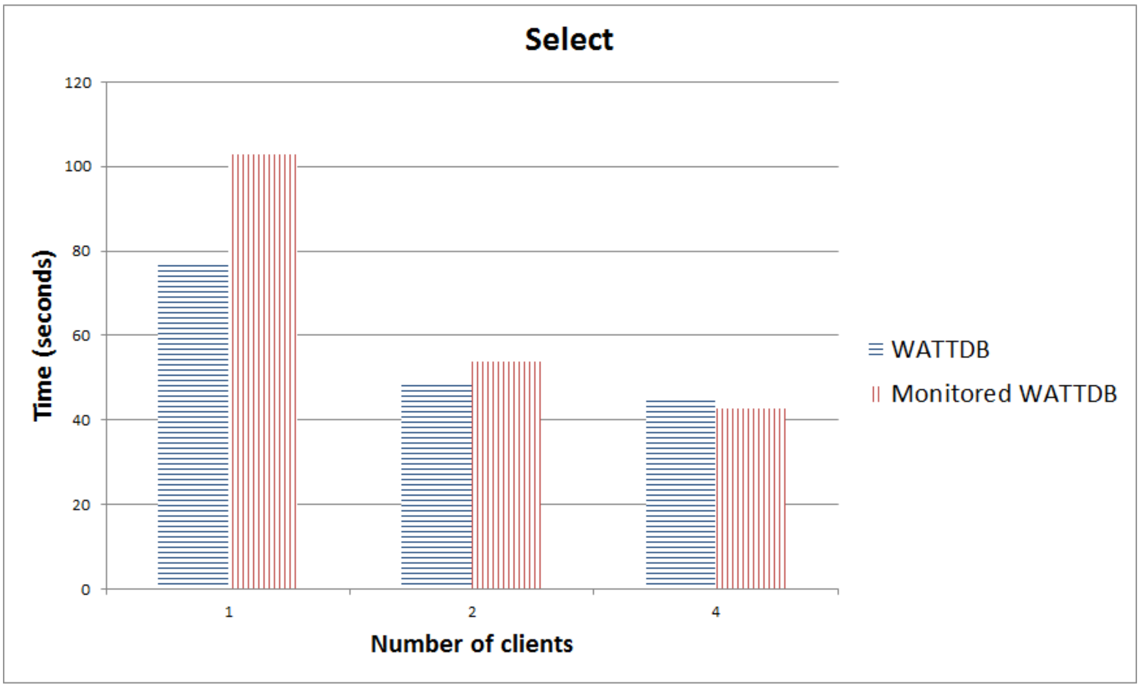

Figure 5.1: Comparison between select tests 1, 2 and 3

tically, the configuration with four clients shows that both systems are not different (Table A.3).

