ANTONIO CARLOS SALZVEDEL FURTADO JUNIOR

IMPROVED RESOURCE CONSOLIDATION FOR DATABASE WORKLOADS IN A CLOUD

Term paper presented as a requirement in the Bachelor in Computer Science program from the Informatics Depatment, Exact Sciences Faculty, Universidade Federal do Paraná.
Orientator: Prof. Dr. Eduardo Almeida

CURITIBA

2011

ANTONIO CARLOS SALZVEDEL FURTADO JUNIOR

IMPROVED RESOURCE CONSOLIDATION FOR DATABASE WORKLOADS IN A CLOUD

Term paper presented as a requirement in the Bachelor in Computer Science program from the Informatics Depatment, Exact Sciences Faculty, Universidade Federal do Paraná.
Orientator: Prof. Dr. Eduardo Almeida

CURITIBA

2011

CONTENTS

LIST OF FIGURES

ABSTRACT

Cloud computing has attracted a lot of attention, both in researches and as a business model. It promises a lot of benefits, which include the possibility of cutting costs, better manageability of applications, better utilisation of computational resources, among others. In order to achieve its goals, the cloud is highly dependable on resource virtualization to provide its elasticity. At the same time, the demand for running database systems in virtualized environments grows as a topic in research papers.

Database systems have particularities involving their workloads, when compared to regular applications. This raises some questions about how to optimize the resource allocation among database workloads in a elastic cloud environment. In this paper, we expose an implementation of a *virtual design advisor* in the cloud. Its goal is to improve automatic resource allocation among database management systems running inside virtual machines, by recommending configuration parameters for them according to database workloads.

CHAPTER 1

INTRODUCTION

Cloud computing has attracted a lot of attention lately. It was popularized as a business model by Amazon's Elastic Compute Cloud (EC2), which started selling virtual machines (VMs) in 2006. Over time, more cloud providers have appeared in the market, offering their computational resources (CPU, memory, and I/O bandwidth). This definition of cloud, in which the IT infrastructure is deployed through virtual machines, is referred as Infrastructure-as-a-service (IaaS). One of the most appealing benefits of this paradigm, when associated to cloud providers, is the ability to cut costs. Companies may base their IT strategies on cloud-based resources, spending very little or no money managing their own IT infrastructure. They pay for these resources on-demand, in contrast to the traditional resource provisioning model, in which they would need to deal with both under- and over- utilization of their own resources. Moreover, the cloud providers may offer lower prices because they are benefited from the economy of scale.

These gains are seen on public clouds – clouds made available in a pay-as-you-go manner to the public by an external provider. Although the market has evolved around this type of cloud, organizations might build IaaS clouds using their own infrastructure, known as private clouds. Their aim is not to sell capacity over the internet, but to give local users an agile and flexible infrastructure to run service workloads in their administrative domains. These users are offered VMs, which are scheduled in a group of physical machines within their organization, which we call a cluster. This leads to a better utilization of resources, since services with little demand can be packed into the same machine, process known as server consolidation. This means that on private clouds there is also the possibility to cut IT costs, since the consolidation aims to minimize the number of physical servers needed. Other benefits, such as the migration of VMs between hosts and the ability to dynamically change the amount of resources provided to it, enabled

by the technology present in virtual machine monitors (VMMs), make it possible to deal with fluctuations in the workload. These characteristics propitiate an elastic environment, which is good for a private cloud, and vital for the pay-on-demand model used in public clouds. It's also possible for an organization to mix these two types of clouds, creating a hybrid cloud, which is useful to supplement a private cloud's infrastructure with external resources from a public one. However, this paper will focus on a private cloud environment.

1.1 Context

A cloud is highly dependable on machine virtualization, essential to achieve its goals. Database management systems (DBMSes), like other software systems, are also increasingly being run on virtualized environments for many reasons. Some of them are mentioned in [5], [6] and [7], which include the reduction on the cost of ownership, better provisioning and manageability of applications and the ability to migrate them among physical hosts. Our paper is motivated by the possibility to take a variety of databases that run on dedicated computing resources and move them to a shared resource pool, on a private cloud, process known as *Database Consolidation*. This scenario is discussed in [10]. This white paper gives a good example of how consolidation is applied in a production environment. It lists two deployment models in which this process may be performed onto a private cloud.

In our paper, the deployment model considered is the *Infrastructure Cloud* model, illustrated in figure 1.1. In this model, there is generally a one-to-many relationship between servers and VM guests, considered by this paper. When a database service is requested, the whole operating system stack is built and provided. Elasticity in this model is limited. Although VM guests may be provided more virtual resources (CPU or Memory), they cannot span across servers (i.e. they are limited to the resources from the server they are running on). This means that the full resources of the private cloud cannot be brought to bear on a workload requirement. However, the guests will be able to benefit from the live migration feature, supported by most hypervisors, and thus be

moved to a host with more resource availability.

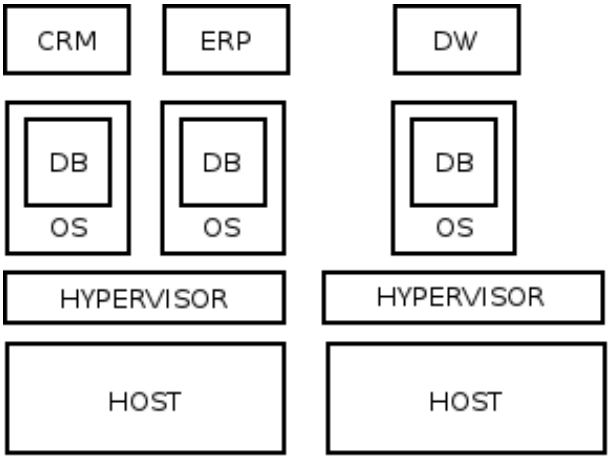

Figure 1.1: Infrastructure cloud deployment model

Some considerations should be made before adopting this deployment model. First, the virtualization won't reduce the number of DBMSes and operating systems, which will surely result in an overhead. It also won't probably be as high performing as other existing non-virtualized alternatives, as I/O intensive workloads may not perform as well in virtualized environments. However, it provides high fault and resource isolation, straightforward database deployment via VM templates and support for multiple DBMS versions and configurations.

1.2 Objective

The problem addressed by our paper is formalized in [6], which defined it as the *Virtualization Design Problem* (a.k.a. resource consolidation problem) for relational database workloads. It can be defined as follows: *"Given N database workloads that will run on N database systems inside virtual machines, how should we allocate the available resources to the N virtual machines to get the best overall performance?"*. According to the mentioned paper, this problem may find a better solution when applied to relational database systems due to three factors. First, relational database workloads consist of SQL queries with constrained and highly specialized resource usage patterns. Second , queries are highly variable in the way they use resources – one query might heavily need CPU, while another might need I/O bandwidth instead. Thus, they could benefit from the dynamism

in resource allocation. Third, database systems already have a way of modelling their own performance, namely the query optimizer.

As mentioned, DBMSes have particularities involving their workloads. Therefore, the application running inside a VM shouldn't be treated as a black box. Instead, the database system cost model should be exploited. As VMMs have parameters to control the share of physical resources, database systems also have tuning parameters to manage their own performance. These two sets need to be simultaneously analysed and tuned. In [7], this is a principle of its proposed *virtualization design advisor*. It works by recommending configuration parameters for a group of VMs, each one containing a DBMS. These parameters determine how the shared resources will be allocated to each VM, and consequently to each DBMS. It uses information about anticipated workloads to specify these parameters offline. Furthermore, runtime information collected after the deployment of the recommended configuration can be used to refine this recommendation and to handle fluctuations in the workload. This solution has not been proposed to run this advisor in a cloud yet, rather it has been implemented and tested in a single physical machine, in which two DBMS instances were deployed.

This paper demonstrates an implementation of the virtualization design advisor proposed in [7] in a cloud environment. Our objective is to solve the *Virtualization Design Problem* in a distributed manner. Each node in a private cloud has its own corresponding instance of our advisor. Because of the *Infrastructure model*, only the resources within the node are be allocated among the VM guests hosted there. Due to time limitations, this paper is restricted to the consolidation of CPU. However, it could be expanded to other types of resource on future work.

The rest of this paper is structured as follows. Chapter 2 is used to discuss related work. In chapter 3, the *virtualization design advisor* is described. Chapter 4 is used to show how a cloud infrastructure is managed. In section 5, we show how the integration of the advisor within the cloud management system was implemented. Section 6 is used to show the results obtained. Finally, section 5 presents some final considerations and ideas for future work.

