Balancing Money and Time for OLAP Queries on Cloud Databases

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Science (Engineering)

IN THE

Faculty of Engineering

BY

Rafia Sabih

Department of Computational and Data Sciences

Indian Institute of Science

Bangalore – 560 012 (INDIA)

November, 2016

Declaration of Originality

I, **Rafia Sabih**, with SR No. **06-02-00-10-21-13-1-10500** hereby declare that the material presented in the thesis titled

Balancing Money and Time for OLAP Queries on Cloud Databases

represents original work carried out by me in the **Department of Computational and Data Sciences** at **Indian Institute of Science** during the years **2013-16**.

With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

The One

who resides above all clouds and balances everything beneath it

Signature of the Author: . . . .

Rafia Sabih

Department of Computational and Data Sciences

Indian Institute of Science, Bangalore

Signature of the Thesis Supervisor: . . . .

Jayant R Haritsa

Professor

Department of Computational and Data Sciences

Indian Institute of Science, Bangalore

Acknowledgements

I can neither begin nor complete this acknowledgment without thanking Prof. Jayant Haritsa for his utmost sincerity and support during the course of this work. Working with him was both pleasure and privilege. Among many things I learned from him, the importance of humor and humility is the most profound one.

Just like my time in IISc, this experience could not have been this great without the support of Anshuman. Though I find thanks a small word to express my gratitude for you, but being short on vocabulary, thanks for your critical and honest comments on almost everything. I also thank some of my dear friends who helped me throughout my stay in IISc, specially Vineet for all those encouragement and 'I share the pain' sessions. On a lighter note, thanks a ton Pankhuri and Vibhuti for never believing that we all will live upto this day, and we lived each day as the last day together (credits to Vibhuti). Thank you my namesake for your warm friendship and adding an amazing experience in my life. Not to be missed, DSL-ites, thanks a lot for everything.

Lastly, *Jazak Allahu Khayran* to my parents, siblings, and Sheez for their endless support. Cheers!

Abstract

Enterprise Database Management Systems (DBMSs) have to contend with resource-intensive and time-varying workloads, making them well-suited candidates for migration to cloud platforms – specifically, they can dynamically leverage the resource elasticity while retaining affordability through the pay-as-you-go rental interface. The current design of database engine components lays emphasis on maximizing computing efficiency, but to fully capitalize on the cloud's benefits, the outlays of these computations also need to be factored into the planning exercise. In this thesis, we investigate this contemporary problem in the context of industrial-strength deployments of relational database systems on real-world cloud platforms.

Specifically, we consider how the traditional metric used to compare query execution plans, namely response-time, can be augmented to incorporate monetary costs in the decision process. The challenge here is that execution-time and monetary costs are *adversarial* metrics, with a decrease in one entailing a rise in the other. For instance, a Virtual Machine (VM) with rich physical resources (RAM, cores, etc.) decreases the query response-time, but is expensive with regard to rental rates. In a nutshell, there is a tradeoff between money and time, and our goal therefore is to identify the VM that offers the best tradeoff between these two competing considerations. In our study, we profile the behavior of *money versus time* for a given query, and define the best tradeoff as the "knee" – that is, the location on the profile with the minimum Euclidean distance from the origin.

To study the performance of industrial-strength database engines on real-world cloud infrastructure, we have deployed a commercial DBMS on Google cloud services. On this platform, we have carried out extensive experimentation with the TPC-DS decision-support benchmark, an industry-wide standard for evaluating database system performance. Our experiments demonstrate that the choice of VM for hosting the database server is a crucial decision, because: (i) variation in time and money across VMs is significant for a given query, (ii) no one VM offers the best money-time tradeoff across all queries.