Join

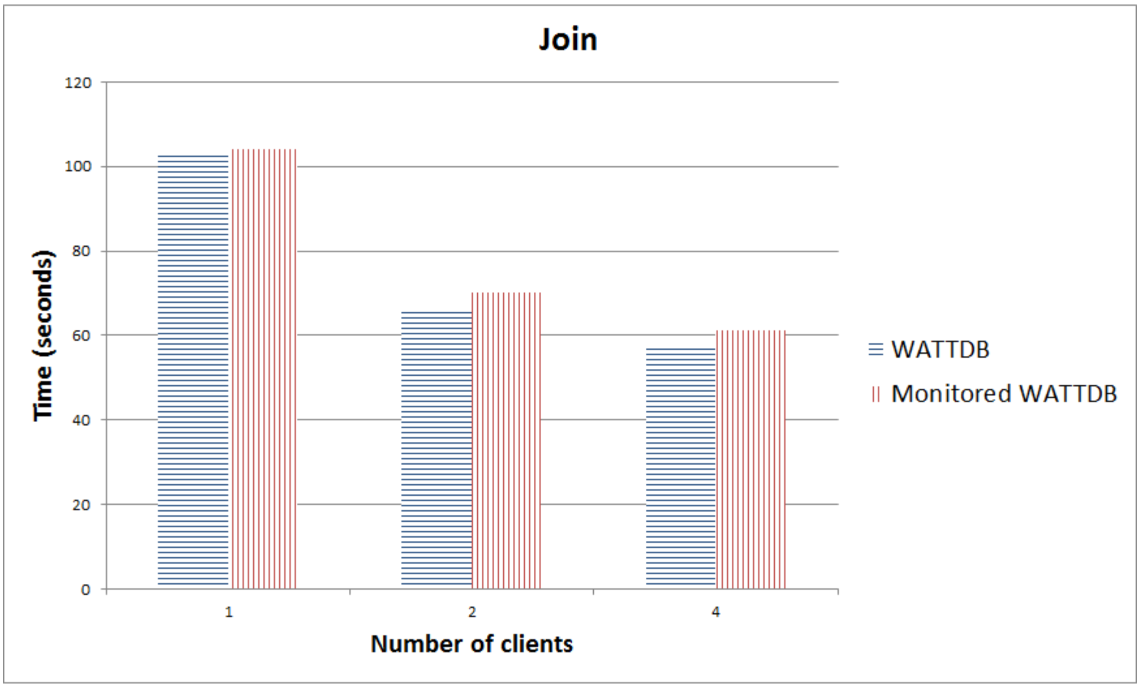

Figure 5.2: Comparison between join tests 4, 5 and 6

Differently from select operation, joining tables requires the database system to generate many intermediate results and operating over them. The monitored database introduce no significant overhead in the scenario with one client. However, the intrusiveness of the monitoring system in the configurations with two and four clients was more significant. When the system is overloaded, turning off services that are not essential (like housekeeping polices and even the monitoring) could increase response times. Figure 5.2 shows the comparison for join transactions. Table A.4 shows no difference for both systems running one client at time, as Table A.5 and Table A.6 conclude that monitoring the database while it executes join operations can increase response times.