CHAPTER 2

RELATED WORK

Some papers discuss the cost of virtualizing DBMSes. [5] shows an experimental study of the overhead of running a database workload in a virtual machine. In their experiments, they use Xen[1] as the hypervisor and PostgreSQL[2] as the DBMS. They show that although Xen does indeed introduce overhead for system calls, page fault handling and disk I/O, they are not translate to a high overhead in query execution times. The average overhead found was less than 10%. With a different perspective, [2] proposes its database consolidation solution, namely Kairos. While our solution is based on virtualization, in Kairos, each physical node runs a DBMS instance that processes transactions on behalf of multiple databases. They compare their solution to database virtualization. In their experiments for the virtualized alternative, each database runs on its own operating system, on top the VMware[3] ESXi hypervisor. At the same request rate, it was shown that his solution has 6x to 12x higher throughput. Even though the results may reflect a huge disadvantage in virtualizing databases, in the virtualization tests the workloads are run in separate database servers, which incurs a significant amount of redundant work, like log writing. There is also a significant increase in the amount of context switches, due to the increased number of processes. The paper points out one specific virtualization problem that may have impacted on this result. It causes RAM to be allocated for multiple copies of the DBMS and the OS, which should be reduced by the hypervisor page sharing feature. However, in their experiments with VMware only a small amount of duplicated RAM was reclaimed.

Regarding the resource allocation problem, [9] proposes a multi-resource allocator that dynamically reallocates resources for database servers running on a virtual storage. Their

[1] http://xen.org
[2] http://postgresql.org
[3] http://vmware.com

aim is to proportionate the database, storage server caches and storage bandwidth among applications, according to performance goals. To achieve this, they built a performance model based on minimal statistics collection (e.g. Trace of I/O accesses at the DBMS buffer pool and periodic sampling of the average disk latency). Although they share a similar goal as [7], which serves as base for this paper, the approach is different. The former considers the interplay between different resources (i.e. how changing one resource may affect the others), while the latter treats them independently. Other difference is the information retrieved from the DBMS for each solution, [9] uses very little information from DBMS to build its cost model. Thus ignoring that there is a whole cost model already built within the DBMS query optimizer. On the other hand,[7] is highly dependable on it, even though it may be inaccurate.

CHAPTER 3

VIRTUALIZATION DESIGN ADVISOR

In [7], the author considers a typical resource consolidation scenario, in which several DBMS instances, each one of them running in a separate VM, share a common pool of physical resources. As discussed earlier, this paper addresses the problem of optimizing the performance of these instances by controlling some configuration parameters of the VM guests. These parameters determine how the resources should be indirectly allocated to each DBMS instance. This scenario is illustrated in 3.1.

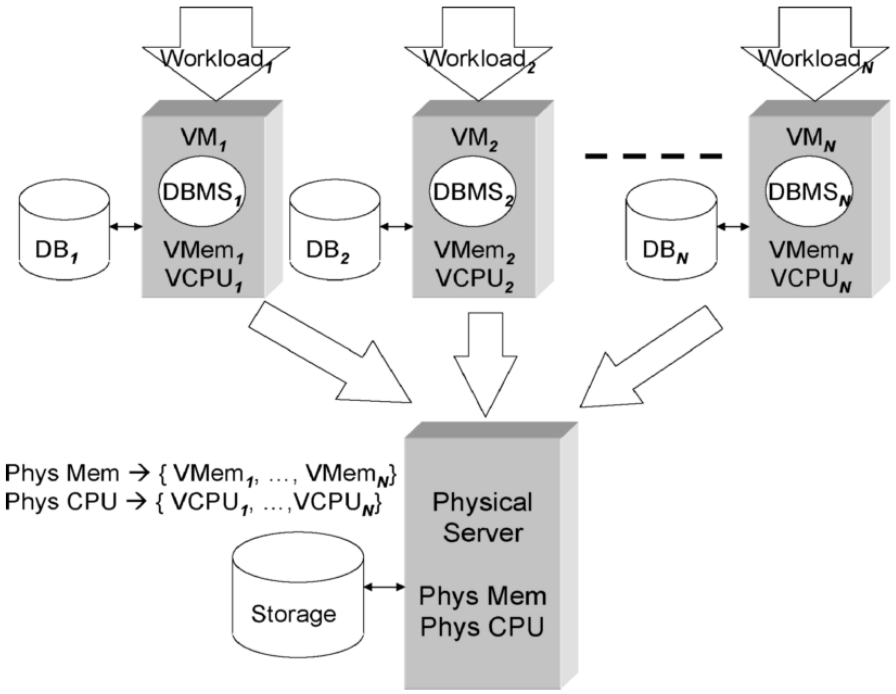

Figure 3.1: Resource Consolidation scenario

3.1 Problem definition

In order to model this problem, the author assumes that there are M types of resources , such as CPU capacity, memory, or I/O bandwidth. The notation used to represent the

share of resources allocated to a VM running a workload W_i is $R_i = [r_{i1}, ...r_{iM}], 0 \leq r_{ij} \leq 1$. It is also considered that each workload W_i has the same monitoring interval for all the VMs (i.e. the workloads represent the statements processed by different DBMS instances in the same amount of time).

Each workload is associated to a cost, which depends on the resource share allocated to the VM in which it runs. The notation $Cost(W_i, R_i)$ is used to represent the cost of running the workload W_i under resource allocation R_i. Considering that there are N workloads, the goal is to choose $r_{ij}, 1 \leq i \leq N, 1 \leq j \leq M$ such that

$$\sum_{i=1}^{N} Cost(W_i, R_i)$$

is minimized.

This problem was generalized to satisfy Quality of Service (QoS) requirements. One of these requirements is the possibility to specify the maximum increase in cost that is allowed for a workload under the recommended resource allocation. In order to model this requirement, it was defined a *cost degradation* as

$$Degradation(W_i, R_i) = \frac{Cost(W_i, R_i)}{Cost(W_i, [1, ..., 1])}$$

, where $[1, ..., 1]$ represents the resource allocation in which all of the available resources are allocated to W_i. It can be specified a *degradation limit* $L_i (L_i \geq 1)$, such that

$$Degradation(W_i, R_i) \leq L_i$$

for all i. This limit is set per workload, so it does not need information about other workloads that it will be sharing the physical server with.

Other QoS requirement introduced involves the ability to specify relative priorities among the different workloads. A *benefit gain factor* $G_i (G_i \geq 1)$ can be used to indicate how important it is to improve the performance of W_i. Each unit of improvement is considered to worth G_i cost units. When this parameter is applied to the problem, it may

cause a workload to get more resources than its fair share. In order to incorporate it to our problem, the cost equation is modified to minimize the following

$$\sum_{i=1}^{N} G_i * Cost(W_i, R_i)$$

3.2 Architecture

The process of determining the allocation of resources to each VM is neither immediate, nor static. The proposed advisor follows a sequence of steps. Initially, it makes a resource allocation based on the workload descriptions and performance goals, which is performed offline (i.e. the VMs are not running yet). Then all of the subsequent steps are performed online. It adjusts its recommendations based on the difference between expected and actual workload costs to correct for any cost estimation errors made during the initial phase. At the same time, it uses continuing monitoring information to dynamically detect changes in the workloads. This last step is important because a workload cannot be considered static, its resource needs may change during execution. In both of these online steps, the resource allocation step may occasionally be performed again with updated cost models. This approach prevents the advisor from allocating resources to DBMS instances that will obtain little benefit from them. These resources need to be given to the workloads that need them the most.

Since the advisor does not consist of one single task, it makes sense to organize the flow of its tasks. An overview of this advisor in a modular way is given in figure 3.2. This paper intends to give a brief explanation of each task.