To *efficiently* identify the VM with the best tradeoff from a large suite of available configurations, we propose a technique to characterize the money-time profile for a given query. The core of this technique is a VM pruning mechanism that exploits the property of partially ordered set of the VMs on their resources. It processes the minimal and maximal VMs of this poset for estimated query response-time. If the response-times on these extreme VMs are *similar*, then all the VMs sandwiched between them are pruned from further consideration. Otherwise, the already processed VMs are set aside, and the minimal and maximal VMs of the remaining unprocessed VMs are evaluated for their response-times. Finally, the knee VM is identified from the processed VMs as the one with the minimum Euclidean distance from the origin on the money-time space. We theoretically prove that this technique always identifies the knee VM; further, if it is acceptable to find a "near-optimal" knee by providing a relaxation-factor on the response-time distance from the optimal knee, then it is also capable of finding more efficiently a satisfactory knee under these relaxed conditions.

We propose two flavors of this approach: the first one prunes the VMs using complete plan information received from database engine API, and named as Plan-based Identification of Knee (PIK). On the other hand, to further increase the efficiency of the identification of the knee VM, we propose a sub-plan based pruning algorithm called Sub-Plan-based Identification of Knee (SPIK), which requires modifications in the query optimizer.

We have evaluated PIK on a commercial system and found that it often requires processing for only 20% of the total VMs. The efficiency of the algorithm is further increased significantly, by using 10-20% relaxation in response-time. For evaluating SPIK , we prototyped it on an open-source engine – Postgresql 9.3, and also implemented it as Java wrapper program with the commercial engine. Experimentally, the processing done by SPIK is found to be only 40% of the PIK approach.

Therefore, from an overall perspective, this thesis facilitates the desired migration of enterprise databases to cloud platforms, by identifying the VM(s) that offer competitive tradeoffs between money and time for the given query.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has emerged as a new force multiplier in the last decade, and has caused a paradigm shift in the IT industry. Among its many features, on-demand availability and pay-as-you-go scheme are pivotal towards its popularization. Cloud platform provides a wide range of units, comprising of applications, infrastructure, and complex software systems, on a rental basis. Consumers can select resources as per their requirements from a large pool available with the chosen cloud provider, and pay only for the resources selected. Benefits of the pay-as-you-go scheme are achieved best when the selected resources optimize the *monetary expenditure and execution-time* of the user-application.

Relational Database Management Systems (RDBMSs) have to contend with resource-intensive and time-varying workloads, making them promising candidates for migration to cloud platforms. When setup on the cloud, RDBMS would be highly scalable: the hardware configuration of the database server can be modified on the go. Moreover, cloud providers perform complex hardware/software maintenance tasks of database servers, thereby provisioning an easy-to-use interface to consumers. However, to leverage the most from this new infrastructure paradigm, existing database systems need to be modified. Traditional database systems aim at minimizing query response-time, but the pay-as-you-go scheme of the cloud makes monetary expenditure of query execution an additional concern to users. Therefore, we need to modify database systems, such that both query response-time and total money expenditure are considered while query processing, which is the focus of this thesis.

This chapter starts with an introduction of cloud computing and description of relevant terminology. Next, we discuss advantages of migrating RDBMS to cloud platform. Further, we analyze the challenges involved in the shift of RDBMS from the *static* setup to the *on-demand and dynamic* platform of cloud. Finally, we mention the contributions and organization of the thesis.

1.1 An Introduction to Cloud Framework

A well accepted definition for cloud computing is given by U.S. NIST (National Institute of Standards and Technology) [21]: "computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." In other words, cloud computing is an easy to access Internet based model with a large pool of configurable computing resources. It enables easy sharing of computing power among a number of users, satiating demands of each by keeping lower-level details of resource scheduling, etc. oblivious to consumers. Cloud providers assign resources to users as per their requirements, keeping physical resources invisible to them as if obscured in the cloud, and hence the name cloud computing. Pay-as-you-go scheme of this model makes computing resources rentable units, just like our other necessities of water, electricity, gas, etc. It also helps in reducing the upfront infrastructure cost for enterprises and makes computing power an economical resource to reach the masses.

Next, we describe some relevant terminology of cloud computing.