Insert

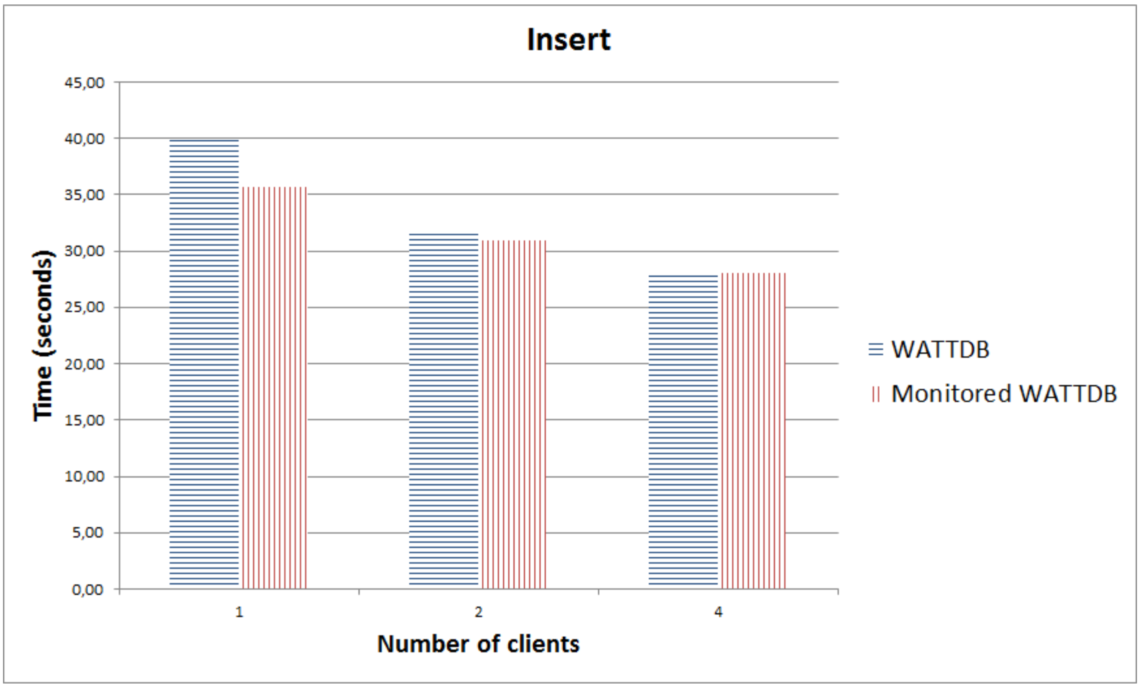

Figure 5.3: Comparison between insert tests 7, 8 and 9

Insert operations have a very specific interpretation. Figure 5.3 show a quite faster monitored database in comparison with the original database. This result derives from the prior discussion about *OnPartionFill*. With only one client inserting into the original database, there is no context switching between clients; it operates at full throttle. This leads to fulfill the storage partitions sooner than the monitored WattDB. The original system starts reallocating data after some partitions are full. In the end, it spends more time handling *OnPartionFill* exceptions than the monitored database. Two and four clients configuration are statistically indifferent (Tables A.8 and A.9), since data is distributively inserted in the available space by the different clients, it takes longer to fill partitions.

All Transactions

The objective for the configurations in tests 10, 11 and 12 is a balanced set of operations, to simulate a more real database workload. The statistical analysis shows an expected overhead in the monitored database compared to the original system for one and four clients configuration (Tables A.10 and A.12), and presents no difference for 2 clients (Table A.11). We conclude that the performance provided this monitoring system, slower than the one found in Supermon system for example, is enough for WattDB needs. In respect to the possible overhead which could be caused, the monitoring service

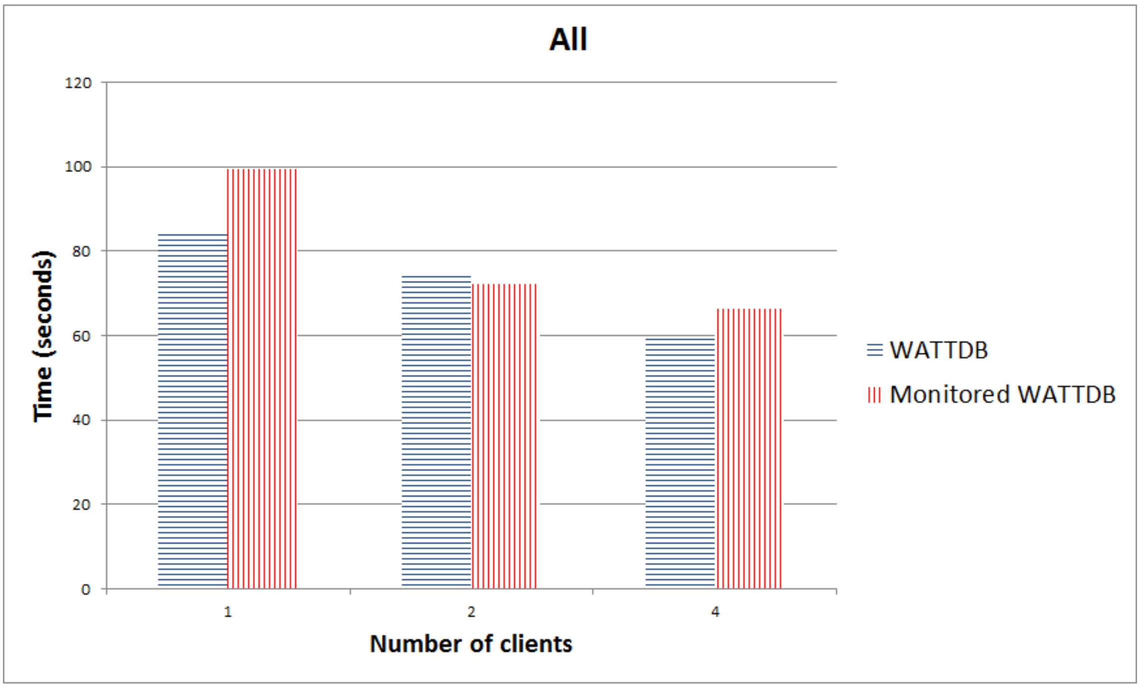

Figure 5.4: Comparison between mixed configuration tests 10, 11 and 12

is a lightweight service to system. In half of the tests executed it performed at least as good as the original database system. As all *procfs* files reside in memory, all operations over them (open, read, parse the data to variables, close) are executed extremely fast. One identified bottleneck today is the historical database for two reasons: first, the master node (through the *InfoServer*) receives the monitored data packages in parallel. However, just the single monitoring manager thread stores that data. And this happens due the second reason: the SQLite. As it implements the database in a file, concurrently opening this file to execute the data operations requires locking it. We could have distributed SQLite databases through the clients and, time to time, synchronize this small sets of monitored data into the master. But SQLite intends to be a temporary solution until the WattDB is fully functional, so this question will have to be answered in the future.