3.2.1 Cost estimation and initial allocation

Given a workload W_i, the cost estimator will predict $Cost(W_i, R_i)$. The strategy used to implement this module is to leverage the cost models built into database systems for query optimization. The query optimizer cost model can be described as $Cost_{DB}(W_i, P_i, D_i)$, where W_i is a SQL workload, $P_i = [p_{i1}, .., p_{iL}]$ is a vector of parameters that are used to

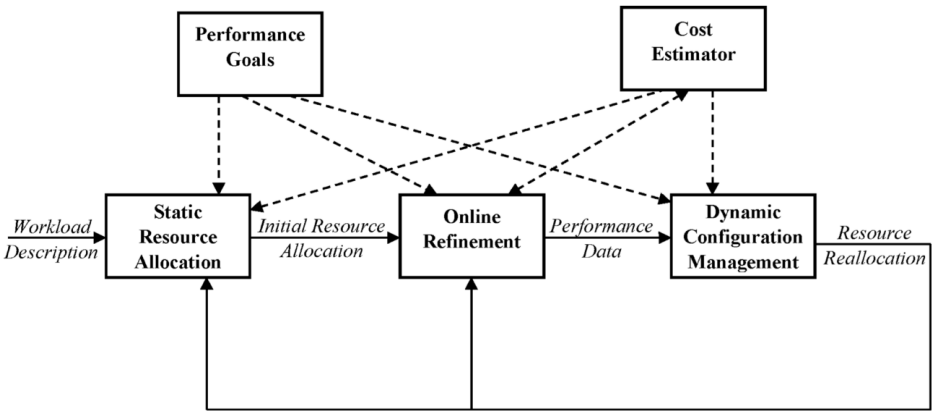

Figure 3.2: Advisor overview

list both the available computing resources and DBMS configuration parameters relevant to the cost model, and D_i is the database instance.

The author identifies two problems in using only query optimizer cost models. The first problem is the difficulty of comparing cost estimates produced by different DBMSes. They may have different cost models. Even if they share the same notion of cost, the normalization process may differ. This problem is addressed partially by the advisor. It assumes that the DBMSes have the same notion of cost, and it proposes a renormalization step to make $Cost_{DB}(W_i, P_i, D_i)$ from different DBMSes comparable.

The second problem is that the query optimizer cost estimates depends on P_i, while the virtualization design advisor is given a candidate resource allocation R_i. The mapping of these two parameters is achieved through a calibration step. This step determines a set of DBMS cost model configuration parameters according to the different possible candidate resource allocations. It is supposed to be performed per DBMS on the physical machine before running the virtualization design advisor. Once the appropriate configuration parameters P_i are determined for every possible R_i, the DBMS cost model is used to generate $Cost_{DB}$.

The calibration step is performed on *descriptive parameters*, which are used to characterize the execution environment. The approach to the *prescriptive parameters*, which control the configuration of the DBMS itself, is to leave it for user definition. For instance, in 3.1 it is shown some descriptive parameters used in PostgreSQL, while in 3.2 it is

shown some prescriptive ones.

Parameter	Description
random_page_cost	Cost of non-sequential page I/O
cpu_tuple_cost	CPU cost of processing one tuple
effective_page_size	size of file system's page size

Table 3.1: Descritive parameters

Parameter	Description
shared_buffers	shared bufferpool size
work_mem	amount of memory used by each sort and hashing operator.

Table 3.2: Prescritive parameters

The calibration step follows a basic methodology for each parameter $p_{ij} \in P_i$, described below:

- (1) Define a calibration query Q and a calibration database D, such that $Cost_{DB}(Q, P_i, D)$ is independent for all descritive parameters in P_i, except for p_{ij};

- (2) Choose a resource allocation R_i, instantiate D, and run Q under that resource allocation, and measure the execution time T_Q;

- (3) Perform the renormalization of $Cost_{DB}$ provided by the DBMS, in order to transform it to a common notion of cost;

- (4) Repeat the two preceding steps for a variety of R_i allocations. $r_{ij} \in R_i$ only needs to be varied if p_{ij} describes that resource. For instance, query optimizer parameters that describe CPU, I/O and memory are independent of each other and can be calibrated independently. This should avoid unnecessary calculations;

- (5) Perform regression analysis on the set of (R_i, p_{ij}) value pairs to determine a calibration function Cal_{ij} that maps resource allocations to p_{ij} values.

During the described methodology, calibration queries should be carefully chosen. They need to be dependent only on the parameter that is being calibrated. If it is not possible to isolate one parameter, a system of k equations is solved to determine the values for the k parameters.

After the calibration is performed, the advisor is ready to be run. First, it needs to provide an initial configuration. In [7], it was proposed a greedy algorithm to search for an initial configuration, as seen in algorithm presented in section 8.1.

The greedy search algorithm presented initially assigns a $\frac{1}{N}$ share of each resource to each one of the N workloads. Then it iteratively considers shifting an amount δ of resources (e.g. $\delta = 5\%$) from one workload to another. It analyses which workload benefits more from gaining that resource and also which loses less. Based on this, it tries to obtain the maximum benefit from adjusting this resource, and it repeats it for every other one. These adjusts are performed until no more optimizations are possible, condition limited by the *REPEAT* loop . It's also important to notice that this algorithm takes into consideration the QoS parameters discussed earlier in this paper, L_i and G_i.

From experimental results, the creator of this algorithm observed that the greedy search is very often optimal and always within 5% of the optimal allocation. This result is a good indicative for the algorithm, as it tells us that it hardly gets stuck in local minimums.

3.2.2 Online refinement

The initial allocation is based on the calibrated query optimizer cost model, as described earlier. This enables the advisor to make recommendations based on an informed cost model. However, this model may have inaccuracies that lead to sub-optimal recommendations. The *online refinement* is based on the observation of the actual times of execution for a workload. It uses these observations to refine resource allocation recommendations.

Then the advisor is rerun with the new cost models, so we can obtain an improved resource allocation for different workloads. These optimizations are performed until the allocations stabilize (i.e. the new recommendation is equal to the last one). It's important to notice that the goal of the *online refinement* is not to deal with dynamic changes in the workload, which are dealt by another module, but to correct cost models. Thus, it's assumed that the workload is not going to change during this process.

In order to optimize the recommendations, it is necessary to identify the cost model behaviour. The author identifies two types of models. The first is the *linear model*, which can describe, among other resources, the allocation of CPU. For this resource, workload completion times are linear in the inverse of the resource allocation level. Therefore, the cost model in this case can be represented by

$$Cost(W_i, [r_i]) = \frac{\alpha_i}{r_i} + \beta_i.$$

The values α_i and β_i are obtained through a linear regression. This regression is performed on multiple points that represent estimated costs for different r_i values used during the initial allocation phase. This cost is adjusted by two parameters, Est_i and Act_i. They represent the estimated cost for workload W_i and the runtime cost, respectively. When the cost is underestimated, these parameters are used to increase the slope of our cost equation. From another standpoint, when this value is overestimated, we need to decrease the slope. This is achieved by refining the cost through the equation

$$Cost'(W_i, [r_i]) = \frac{Act_i}{Est_i} * \frac{\alpha_i}{r_i} + \frac{Act_i}{Est_i} * \beta_i.$$

However, not all resources are linear. The second type of cost model identified is the *piecewise-linear*, which describes, among others, the allocation of memory. Increasing this resource does not consistently result in performance gain. The magnitude and the rate of improvement change according to the query execution plan. The cost equation is similar

to the linear cost model, and it is given by

$$Cost'(W_i, [r_i]) = \frac{Act_i}{Est_i} * \frac{\alpha_{ij}}{r_i} + \frac{Act_i}{Est_i} * \beta_{ij}, r_i \in A_{ij}.$$

The difference here is the parameter A_{ij}, which represents the interval of resource allocation levels corresponding to a particular query execution plan, represented by j. Its intervals are obtained during the initial phase, when the query optimizer is called with different resource allocations and returns different query execution plans with their respective costs. These query execution plans define the boundaries of the A_{ij} intervals. The end of this interval is the largest resource allocation level for which the query optimizer produced this plan. The initial values of α_{ij} and β_{ij} are obtained through linear regression. Together with A_{ij}, they are subsequently adjusted.

Both the equations presented work within their cost model. Nevertheless, a physical machine has more than one type of resource. Therefore, the author extends the cost equations to multiple resources. In this extension, it is considered that M resources are recommended, such that $M - 1$ resources can be modelled using a linear function, while the resource M is modelled using a piecewise-linear function. The cost of workload W_i, given a resource allocation $R_i = [r_{i1}, ..., r_{iM}]$ can be given by

$$Cost'(W_i, R_i) = \sum_{j=1}^{M} \frac{Act_i}{Est_i} * \frac{\alpha_{ijk}}{r_{ij}} + \frac{Act_i}{Est_i} * \beta_{ik}, r_{iM} \in A_{iMk},$$

in which k represents a certain query execution plan, which defines the boundaries of A_{iMk}.