Service models of cloud: Clouds are generally explained as a stack of services. Three most popular service models offered by the cloud providers [6, 7] are shown in Figure 1.1:

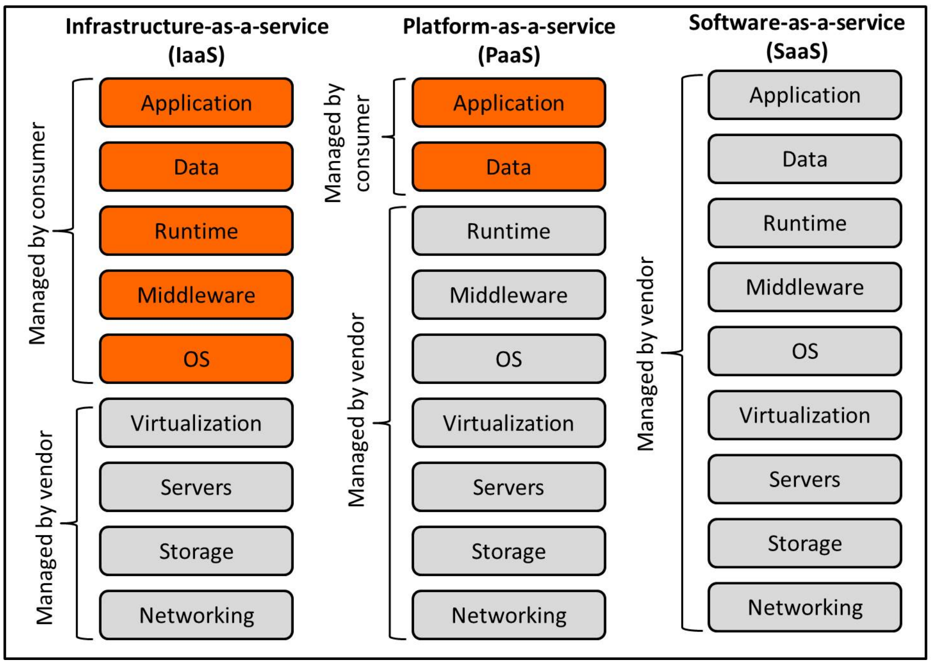


Figure 1.1: Service models of cloud computing

- **Software-as-a-Service (SaaS):** The first in the stack is Software-as-a-service model, providing easy access to the application software. Management of complete infrastructure and required setup is taken care by cloud providers. With pay-as-you-go scheme, users are charged only for the application and the duration of usage. As Figure 1.1 illustrates, an end-user is relieved from the responsibility of configuring and maintaining the system, hence, this model is often referred to as software on demand. Applications using this service model can be easily accessed from around the globe, irrespective of user's location. Additionally any required update or upgrade in any layer of the software stack is taken care by the vendor. Some of the common examples of this model are Gmail, Google docs, Microsoft online services, etc.

- **Platform-as-a-Service (PaaS):** This model provides an accessible environment to develop and provide web-applications without getting into the complexities of buying and managing the required infrastructure. With PaaS, web-developers are free to work on the application development alone and the cloud provider takes care of lower level system details as shown in Figure 1.1. The main difference between SaaS and PaaS service models is that former facilitates hosting of user-applications whereas latter provides the required application development environment. Examples of this service model include Google App engine, Azure service platform, etc.

- **Infrastructure-as-a-Service (IaaS):** Infrastructure-as-a-Service model enables users to choose/construct virtual machines from a large set of system configurations to suit their requirements [2]. These configurations differ in the tradeoffs they provide between processing capability and monetary investment. This model allows consumers to modify their rented resources (RAM size, CPU cores, etc.) as per demands. This particularly helps when number of requests to an application fluctuates with time. With no upfront infrastructure investment and hardware maintenance overheads, IaaS is helping new enterprises to flourish. Examples: Google Compute Engine, Amazon Web Services, etc.