These experimentations were executed on a single machine due architectural reasons — the available cluster at UFRGS have 32bit processors, and WattDB is designed to run on 64bit machines. Split the clients from the master may produce even better results, specially for the client machines.

The static monitoring does not incur an acceptable overhead to the cluster. The amount of monitored events and counters associated with query processing, however, are in function of the number of queries being executed; this means that the more queries the database is handling more monitored data will be generated, known as dynamically monitoring. One second issue is the current bottleneck in storing monitored data. Now, data is distributed collected (over the n wimpy nodes), but centralized stored (in the master). Depending on the number of slave nodes, store the collected data can be a several overhead in the master node. These questions still open, and should be answered in future work.

5.3 Historical Database Size

One big concern about maintaining state of measured counter in a historical database is how much storage space it would cost. Nowadays WattDB uses the SQLite to keep the data, where the INTEGER type consumes 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value (Hipp 2011). For a pessimist calculation, each INTEGER used in the tables will be counter as 8 bytes. The TEXT type of SQLite is saved in a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE). In the same way as the INTEGER type, each character in the string will be expressed in 16 bits (or 2 bytes). For simplicity, every TEXT field will supposedly have 10 characters, resulting in 20 bytes each. The sumary of each table storage space can be seen in Table 5.2:

Table name	Integer fields	Text field	Total integer fields (in bytes)	Total text fields (in bytes)	Total (in bytes)
Time	11	0	88	0	88
Node	2	0	16	0	16
CpuUsage	13	0	104	0	104
DiskUsage	15	1	120	20	140
MemoryUsage	15	1	120	20	140
NetworkUsage	19	1	74 152	20	172
				TOTAL	660

Table 5.2: Amount of data required for each table of the Historical database.

This means that monitored data in a 10 seconds interval would produce 1979MB of data in an year, as can be seen in Table 5.3:

Measure unit	Size (in megabytes)
Minute	0.003
Hour	0.226
Day	5.438
Week	38.06
Year	1 979.516

Table 5.3: Total amount of stored monitored data.

Proximately 2 GB/node in data for currently storage capacities in a year is plenty acceptable. This leaves any the prediction system with plenty of usage statistics to work with, as the same time does not sacrifice database hard disk space. Aggregation will produce more data; nevertheless even doubling the size of the HDB is not a problem for WattDB. Here, we not take in account any compression or aging methods, which would reduce the size of the storage needed.

6 CONCLUSION

This work presented the design and implementation of a Monitoring System for an energy-proportional database cluster. All the relevant concepts were presented, the design was described and the implementation and testing processes were shown.

Initially, the theoretical foundations for this work were presented. Monitoring is a well know discipline with a lot of material to work with. However, distributed monitoring is a research area with work to do. Workstation clusters for distributed computing have become popular only in the last few years, with the emergence of low-cost, powerful workstations and high-speed networking facilities. In order to properly situate this work, a brief overview about the very modern energy problem was presented, together with new enterprise needs — which demands an even bigger amount of natural resources every day. Also, was presented the WattDB project, a distributed database system which aims to balance performance with energy consumption proportionally.

The main aspects which defined the monitoring system architecture were discussed. What kind of data should be collected and from which components; how to collect this data and how to process it once its monitored are questions answered. The communication architecture which connects the slave nodes to the master node inside WattDB system was presented. Also, an internal relational database to store the monitored data was projected and implemented.

The final objective of this work was met. Once the basic aspects were studied and designed, build a functional prototype was the next step. The implementation is then discussed and detailed. As WattDB database system still not fully operational, only a basic testing process was guided. The monitoring system, however, proved to be reliable, attending the requirements that were proposed.