During refinement, this equation is supposed to be performed iteratively, in order to

optimize the parameters α_{ijk} and β_{ijk}. This iteration is shown below.

$$
\begin{aligned}
Cost'(W_i, R_i) &= \sum_{j=1}^{M} \frac{Act_i}{Est_i} * \frac{\alpha_{ijk}}{r_{ij}} + \frac{Act_i}{Est_i} * \beta_{ik} = \sum_{j=1}^{M} \frac{\alpha'_{ijk}}{r_{ij}} + \beta'_{ik}, r_{iM} \in A_{iMk} \\
Cost''(W_i, R_i) &= \sum_{j=1}^{M} \frac{Act_i}{Est_i} * \frac{\alpha'_{ijk}}{r_{ij}} + \frac{Act_i}{Est_i} * \beta'_{ik} = \sum_{j=1}^{M} \frac{\alpha''_{ijk}}{r_{ij}} + \beta''_{ik}, r_{iM} \in A_{iMk} \\
&\vdots
\end{aligned}
$$

The author in [7] proposes a heuristic that changes the equation in the first iteration. Instead of considering the interval A_{iMk}, the first iteration scales to all the intervals of the cost equation (i.e., for all k). This is done because the estimation errors can be partly due to a bias in the query optimizer's view of resource allocation levels. In the second iteration and beyond, this cost will be refined according to the interval A_{iMk}, where the resource r_{iM} will be scaled.

After refining the cost equations, the advisor is rerun. If the newly obtained resource allocation is the same as the original, the refinement process is stopped. Otherwise, it goes on.

3.2.3 Dynamic configuration management

Even if we have an optimal resource allocation, the workload may change its behaviour during execution. It may occur due to the intensity of the workload (e.g. increased number of clients), or the nature of the workload (e.g. new set of queries with different resource needs). These changes are not dealt by the online refinement process, that's why the dynamic configuration management is needed.

The proposed approach consists in monitoring the relative changes in the average cost per query between periods. If the change in the estimated query is above a threshold, θ, we classify this a major change. When this is identified, it's decided to make the virtualization design advisor restart its cost modelling from scratch, before online refinement. The cost model needs to be discarded, since it no longer reflects information about the workload.

In order to deal with minor changes, it's introduced a new metric E_{ip}. It represents

the relative error between the estimated and the observed cost of running workload W_i in monitoring period p. We analyse two consecutive periods. If both $E_{i(p-1)}$ and E_{ip} are below some threshold (e.g. 5%), or if $E_{ip} - E_{i(p-1)} > 0$, then we continue with online refinement. In this case, the errors are either small, or are decreasing. Both of these situations can be efficiently dealt by some iterations of online refinement. However, if this condition is not satisfied, the cost model is discarded once again.

CHAPTER 4

CLOUD INFRASTRUCTURE MANAGEMENT

This section will be used to describe OpenNebula, the virtual infrastructure (VI) manager chosen to have the virtualization design advisor implemented over. Organizations can use it to manage and deploy VMs, individually or in groups that must be co-scheduled on local or external resources. It merges a lot of different technologies, in order to allow users to design their private or hybrid clouds. Some of its key features are:

- It provides a homogeneous view of resources, regardless of the underlying hypervisor (e.g. KVM, Xen, etc.). This makes the virtualization much less restrictive. The physical machine in which the VM is being run does not need to be tied to a specific virtualization technology, which may cause incompatibility issues;
- It manages a VM's full life cycle, it is used to specify storage requirements and setting up the network;
- Supports configurable resource allocation policies.

The OpenNebula architecture is illustrated in figure 4.1. It can basically be split into three layers. The core (middle layer), which has three management areas. One of them is dealing with the creation of virtual networks, which is done through the Virtual Network (VN) manager. This component is responsible for handing out and keeping track of leases (composed by one IP and one MAC address valid on a particular network), and their associations with VMs and physical bridges used. Second, it needs to manage and monitor the physical hosts, which is the responsibility of the host manager. And finally, there is the VM manager, responsible for taking care of a VM's life cycle. It prepares the disk images for them, by controlling the image and storage technologies. It also needs to control the hypervisors, responsible for their creation and control. And combined with the VN manager, this manager provides the VM with the network environment. Another

important feature in the core is achieved with the SQL Pool. It is responsible for saving the OpenNebula state. Even after a shutdown or a failure, it's possible to restore a previous state. All the managers make use of this Pool to store information.

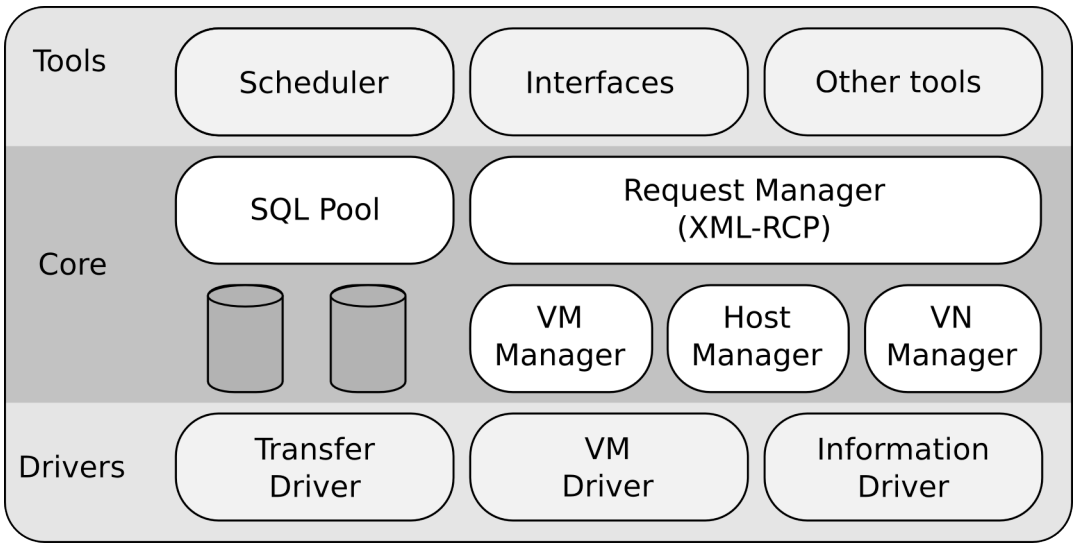

Figure 4.1: OpenNebula architecture

One important feature of the core module is the service deployment support. The Request manager exposes a XML-RPC interface that decouples most of the functionality present in the core. Through this interface it is possible to control and manage any OpenNebula resource, including virtual machines, networks, images, hosts, etc. As OpenNebula is in continuing development, it is expected that this API will support even more functionality. Currently, it is known that dynamic resource allocation is not supported yet. All the resource allocation is performed offline, through VM templates. The amount of resources allocated is passed from the templates directly to the hypervisors, and they'll be the same until the end of that VM's life cycle. We'll discuss more about this issue on the next chapter, as resource reallocation is important for the advisor.

All these described activities are performed through pluggable drivers, present in the bottom layer. This modularity makes the system easier to extend and avoids tying it to any specific technology. The drivers shown in figure 4.1 address some particular areas, such as virtualization (by communicating with the hypervisor), storage operations, gather monitoring information and authenticate user requests.

Other way of extending OpenNebula is through hooks, controlled by the Hook manager. They enable the triggering of custom scripts tied to a change in a particular resource, being that a Host or a Virtual Machine. This opens a wide area of automation for system administrators. Below is an example of a hook, namely "error", defined in the proper configuration file. It is used to perform recovery actions when a host fails (i.e. it enters in the ERROR state). When this happens, the script host_error.rb is called with the host id passed as an argument. The "remote" parameter tells that this script should be executed in the OpenNebula server, and not in the failed host. The purpose of this script is to resubmit all the VMs that were running on the failed host, for them to be run on active hosts.

- HOST_HOOK = [
  name = "error",
  on = "ERROR",
  command = "host_error.rb",
  arguments = "$HID",
  remote = no]

In the top layer there are user interfaces to access OpenNebula, and also APIs extended from the XML-RPC interface, which can work between the core and other tools. One of these APIs is OCA[1]. It exposes the same functionality as XML-RPC, in a more convenient way. Other relevant module for this paper is the OpenNebula's scheduler, also in the top layer. Its job is to make placement decisions for the VMs (i.e. it assigns physical machines for VMs). It has access to all requests OpenNebula receives. Based on these requests, it keeps track on allocations and sends the appropriate deployment commands to OpenNebula's core. As other modules, it can be replaced by third party solutions, such as Haizea[2], which offers more sophisticated placement policies. However, in this paper we will stick to the OpenNebula's default scheduler. Next, we'll describe it in more details.