Virtual Machines (VM): The concept of virtualization is used extensively in cloud computing to create an illusion of infinite resources [9]. The mapping of virtual resources to the physical ones enables cloud vendors to provide virtual machines of requested configuration. These VMs might share the underlying resources, but to the user they appear as a single unit. Thus, cloud is a setup of numerous computing resources which are available to end-users as smaller virtual units.

Rental rates and pricing policies: The VM configuration selected by a user determines its rental rate. Rental rate together with pricing policy of vendor, accounts for total monetary expenditure of the user. Some of the most popular pricing policies offered by current cloud providers [2, 5, 6] are per-minute, per-hour, per-month, etc. Policies like per-minute and per-hour are preferred for tasks of shorter duration. However, when the applications are long running, user gains additional monetary savings with per-month pricing. Other attractive policies like sustained discount offers extra savings if the rented resources are heavily used. Moreover, for load sharing at peak hours, consumers can utilize spot or preemptible instances [1, 4]. Instead of fixed rental rate, spot instances are available at bidding, these instances are accessible to the consumer till her bid is highest.

1.2 Benefits for Databases in Cloud

Before migrating to cloud, it is customary to check the offerings of the platform with respect to requirements. Some noteworthy benefits of migrating RDBMSs to the cloud setup are as follows:

- **No maintenance overheads:** Users are freed from maintenance issues of database servers, such as wear-tear of machines or software, hardware/software upgradation, etc. Thus, the overall maintenance expenses, including air-conditioning, power, etc., are taken care of by the cloud providers. Additionally, cloud vendors prevent total loss of data in the events of natural disasters, etc., by data replication across locations. Otherwise, such replication might be a costly affair for enterprises.

- **Scalability:** On-demand availability of resources allows on-the-fly modifications of VM configuration with fluctuations in requirements. This is useful for databases: at peak hours users can rent high end resources, and free them when demand falls. It is expected that such dynamic modifications in configurations would help in overall financial savings.

- **Location independence:** Cloud framework is a web-based model, this makes our data easily available in different physical regions. Data collection, team collaboration, etc., can be easily carried out irrespective of the residential location of data or users.

- **Economical:** Previously, setting up the infrastructure, maintenance of data servers, etc., required huge upfront investment by enterprises. It is reduced to several cents with pay-as-you-go scheme of cloud, making computing resources rent-able units. This model encourages start-up culture, as now infrastructure is available on-demand, and developers have almost any system configuration at their disposal.

To summarize, cloud framework is in vogue because of its flexible, scalable and on-demand nature that is oriented towards self-service and easy management. Databases are good candidates for migration to the cloud as they exhibit varied workload requirements, huge maintenance overheads, and high infrastructural demands. There might be queries that have response-times varying from minutes to hours depending on the underlying system configuration; additionally, individual query requirements may also differ greatly, resulting in poor performance of some queries when a configuration is fixed for the entire workload. Therefore, availability of an on-demand infrastructure is beneficial for setting up a database server.

1.3 Conflicting Objectives for Cloud Databases: Money and Time

Typically, one of the main concerns of end-users on the cloud are money and time required for their query execution. This resulted into a significant amount of work on the optimization of money and time for the database queries on the cloud in recent years [22, 28, 29].

Naturally, to optimize the query response-time we can rent a VM with higher resources. However, as the resources of the VMs increase so do their rental-rates, which eventually increases their total monetary expenditure. The expected behavior of VMs on money versus time space for a given query is shown in Figure 1.2. For the expensive VMs, query response-time is low, but money expenditure is high, similarly, for cheaper VMs required monetary investment is less but the response-time of the query is high.

In essence, total money expended and query response-time are conflicting in nature, and a tradeoff exists between the two objectives. Hence, simultaneous optimization of both is not possible. We will present the empirical evidence of such a behavior on real-world cloud platform with a commercial engine and benchmark queries, in Chapter 4.