6.1 Future Work

Decision-making components benefit directly from monitoring. The monitoring system is a tool inside the cluster, built to serve these components. It is meaningless to have a monitoring system running without using its collected and consolidated data. As mentioned before, the long term of this work is to provide information for the cluster dynamics operate during the execution path of database operations. Node fluctuations can be controlled by a Rule Engine (RE). The rule engine evaluates event-condition-action rules (ECA rules) as part of the execution code path. An ECA rule takes an action when a particular condition is met, triggered by some event.

The Rule Engine supports a number of different events to be used in the event clause, monitored during the query execution — like commit, compile, rollback, blocked, block released, just to name a few; timers can be set also to trigger the events. The RE then evaluates conditions defined over counters, using logical operators (and, or, greater than, less than, not equal to, equal to, if then etc.) and mathematical operators (addition, subtraction, multiplication and division) to evaluate query executions. Actions can vary from basic usage reports to complex node activation and deactivation in respect to current query workload and forecasted workloads shifts. Rules should be able to be combined with other rules, building complex behavior changes in the cluster with many simple rules. An interface should be provided to facilitate the creating and maintaining of the ECA rules. Some examples are:

Event:	Query.Commit
Condition:	Query.Duration > 100s
Action:	Query.Report(query)
Event:	Timer.Ring
Condition:	Cluster.AvarageCpuLoad > 70% AND
	Query.IncomingQueries > 5 AND
	Cluster.HasAvailableNodes = TRUE
Action:	Cluster.TurnOnNode(nodeNumber7, nodeNumber8)

Table 6.1: Example of ECA Rules.

APPENDIX A CONFIDENCE INTERVALS FOR COMPARED PAIRED OBSERVATIONS

Here are presented the tables of the statistical analysis.

Sample mean	-26
Variance	20
Standard Deviation	4,472136
Confidence interval for mean	-26 ± t(0,816496581)
90% confidence interval	-26 ± 1,3872 = (-27,39;-24,61)

Table A.1: Test 1 — 10000 Select Transactions with 1 client

Sample mean	-5,03333
Variance	3,757471
Standard Deviation	1,93842
Confidence interval for mean	-5,03333 ± t(0,353905)
90% confidence interval	-5,03333 ± 0,6012 = (-5,6346; -4,432)

Table A.2: Test 2 — 10000 Select Transactions with 2 clients

Sample mean	0,4
Variance	6,248276
Standard Deviation	2,499655
Confidence interval for mean	0,4 ± t(0,456373)
90% confidence interval	0,4 ± 0,7753 = (-0,37538; 1,17537)

Table A.3: Test 3 — 10000 Select Transactions with 4 clients

Sample mean	-1,333333
Variance	25,40229885
Standard Deviation	5,04006933
Confidence interval for mean	-1,3333 ± t(0,920186554)
90% confidence interval	-1,3333 ± 1,5633 = (-2,89673; 0,23)

Table A.4: Test 4 — 10000 Join Transactions with 1 client

Sample mean	-4,5
Variance	1,637931034
Standard Deviation	1,279816797
Confidence interval for mean	-4,5 ± t(0,23366151)
90% confidence interval	-4,5 ± 0,3969 = (-48969; -4,103)

Table A.5: Test 5 — 10000 Join Transactions with 2 clients

Sample mean	-3,858333
Variance	5,982083333
Standard Deviation	2,445829784
Confidence interval for mean	-3,858333 ± t(0,4465)
90% confidence interval	-3,8583 ± 0,7586 = (-4,617; -3,099)

Table A.6: Test 6 — 10000 Join Transactions with 4 clients

Sample mean	4,3
Variance	1,803448276
Standard Deviation	1,342925268
Confidence interval for mean	4,3 ± t(0,245183487)
90% confidence interval	4,3 ± 0,4166 = (3,8834;4,7166)

Table A.7: Test 7 — 5000 Insert Transactions with 1 client

Sample mean	0,733333
Variance	59,92643678
Standard Deviation	7,741216751
Confidence interval for mean	0,7333 ± t(1,413346346)
90% confidence interval	0,7333 ± 2,4013 = (-1,668; 3,1346)

Table A.8: Test 8 — 5000 Insert Transactions with 2 clients

Sample mean	-0,1
Variance	88,16206897
Standard Deviation	9,389465851
Confidence interval for mean	0,1 ± t(1,7142)
90% confidence interval	-0,1 ± 2,9126 = (-3,013; 2,8126)