[1] OpenNebula Cloud API
[2] http://haizea.cs.uchicago.edu/

4.1 Scheduler

The scheduler already present in OpenNebula works only with immediate provisioning. Its concept is pretty straightforward, the resources are only provisioned at the moment they are required, if that's not possible, then the requirement is ignored. The requirements are done in a manual and static way, in which the amount of resources, among other configurations, are defined in a file. The assignment is performed through a classification policy. The administrator needs to set a *RANK* variable, which defines which host is more suitable for a VM. Each VM has its own RANK equation, and the scheduler assigns a host with the highest value for this variable to a VM. In OpenNebula, deployments are performed through VM templates, so you can share the same scheduling policy to a group of VMs. For instance, here are some possible *RANK* definitions, each one serving a specific policy:

- Load-aware Policy
  - **Heuristic:** Use nodes with less load;
  - **Implementation:** Use nodes with more *FREECPU* first.
  - $RANK = FREECPU$
- Striping policy
  - **Heuristic:** Spread the VMs in the cluster node;
  - **Implementation:** Use the nodes with less VMs running first.
  - $RANK =$ "- $RUNNING\_VMS$"

Furthermore, users may use the same syntax applied to *RANK* statements to define a *REQUIREMENT*. Requirements are used to set placement constraints. They rule out hosts from the list of machines suitable to run a certain VM. Below is an example of a requirement statement. It only allows a VM to be run by nodes with CPUs that have clock higher than $1000MHz$:

- $REQUIREMENT = CPUSPEED > 1000$

CHAPTER 5

IMPLEMENTATION

In this chapter, it's discussed the virtualization design advisor implementation over OpenNebula. Our aim is to optimize the distribution of resources inside a private cloud for VM guests running database workloads. However, in order to make this work feasible, some restrictions were made. The major restriction involves the list of resources which would be modelled and optimized. In this paper, this was limited to CPU. Further restrictions were established, and they'll be explained as the advisor is detailed.

For organizational purposes and better integration with OpenNebula, our solution is part of a module called OpenRC, which stands for Open Resource Consolidation. It was designed to work alongside OCA, since it will need full access to OpenNebula functionality. As OCA, it was also implemented in Ruby[1] language. In this module, it was implemented both the steps presented in figure 3.2, from chapter 3, and missing features from OpenNebula, needed for the advisor.

One of the missing features in OpenNebula is the ability to dynamically reallocate resources,as mentioned in chapter 4. Although it is not supported by OpenNebula, most hypervisors currently support it for some types of resource. Besides the need for changes in the OpenNebula drivers to support these operations, it would be necessary to extend the OpenNebula's core to handle them. Instead, it was decided to implement this functionality in OpenRC, with information provided by OCA. The approach to achieve this was the use of libvirt[2], which defines itself as "A toolkit to interact with the virtualization capabilities of recent versions of Linux (and other OSes)". It offers an API that works with all the hypervisors supported by OpenNebula, offering many features, including CPU limitation. Currently, OpenNebula already has a libvirt API, which enables VM management over the core layer. OpenRC uses libvirt at a lower level, correspondent to OpenNebula's

[1] http://www.ruby-lang.org
[2] http://libvirt.org/

bottom layer. At this time, libvirt has three parameters that allows us to control the cpu scheduling. Here is an explanation of their use:

- **quota** and **period**: These parameters work together, they allow us to set a hard limit for the cpu usage. **Period** should be in range $[1000, 1000000]$. Within it, each virtual CPU of the VM will not allowed to consume more than **quota** worth of runtime. For instance, the following values define that the VM guest will be limited to 20% of CPU time:
  - **period**= 100000;
  - **quota**= 20000;
- **shares**: Opposite to the previous parameters, **shares** enables us to set a soft limit for CPU usage. It specifies the proportional weighted share for the VM guest. Its value is relative. For instance, a VM configured with value 2048 will get twice as much CPU time as a VM with value 1024. However, if the CPU is not being used by the VM with a higher parameter value, the other will not be denied CPU idle time.

These parameters are indirectly updated and retrieved through a class called **CPULimitation**, within OpenRC. The objective of this class is to set hard/soft limits for CPU usage of OpenNebula VMs on remote hosts. Our approach to execute these remote commands is the same as OpenNebula, in fact for this task we only import the OpenNebula's class **RemotesCommand**, which is responsible for executing OpenNebula commands through SSH[3] on remote hosts.

Other implemented feature was the communication with the DBMS running inside the VM guest. This was achieved by the creation of the **DatabaseHelper** class. Structured according to the facade pattern, it enables access to a set of classes, responsible for establishing a database connection, set tuning parameters, process single queries or workloads, obtain estimated and actual runtime costs, etc. This pattern was chosen to

[3]Secure Shell

make the support of other DBMSes easier in future work. For the purposes of this paper, only PostgreSQL[4] was supported and tested as DBMS. This decision was not only based on implementation details, but also to ease cost modelling and parameter calibration. The support of a new DBMS starts by the creation of a module, representing the DBMS. In this module, it should be defined classes which have its functionality exposed by **DatabaseHelper**, as illustrated by figure 5.1. As these classes have database specific methods, they shouldn't be directly implemented in the facade object. For instance, when a query is run for analysing purposes, the costs are not obtained the same way for different DBMS types. Once the database module is created, following specific implementation rules, it can be easily included.

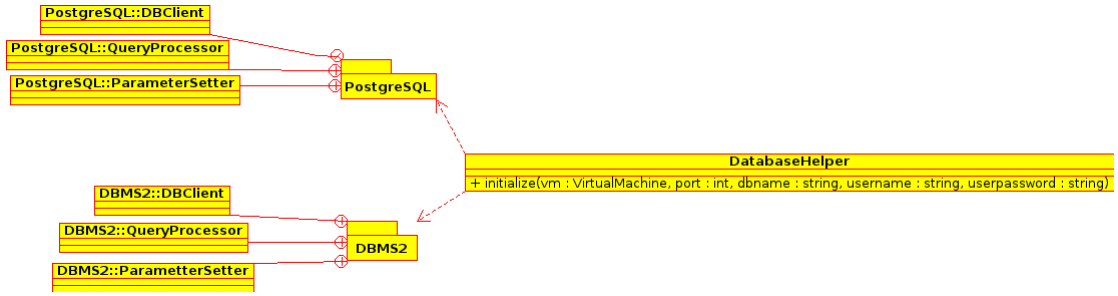

Figure 5.1: DatabaseHelper facade with an eventual support of a second DBMS

A **DatabaseHelper** instance needs to be initialized with a **VirtualMachine** object, defined in OCA. As mentioned in chapter 4, OpenNebula is able to define both the MAC and IP address of a Virtual Machine, in form of a lease. It also keeps track of these leases in its SQL Pool. As any other pool element, this information can be retrieved through OCA, in XML format. So when a VM is passed to our class, it's possible to retrieve the IP address of that VM and connect to PostgreSQL. As some authentication parameters (database name, user name, user password) are also needed for database connection, and they are not stored by OpenNebula SQL pool, they must be passed manually to our solution.

As soon as the communication with the DBMS and the CPU limitation were implemented, it was possible to start developing modules inherent to our advisor. In the following subsections, every part of the process has its implementation detailed.

[4]http://www.postgresql.org/

5.1 Calibration

The first step in order to implement the advisor is the creation of a calibration process. As discussed earlier, this task is responsible for mapping the query optimizer's cost model, which depends on its parameters P_i, to the advisor's cost model, based on resources R_i. Without it, it's not possible to build a proper cost estimator. Since the calibration should be performed before any VM deployment on each physical host on our cloud, it was decided that this process should become an OpenNebula hook. The hook definiton is given below:

- HOST_HOOK = [
  name = "Calibration",
  on = "CREATE",
  command = "openrc/host_db_calibration.rb",
  arguments = "$HID",
  remote = no]

This hook will execute the host_db_calibration.rb script on the OpenNebula server whenever a new host is created, passing the host id as argument. In this script, the whole calibration process is performed, as described in chapter 3. Initially, it searches for the image of a specific VM, which basically contains the calibration database. As hosts and virtual machines, images are also objects managed by OpenNebula. Thus, it needs to be added to the image pool before running the calibration script.