Observe that in Figure 1.2, all the Pareto-optimal VMs give lesser value for both the objectives as compared to the dominated VMs. But, among the Pareto-optimal VMs, each VM is lesser in one but greater in other objective, hence, none is better than the other. However, the VM marked as the "knee" in green, balances the two objectives – further decrement in the monetary expenditure comes at the high expense of response-time; and for a minuscule improvement in time, substantial money has to be expended. Therefore, in this thesis we aim at finding the VM with best tradeoff, also referred as the knee VM for the given query. The mathematical notion of domination, Pareto-optimality, knee, etc. is discussed later in Chapter 3.

Note that for this thesis we consider the reservation-based rental policy i.e. the users are charged for the resources reserved irrespective of how much of these resources are actually used. Also, it is one of the most popularly used rental policy by the current cloud providers [3, 5]. Additionally, this thesis targets the scenario only when money and time are conflicting objectives.

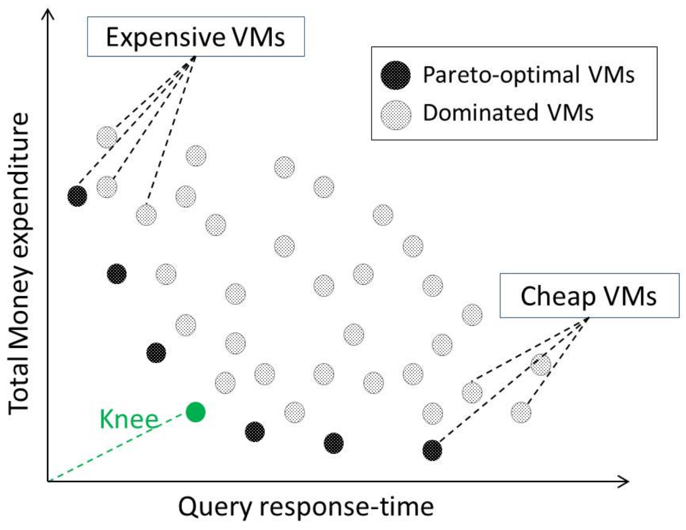


Figure 1.2: Expected money-vs-time profile of a query on the cloud

1.4 Traditional Versus Cloud Query Processing Model

In traditional query optimization, an execution plan with minimum response-time is identified for the given query; underlying system, database schema, etc. are fixed in this setup. Evidently, hardware parameters are treated as constants in such setups. It is justified there, because configuration of the underlying system largely remains the same. Precisely, as given in Equation 1.1, query plan is a function of the given query, database and the underlying system.

$$Query\ Plan = f(Query, Database, System) \tag{1.1}$$

Cloud provides a flexibility of choosing a VM that is best suited for our query requirements, from a large pool of configurations. To know the performance of a query on a number of different VMs, the execution plan of query on all of these VMs are required. One alternative for this is to include system configuration as a variable parameter in query processing. Thus, changing the overall model of query optimization. For this new model, the query plan is a function of only the given query and database as given in Equation 1.2.

$$Query\ Plan_C = f'(Query, Database, Cloud) \tag{1.2}$$

Where, a query plan $Query\ Plan_C$ is identified alongwith a VM, which balances the monetary investment and execution-time of the query. Note that unlike the traditional setups, here the underlying system is not fixed, instead a set of VM configurations available on the cloud platform is considered by the query optimizer. Additionally, the goal of cloud query optimizer is to find the query plan that provides the best tradeoff between monetary investment and response-time.

1.5 Contributions

In this thesis, we consider how the traditional metric used to compare query execution plans, namely response-time, can be augmented to incorporate monetary costs in the decision process. Since, there is a tradeoff between money and time, our goal therefore is to identify the VM configuration that offers the best tradeoff between these two competing considerations.

Database Performance on Real-world Cloud Platforms

To study the performance of modern database engines on available cloud platforms, we test a popular commercial database engine – **ComOpt** [1] on the Google cloud platform for a representative set of queries sourced from the TPC-DS decision-support benchmark. An exemplar money-vs-time plot is shown in Figure 1.3 for Query 55 of the benchmark – in this figure, the red dots denote Pareto-optimal VMs, whereas the one labeled in green is the knee VM.