Table A.9: Test 9 — 5000 Insert Transactions with 4 clients

Sample mean	-14,4667
Variance	116,8782
Standard Deviation	10,81102
Confidence interval for mean	-14,4667 ± t(1,9738)
90% confidence interval	-14,4667 ± 3,3535 = (-17,82; -11,11)

Table A.10: Test 10 — 10000 Mixed Transactions with 1 client

Sample mean	2,03333
Variance	78,5160
Standard Deviation	8,8609
Confidence interval for mean	2,0333 ± t(1,6177)
90% confidence interval	2,0333 ± 2,7486 = (-0,715; 4,7819)

Table A.11: Test 11 — 10000 Mixed Transactions with 2 clients

Sample mean	-6,5220
Variance	75,00082
Standard Deviation	8,6603
Confidence interval for mean	-6,5220 ± t(1,5811)
90% confidence interval	-6,5220 ± 2,6884 = (-9,208; -3,836)

Table A.12: Test 12 — 10000 Mixed Transactions with 4 clients

APPENDIX B HISTORICAL DATABASE STRUCTURE

The historical database tables and its respective columns, with name, type and extre information.

Column name	Column type	Column properties
ID	INTEGER	PRIMARY KEY
NODE_ID	INTEGER	UNIQUE; NOT NULL

Table B.1: Node Table

Column name	Column type	Column properties
ID	INTEGER	PRIMARY KEY
SECOND	INTEGER	NOT NULL
MINUTE	INTEGER	NOT NULL
HOUR	INTEGER	NOT NULL
M_DAY	INTEGER	NOT NULL
MONTH	INTEGER	NOT NULL
YEAR	INTEGER	NOT NULL
W_DAY	INTEGER	NOT NULL
Y_DAY	INTEGER	NOT NULL
IS_DST	INTEGER	NOT NULL
SPAN	INTEGER	NOT NULL

Table B.2: Time Table

Column name	Column type	Column properties
ID	INTEGER	PRIMARY KEY;
NODE_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Node(NODE_ID)
TIME_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Time(ID)
INTERFACE	TEXT	
R_BYTES	INTEGER	
R_PACKETS	INTEGER	
R_ERRORS	INTEGER	
R_DROP	INTEGER	
R_FIFO	INTEGER	
R_FRAME	INTEGER	
R_COMPRESSED	INTEGER	
R_MULTICAST	INTEGER	
T_BYTES	INTEGER	
T_PACKETS	INTEGER	
T_ERRORS	INTEGER	
T_DROP	INTEGER	
T_FIFO	INTEGER	
T_COLLS	INTEGER	
T_CARRIER	INTEGER	
T_COMPRESSED	INTEGER	

Table B.3: NetworkUsage Table

Column name	Column type	Column properties
ID	INTEGER	PRIMARY KEY;
NODE_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Node(NODE_ID)
TIME_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Time(ID)
SIZE	INTEGER	
RESIDENT	INTEGER	
SHARE	INTEGER	
TEXT	INTEGER	
DATA	INTEGER	
VIRTUALSIZE	INTEGER	
RSS	INTEGER	
RSSLIM	TEXT	
MEM_TOTAL	INTEGER	
MEM_USED	INTEGER	
MEM_FREE	INTEGER	
MEM_BUFFERS	INTEGER	
MEM_CACHED	INTEGER	

Table B.4: MemoryUsage Table

Column name	Column type	Column properties
ID	INTEGER	PRIMARY KEY
NODE_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Node(NODE_ID)
TIME_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Time(ID)
PARTITION_NAME	TEXT	
PARTITION_BLOCKS	INTEGER	
READS_COMPLETED	INTEGER	
READS_MERGED	INTEGER	
WRITES_MERGED	INTEGER	
SECTORS_READ	INTEGER	
MILLISECONDS_READING	INTEGER	
WRITES_COMPLETED	INTEGER	
SECTORS_WRITTEN	INTEGER	
MILLISECONDS_WRITING	INTEGER	
IO_IN_PROGRESS	INTEGER	
MILLISECONDS_SPENT_IN_IO	INTEGER	
WEIGHTED_MILLISECONDS DOING_IO	INTEGER	