The calibration database running inside the VM is generated by pgbench[5]. This tool is able to run benchmark tests, based on TPC-B, for PostgreSQL. During implementation, it was only used to generate and populate our calibration database. Its choice relies on its integration with PostgreSQL, usability, and the ability to populate databases with custom sizes.

[5] http://www.postgresql.org/docs/devel/static/pgbench.html

Once the calibration database is up and running, it's necessary to run specially designed queries to find out the relation between different resource allocation levels and tuning parameters costs. As mentioned in subsection 3.2.1, DBMSes don't generally share the same notion of cost. It means that we also must find the relation between DBMS estimated costs and actual costs, which would be the execution times. In PostgreSQL, all costs are normalized with the respect of time required for a single page to be fetched from disk. This cost is represented by the parameter *seq_page_cost*. Therefore, by finding it's actual value, we also find this relation.

Our approach to find the actual cost values for *seq_page_cost* was to use a tool to find the virtual disk throughput. In this implementation, the tool hdparm[6] was chosen. Among other features, this utility serves as a disk benchmarking alternative for Linux. It must be previously installed in the calibration VM. Its results are obtained through a specially designed PostgreSQL function, added to the calibration database. Based on the results retrieved from this function, the cost of *seq_page_cost* is calculated. Besides this calculation, it's also important to decide how to simulate I/O disk contention at this step. This would happen in production environments, where multiples VM guests are being run concurrently. Calculating *seq_page_cost* without considering a realistic scenario could lead to inaccurate results. It was decided to magnify this problem by stressing the host disk while the costs are being generated and retrieved. The stress[7] utility was chosen to generate disk stress. It's used to spin several workers writing on files and removing directory entries on the remote host. In a more realistic setup, the VMs would compete less for I/0, and probably this cost would have a certain variability. By stressing the disk, our aim is to simulate a worst-case scenario.

After finding the relation between the DBMS estimated cost and the actual cost (execution time), it's possible to calibrate the tuning parameters. Since this paper is restricted to reallocate CPU among VMs, only parameters that describe CPU need to be calibrated. For PostgreSQL, these are shown below.

[6]http://sourceforge.net/projects/hdparm/
[7]http://freecode.com/projects/stress

Parameter	Description
cpu_operator_cost	Cost of processing each operator or function call
cpu_tuple_cost	Cost of processing one tuple (row)
cpu_index_tuple_cost	Cost of processing each index entry during an index scan

Table 5.1: Parameters that describe CPU

PostgreSQL uses these parameters to build its query optimizer's cost estimates. That's why a lot of documentation can be found on how to tune these parameters in order to obtain better cost execution plans. However, the way these parameters are exactly used within these plans is poorly documented. Only by observing the source code is possible to determine how they are applied to simple query plans. The first two parameters at table 5.1, *cpu_tuple_cost* and *cpu_operator_cost* can be obtained by the query presented at subsection 8.2.1. Without changing the values of any PostgreSQL's default parameter values, this query will always have the same execution plan generated. It will access the table *pgbench_accounts* using a sequential scan, and subsequently perform an aggregation. PostgreSQL has the following equations for these two operations:

$$SEQSCAN = (seq\_page\_cost * num\ pages\ fetched) + (cpu\_tuple\_cost * number\ rows)$$
$$AGGREGATE = SEQSCAN + (cpu\_operator\_cost * number\ rows) \tag{5.1}$$

The estimated cost and execution times for these equations can be obtained by executing this query on PostgreSQL with the command $EXPLAIN(ANALYZE)$. In this case, the query will be actually executed and an execution plan will be returned. Each step in the execution plan has its corresponding costs indicated. This means that it's possible to isolate costs for each operation from the total. Based on the actual costs returned, our script is able to deduct the parameters from the equations. However, by retrieving these costs we get a different scale, as the actual costs are in milliseconds, instead of page fetches. We use the actual value of *seq_page_cost* to make conversions between different

scales. The equation is given below:

$$param_{estimated} = \frac{param_{actual}}{seq\_page\_cost_{actual}}$$

To calculate the first two descriptive parameters, only one modification to the original equation was made. PostgreSQL considers the cost fetching pages during a $SEQSCAN$. The problem is to determine how many pages were actually fetched from disk. One approach would be running the query with the command $EXPLAIN(ANALYZE, BUFFERS)$. By using this command, it's informed the number of pages hit in the PostgreSQL cache and number of pages that needed to be read. However, the PostgreSQL cache may not reflect the state of the operating system cache. During experimentations, changes in the hit/miss ratio did not incur in an expected change in actual costs. Our approach was to cache all the data for the consults and limit the number of page accesses of a determined query. After doing that, costs related to disk page fetches were ignored from the equation.

The third descriptive parameter, $cpu\_index\_tuple\_cost$, is used during index scans. It can be obtained through the query presented in subsection 8.2.1. Before running this query, it is necessary to set the following parameter:

$$enable\_seqscan = off$$

. This will force the planner to use an index scan plan, as *aid* is a primary key, instead of a sequential scan. Its equation is much more complex than the previous one. Mainly because it involves a lot of I/O cost estimates. In addition to seq_page_cost, its plan is highly dependable on another parameter, namely $random\_page\_cost$. It represents the cost of fetching a non-sequentially disk page. During estimation, PostgreSQL uses an approximation of pages actually fetched after accounting for cache effects. This approximation is proposed in [4]. The number of fetched pages is

$$PF = \begin{cases} min(2TNs/(2T+Ns), T) & when \quad T \leq b \\ 2TNs/(2T+Ns) & when \quad (T > b) \wedge (Ns \leq 2Tb/(2T-b)) \\ b + (Ns - 2Tb/(2T-b)) * (T-b)/T & when \quad (T > b) \wedge (Ns > 2Tb/(2T-b)) \end{cases}$$

where

T = number of pages in table;

N = number of tuples in table;

s = selectivity = fraction of table to be scanned;

b = number of buffer pages available.

For the same reasons why I/O cost was discarded for the first equation, we decided to eliminate it from the *cpu_index_tuple_cost* calculation. Since we are only interested in CPU cost, instead of I/0, we cache all the data to be retrieved and consider the following parameters to be

$$\begin{aligned} seq\_page\_cost &= 0 \\ random\_page\_cost &= 0 \end{aligned} \tag{5.2}$$

Given this configuration, the equation for a single index scan can be defined as

$$
\begin{aligned}
Cost = & \\
& ntuples * (cpu\_index\_tuple\_cost + qual\_op) + \\
& 100 * cpu\_operator\_cost + \\
& cpu\_per\_tuple\_cost * ntuples + (I/0\ Cost) \\
qual\_op = & \\
& cpu\_operator\_cost * ncond \\
cpu\_per\_tuple\_cost = & \\
& cpu\_tuple\_cost + cpu\_operator\_cost * nfilters
\end{aligned}
$$

,where

ntuples = number of tuples retrieved;

ncond = number of index conditions;

nfilters = number of filters (including index conditions);

I/O cost = indicates the cost of fetching pages from disk, related to parameters *seq_page_cost* and *random_page_cost*. As it is discarded by the calibration process, it won't be detailed.

The values for *cpu_tuple_cost* and *cpu_operator_cost*, the details about our calibration query and database are already known. As we discared the I/O cost from our equation, there is only one variable left to be deducted, that is *cpu_index_tuple_cost*. The calibration process then isolates this variable from the rest of the equation and finds its value under different resource allocations, like it was done for the two first parameters. The library built to enable CPU limitation is used here to set a hard limit for CPU usage.

The execution costs found during the calibration step are then stored in files, with the corresponding CPU allocation levels. The collected data is used later by the virtualization design advisor.

5.2 Virtualization Design Advisor

After the end of the calibration process for a certain host, the virtualization design advisor can start handling database workloads and resource allocation levels. The advisor modules, shown in figure 3.2, served as a base on how to organize our implementation. This can be seen on figure 5.2. Here, a new process was added, namely **LoadGenerator**. Its task is to run the workloads under resource allocations conditions established during **GreedySearch**. The costs obtained through execution are passed to both **OnlineRefinement** and **DynamicConfigurationManagement**. During workload execution, this class may call **GreedySearch** several times for reallocation, regarding changes in the workload or recurrent allocation optimizations. In this section, each class within the advisor will be detailed.