Table B.5: DiskUsage Table

Column name	Column type	Column properties
ID	INTEGER	PRIMARY KEY;
NODE_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Node(NODE_ID)
TIME_ID	INTEGER	PRIMARY KEY; NOT NULL; REFERENCES Time(ID)
CORE_ID	INTEGER	
USER	INTEGER	
NICE	INTEGER	
SYSMODE	INTEGER	
IDLE	INTEGER	
IOWAIT	INTEGER	
IRQ	INTEGER	
SOFTIRQ	INTEGER	
STEAL	INTEGER	
GUEST	INTEGER	

Table B.6: CpuUsage Table

APPENDIX C CODE SNIPPETS

Here are some code snippets from the monitoring framework. They are organized by the method name, followed by the file where it is implemented.

C.1 ParseData

```
/*
 * Opens a stream for the file, and return the stream.
 */
std::ifstream& ProcInfoParser::parseData(const std::string& _path) {
        std::ifstream* pFileStream = 0;
        for (int i = 0; i < 5; i++) {
                pFileStream = new std::ifstream(_path.c_str());
                        if (!pFileStream->good()) {
                                std::cerr << "Could not open " << _path
                                << ". Tentative " << i << " of 5" << std::endl;
                                sleep(500);
                        }
        }
        return *pFileStream;
}
```

C.2 Split

```
/*
 * Split a string based in the delimiter received and return a vector
 * with each sub-string.
 */
std::vector<std::string> ProcInfoParser::split(const std::string& s,
char delim, std::vector<std::string>& elems) {
        std::stringstream ss(s);
        std::string item;
        while (std::getline(ss, item, delim)) {
                if (item != "" && !item.empty()) {
                        //Remove ':'
                        size_t foundColon = item.find(":");
                        while (foundColon != std::string::npos) {
                                item.replace(foundColon, 1, " ");
                                foundColon = item.find(":");
                        }
                        //Remove ' '
                        size_t foundSpace = item.find(" ");
                        while (foundSpace != std::string::npos) {
                                item.replace(foundSpace, 1, "");
```

```
                                foundSpace = item.find("␣");
                        }
                        elems.push_back(item);
                }
        }
        return elems;
}
```

C.3 FillNetworkObject

```
usage::NetworkUsage NodeInfoGather::fillNetworkObject(const std::vector<
        std::string> _values, usage::NetworkUsage& _networkObj) {
        int base = 0;
        int offset = 17;
        usage::NetInterface* tempObj = 0;
        for (int base = 0; base < _values.size(); base += offset) {
                tempObj = new usage::NetInterface(_values.at(base),
                        boost::lexical_cast<long>(_values.at(base + 1)),
                        boost::lexical_cast<int>(_values.at(base + 2)),
                        boost::lexical_cast<int>(_values.at(base + 3)),
                        boost::lexical_cast<int>(_values.at(base + 4)),
                        boost::lexical_cast<int>(_values.at(base + 5)),
                        boost::lexical_cast<int>(_values.at(base + 6)),
                        boost::lexical_cast<int>(_values.at(base + 7)),
                        boost::lexical_cast<int>(_values.at(base + 8)),
                        boost::lexical_cast<long>(_values.at(base + 9)),
                        boost::lexical_cast<int>(_values.at(base + 10)),
                        boost::lexical_cast<int>(_values.at(base + 11)),
                        boost::lexical_cast<int>(_values.at(base + 12)),
                        boost::lexical_cast<int>(_values.at(base + 13)),
                        boost::lexical_cast<int>(_values.at(base + 14)),
                        boost::lexical_cast<int>(_values.at(base + 15)),
                        boost::lexical_cast<int>(_values.at(base + 16)));
                _networkObj.SetInterfaces(*tempObj);
        }
        return _networkObj;
}
```

REFERENCES

[Andersen et al. 2009]ANDERSEN, D. G. et al. FAWN: A fast array of wimpy nodes. In: *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT: [s.n.], 2009.

[Barroso e Hölzle 2007]BARROSO, L. A.; HÖLZLE, U. The case for energy-proportional computing. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 40, p. 33–37, December 2007. ISSN 0018-9162.

[Baulig e Poó-Caamaño]BAULIG, M.; POÓ-CAAMAÑO, G. *Libgtop Reference Manual*. [S.l.]: The GNOME Project. Http://developer.gnome.org/libgtop/stable/.

[Bray 2006]BRAY, M. *Review of Computer Energy Consumption and Potential Savings*. December 2006. Available from Internet: <http://www.dssw.co.uk/research/computer_energy_consumption.pdf>.

[Costa et al. 2010]COSTA, R. et al. *Just in Time Clouds: Enabling Highly-Elastic Public Clouds over Low Scale Amortized Resources*. Brasil, 2010.

[Google 2011]GOOGLE. *Google Green: Better Web. Better for the Environment.* 2011. Http://www.google.com/green/the-big-picture.html.

[Härder et al. 2011]HÄRDER, T. et al. Energy efficiency is not enough, energy proportionality is needed! In: *Proceedings of the 16th international conference on Database systems for advanced applications*. Berlin, Heidelberg: Springer-Verlag, 2011. (DASFAA'11), p. 226–239. ISBN 978-3-642-20243-8. Available from Internet: <http://dl.acm.org/citation.cfm?id=1996686.1996716>.

[Hipp 2011]HIPP, R. *SQLite*. 2011. Http://www.sqlite.org/. Http://www.sqlite.org/. Available from Internet: <http://www.sqlite.org/>.

[Koomey et al. 2009]KOOMEY, J. G. et al. *Assessing Trends in the Eletrical Efficiency of Computation Over Time*. August 2009.

[Liang et al. 1999]LIANG, Z. et al. Clusterprobe: an open, flexible and scalable cluster monitoring tool. In: *Cluster Computing, 1999. Proceedings. 1st IEEE Computer Society International Workshop on*. [S.l.: s.n.], 1999. p. 261–268.

[Malony et al. 1992]MALONY, A. D. et al. Performance measurement intrusion and perturbation analysis. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 3, n. 4, p. 433–450, July 1992. ISSN 1045-9219. Available from Internet: <http://dx.doi.org/10.1109/71.149962>.

[Massie et al. 2003]MASSIE, M. L. et al. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, v. 30, 2003.

[Ramey 2004]RAMEY, R. *Boost Serialization Overview*. November 2004. Http://www.boost.org/doc/libs/1_45_0/libs/serialization/doc/index.html.

[RedHat 2007]REDHAT. *Red Hat Enterprise Linux 5.0.0: Red Hat Enterprise Linux Deployment Guide*. Documentation-deployment. [S.l.]: Red Hat, Inc., 2007.

[Reus 2012]REUS, V. U. A GPU operations framework for wattdb. Http://www.inf.ufrgs.br/ vureus/gpu-framework.html. July 2012.

[Ritter 2011]RITTER, K. *Facebook Data Center To Be Built In Sweden*. [S.l.]: Huffington Post, October 2011. Http://www.huffingtonpost.com/2011/10/27/facebook-data-center-sweden_n_1034780.html.

[Schall e Hudlet 2011]SCHALL, D.; HUDLET, V. Wattdb: an energy-proportional cluster of wimpy nodes. In: *Proceedings of the 2011 international conference on Management of data*. New York, NY, USA: ACM, 2011. (SIGMOD '11), p. 1229–1232. ISBN 978-1-4503-0661-4. Available from Internet: <http://doi.acm.org/10.1145/1989323.1989461>.

[Sottile e Minnich 2002]SOTTILE, M. J.; MINNICH, R. G. Supermon: A high-speed cluster monitoring system. In: *In Proc. of IEEE Intl. Conference on Cluster Computing*. [S.l.: s.n.], 2002. p. 39–46.

[Szalay et al. 2010]SZALAY, A. S. et al. Low-power amdahl-balanced blades for data intensive computing. *Operating Systems Review*, v. 44, n. 1, p. 71–75, 2010.

[Tsirogiannis et al. 2010]TSIROGIANNIS, D. et al. Analyzing the energy efficiency of a database server. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 231–242. ISBN 978-1-4503-0032-2. Available from Internet: <http://doi.acm.org/10.1145/1807167.1807194>.