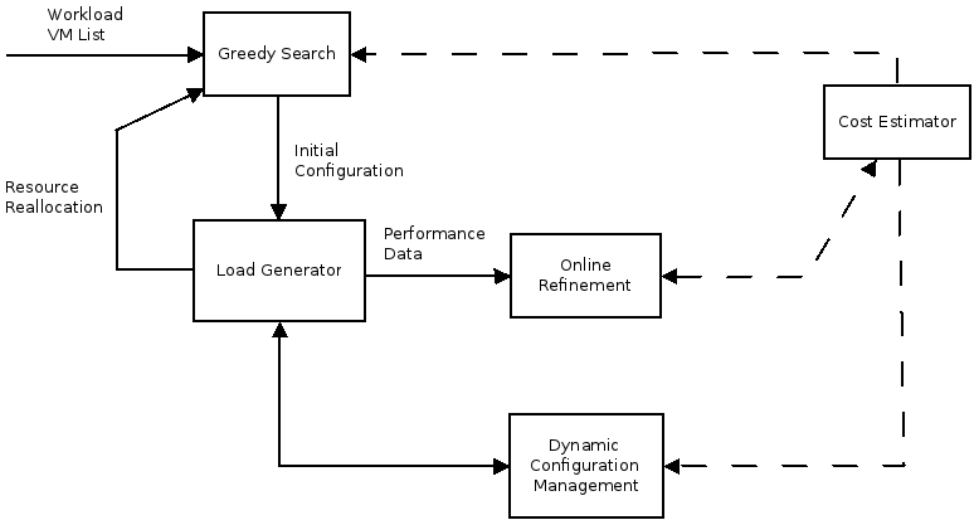

Figure 5.2: Implementation overview

5.2.1 Greedy Search

The **GreedySearch** class, from which the advisor is started, is an implementation of the algorithm presented in section 8.1. This is the only class that actually changes the resource allocation levels. Considering that a private cloud is composed of multiple hosts, it is necessary to run multiple instances of this class. It was created a daemon to manage these multiple searches. It waits for virtual machines to be started and also for workloads

to be run on these VMs. Then it resends this information to the **GreedySearch** instance that corresponds to the host in which that particular VM was designated to be run. Information about newly created VMs can be automatically added as soon as they are created, through a OpenNebula hook. On the other hand, workloads are sent with a script, which sends their information through a TCP Socket, in an expected format.

The search is used in three occasions. The first is when the advisor is started. The objective, in this case, is to find a better initial allocation than the default (i.e. find an allocation of resources for the N VM guests better than the proportion $\frac{1}{N}$). It relies on estimates provided by the **CostEstimator** class to accomplish it. In a production environment, a good approach would be to base these estimates in a workload history. As we only have a testing scenario, it was opted to base our search on the first workload that is yet to be run.

In the other two occasions, the search is called by the **LoadGenerator** class. The first is related to the online refinement process, described in subsection 3.2.2. As the cost estimator model is refined after each workload execution, new calls are made to reflect these refinement changes in the resource allocation. In this case, the searches are not restarted from the default allocation, instead we search for improvements to the current allocation. If the allocation remains the same, this search will not be called any longer by **LoadGenerator**. And finally, the last occasion happens when a workload change is detected. Then **LoadGenerator** calls this class, and the search will be started from scratch. This case is similar to the first search, as both of them start from the default allocation and the cost model is restarted.

Independently on how the search is called, it always ends by applying the resource allocation levels found. **GreedySearch** is the only class within the advisor that actually changes them. This is applied in two steps. The first corresponds to setting a soft limit on the CPU level for each VM, according to the resource allocation levels found during search. The second step is to set the parameters for each DBMS, also according to the search. After they are applied, the rest of the workload can be run.

5.2.2 Cost Estimator

The cost estimation process is performed by the class **CostEstimator**. The description of how this task should be performed in the virtualization design advisor is given in subsection 3.2.1. Basically, it needs to provide an estimated cost for running a certain workload under a determined resource allocation level. During initial calls to this estimator, the cost model is not built yet. So **CostEstimator** leaves this task to be performed by the cost estimator built inside the DBMS query optimizer. Here we face a problem described previously, while our cost model depends on a resource allocation R_i, the query optimizer cost estimates depend on a set of tuning parameters P_i. This mapping has already been performed in the calibration process. Therefore, this class collects all calibration data for the tuning parameters and uses linear regression on the observed points.

The equations obtained through linear regression for each parameter will be used to map R_i to P_i. Besides being useful to the **CostEstimator** class, they also help the **GreedySearch** class to know how to set the DMBS parameters under a certain resource allocation level. They stop being used for cost estimation purposes when the cost model is built. Our approach to build the cost model is by keeping statistics of previous calls to this class. During an initial search, **CostEstimator** may be called several times for different resource allocation levels, and so the estimated results are stored. Before running the workload, a linear regression is performed on these points. As in [7] CPU is described as a resource that varies linearly, a linear regression model provides a good approximation. After building the cost model, this class stops calling PostgreSQL for estimates. This significantly reduces the extra calls to the DBMS. So the advisor is expected to have a better performance as it is being run.

Other difference between our cost estimator and the one built inside the query optimizer is the scale used. Our approach was to use use costs based on query runtime. The reason is that this is easy to determine and a better approach to compare different DBMSes. As already mentioned in section 5.1, PostgreSQL has a different notion of cost. Like it was done during the calibration process, we simply use the actual value of

seq_page_cost to convert between the two scales. Our renormalization equation is

$$Cost(W_i, [r_i]) = Cost_{DB}(W_i, P_i, D_i) * seq\_page\_cost_{actual}$$

. Once the cost model is built and the we are able to renormalize the cost, it can be refined by the **OnlineRefinement** class.

5.2.3 Online Refinement

The online refinement process adopted in our advisor is straightforward. Considering that the CPU has a linear model and that we only need to refine this resource, the **OnlineRefinement** class is an implementation of the first refinement equation presented in subsection 3.2.2. This equation is designed specifically for refining resources with linear variation. During execution, the cost model is refined the following way:

$$\begin{aligned} Cost(W_i, [r_i]) &= \frac{\alpha_i}{r_i} + \beta_i \\ Cost'(W_i, [r_i]) &= \frac{Act_i}{Est_i} * \frac{\alpha_i}{r_i} + \frac{Act_i}{Est_i} * \beta_i = \frac{\alpha_i'}{r_i} + \beta_i' \\ Cost''(W_i, [r_i]) &= \frac{Act_i}{Est_i} * \frac{\alpha_i'}{r_i} + \frac{Act_i}{Est_i} * \beta_i' = \frac{\alpha_i''}{r_i} + \beta_i'' \\ &\vdots \end{aligned} \tag{5.3}$$

. α_i and β_i are parameters of the linear model for workload W_i, and Act_i and Est_i are the actual and estimated costs for running this workload, respectively. In order to apply these equations, this class needs to receive the estimated and the actual costs for this workload. The actual runtime costs are provided by the **LoadGenerator** class, and the estimated costs come from the **CostEstimator**. The latter will have then its cost model updated through the refinement performed.

5.2.4 Load Generator and Dynamic Configuration Management

The **LoadGenerator** class was added to the advisor with the objective to simply run the workloads. Besides increasing the cohesion among the advisor classes, it helps us

to determine which parts of the advisor we want to enable. Both the online refinement process and the workload change detection may be disabled in order to test a specific scenario in the advisor.

This class launches a thread for each VM running on a certain host, in order to run the workloads concurrently. Each VM is designated a workload queue. Even though workload units may be received from multiple clients, they are stored in a single queue, which is processed sequentially. This definitely differentiates our tests from a production environment, where a database may receive multiple queries concurrently. It limits our advisor in testing workload intensities. Therefore, in this paper our results are focused on the workload nature, rather than its intensity. As the execution results are obtained, they are passed to **OnlineRefinement**, if it is enabled.

While the **LoadGenerator** runs the queries for each workload, the **DynamicConfigurationManagement** monitors this execution periodically. It is a simplified version of the module described in subsection 3.2.3. Every monitoring period, it checks for relative changes in the cost per query. If the change is above a threshold θ (i.e. it's a major change), it reports it. Otherwise, the changes are ignored.

The relation between these two classes was implemented according to the observer pattern. **LoadGenerator** acts as the observer, although it only starts observing the class after all the cost models have been defined. If a major change in the workload is detected by **DynamicConfigurationManagement**, it notifies the observer . When this happens, the **LoadGenerator** class decides to restart from the greedy search process, discarding the cost model.

One of the virtualization design advisor features that was left out in this implementation was the support of QoS parameters. This means that during execution, all the workloads are treated equally, and no retsrictions are made to their improvement or decrease in performance. The implementation of the *benefitial gain factor* and the *cost degradation* will be left for future work.

CHAPTER 6

RESULTS

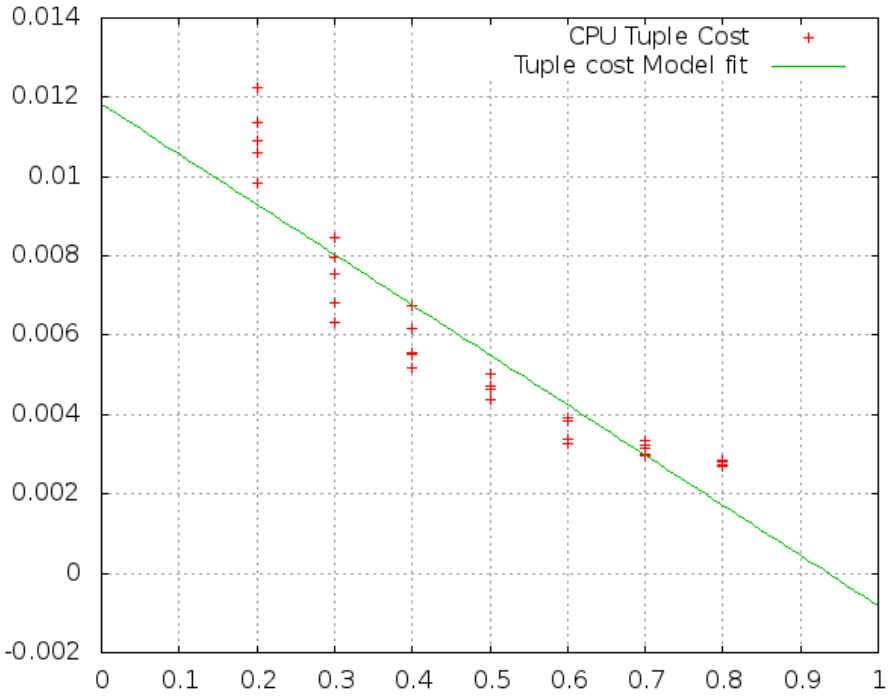

Figure 6.1: *cpu_operator_cost* calibration

Figure 6.2: *cpu_tuple_cost* calibration

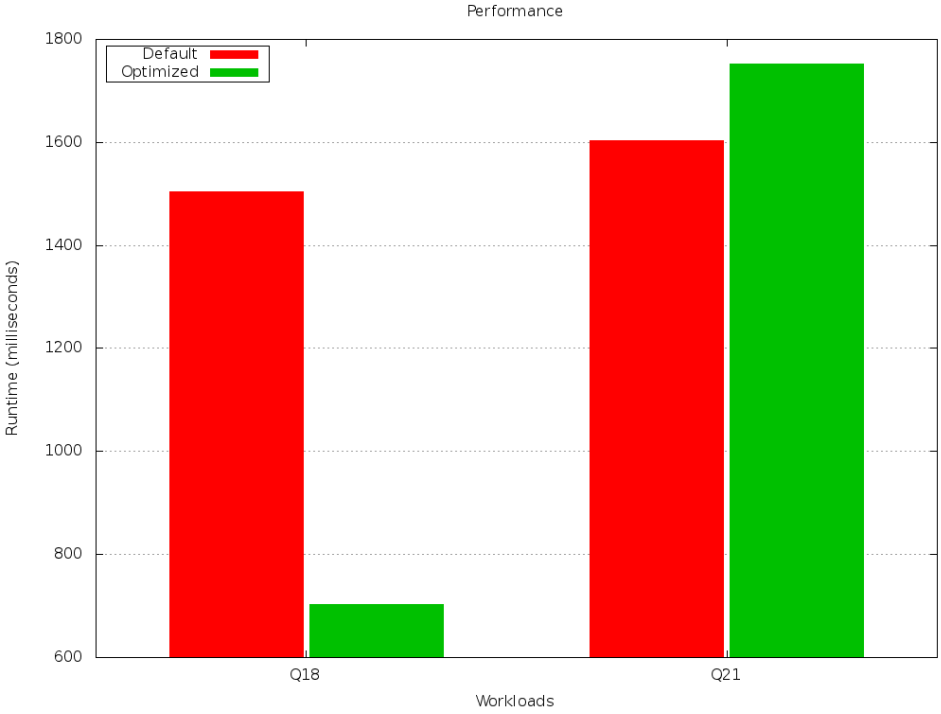

Figure 6.3: Overall performance

CHAPTER 7

FINAL CONSIDERATIONS

CHAPTER 8

APPENDICES

8.1 Greedy search algorithm

Algorithm 1 Greedy search algorithm

for $i = 1 \rightarrow N$ **do**
  $R_i \leftarrow [\frac{1}{N}, .., \frac{1}{N}]$
end for
$done \leftarrow$ **false**
repeat
  $MaxDiff \leftarrow 0$
  for $j = 1 \rightarrow M$ **do**
    $MaxGain_j \leftarrow 0$
    $MaxLoss_j \leftarrow \infty$
    for $i = 1 \rightarrow N$ **do**
      $C' \leftarrow G_i{*}Cost(W_i, [r_{i1}, r_{ij}{+}\delta, r_{iM}])$ # Maximize gain
      if $C_i - C' > MaxGain_j$ **then**
        $MaxGain_j \leftarrow C_i - C'$
        $i_{gain} \leftarrow i$
      end if
      $C' \leftarrow G_i{*}Cost(W_i, [r_{i1}, r_{ij}{-}\delta, r_{iM}])$ # Minimize loss
      if $(C' - C_i < MaxLoss_j) \wedge (C' satisfies L_i)$ **then**
        $MaxLoss_j \leftarrow C' - C_i$
        $i_{loss} \leftarrow i$
      end if
    end for
    # Maximum benefit from adjusting this resource
    if $(i_{gain} \neq i_{lose}) \wedge (MaxGain_j - MinLoss_j > MaxDiff)$ **then**
      $MaxDiff \leftarrow MaxGain_j - MinLoss_j$
      $i_{maxgain} \leftarrow i_{gain}$
      $i_{maxloss} \leftarrow i_{loss}$
      $j_{max} \leftarrow j$
    end if
  end for
  if $MaxDiff > 0$ **then**
    $r_{i_{maxgain}j_{max}} \leftarrow r_{i_{gain}j_{max}} + \delta$
    $r_{i_{maxloss}j_{max}} \leftarrow r_{i_{loss}j_{max}} - \delta$
  else
    $done \leftarrow$ **true**
  end if
until done

8.2 Calibration queries

8.2.1 cpu_tuple_cost and cpu_operator_cost

```
SELECT max(abalance) FROM pgbench_accounts;
```

8.2.2 cpu_index_tuple_cost

```
SELECT *
        FROM pgbench_accounts AS pa
        WHERE pa.aid > 0
        LIMIT 2000;
```

REFERENCES

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

[2] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[3] Donald Kossmann and Tim Kraska. Data management in the cloud: Promises, state-of-the-art, and open questions. *Datenbank-Spektrum*, 10:121–129, 2010. 10.1007/s13222-010-0033-3.

[4] Lothar F. Mackert and Guy M. Lohman. Index scans using a finite lru buffer: a validated i/o model. *ACM Trans. Database Syst.*, 14(3):401–424, September 1989.

[5] U.F. Minhas, J. Yadav, A. Aboulnaga, and K. Salem. Database systems on virtual machines: How much do you lose? In *Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on*, pages 35 –41, april 2008.

[6] A.A. Soror, A. Aboulnaga, and K. Salem. Database virtualization: A new frontier for database tuning and physical design. In *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, pages 388 –394, april 2007.

[7] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 953–966, New York, NY, USA, 2008. ACM.

[8] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14 –22, sept.-oct. 2009.

[9] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic resource allocation for database servers running on virtual storage. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.

[10] Nitin Vengurlekar. Database Consolidation onto Private Clouds. Technical report, Oracle Corporation, February 2011.

ANTONIO CARLOS SALZVEDEL FURTADO JUNIOR

IMPROVED RESOURCE CONSOLIDATION FOR DATABASE WORKLOADS IN A CLOUD

Term paper presented as a requirement in the Bachelor in Computer Science program from the Informatics Depatment, Exact Sciences Faculty, Universidade Federal do Paraná.
Orientator: Prof. Dr. Eduardo Almeida

CURITIBA

2011