Some of the notable observations from these experiments include that:

- There are VMs requiring large monetary investments but their query performances are similar to that of substantively cheaper configurations. For instance, in Figure 1.3, the VMs [104 GB RAM, 16 core] and [60 GB RAM, 16 core], both complete the query in around 22 minutes, but the monetary expenditure on the former is close to twice that of the latter.
- There are VMs whose monetary investments are comparable, but the variation in their response times for a query is significant. For example, in Figure 1.3, both the VMs [30 GB, 8 core] and [6 GB, 1 core] entail a payment of 10 cents to complete the query, but the former executes three times faster.
- No single VM acts as the knee across all the queries.

To summarize, our experiments demonstrate that the choice of VM for a user query is a crucial decision, because: (i) variation in money and time across VMs is significant for a given query, (ii) no one VM offers the best money-time tradeoff across all queries.

Identifying the Knee VM

A straightforward approach to identify the knee VM from among a large pool of VM configurations is to exhaustively enumerate the behavior of the individual VMs. However, this can

[1]The name of the database engine is masked for legal reasons.

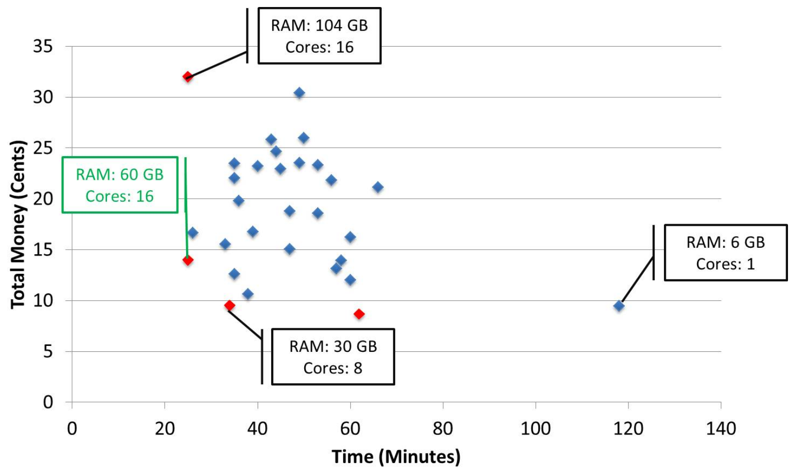


Figure 1.3: Performance of TPC-DS query 55 on the Google cloud platform

prove to be inefficient, especially since the process has to be carried out afresh for each new query, given that the knee VM is query-specific, as highlighted above.

We can do better by observing that the computation power of a VM is a function of its resources, such as RAM size, number of cores, etc. Specifically, query response-time is an *anti-tone* function of the hardware resources, i.e., response-time of a query monotonically decreases with increase in resources. Leveraging this fact, we propose PIK (Plan-based Identification of Knee), which exploits this behavior to efficiently identify the knee VM, without requiring any modifications to the database engine. Specifically, it creates a partial order of the VMs on their resources, and subsequently uses this poset order, on the money-time space to identify the knee VM for the query. To estimate the query response-time for different VMs, it uses the query execution plan available from the database engine's API on the respective VMs.

The empirical evaluation of PIK on the commercial engine indicates that, for most of the queries, only 20% of the total available VMs are processed to identify the knee VM. Further, for most of the queries, the efficiency of the algorithm increases materially when a relaxation factor of 10-20% is provided with respect to the response-time.

Sub-plan-based Identification of the Knee VM

As described above, the PIK algorithm decides the VMs to prune based on information received in the form of complete query plans from the engine's API. However, on observing query plans across VMs, it is often found that sub-plans are repeated. This motivates us to internally modify the *query optimizer* such that VMs can be pruned at a *sub-plan* level also, and increase the efficiency of knee identification.

We propose sub-plan-based pruning algorithm – Sub-Plan-based Identification of Knee (SPIK), wherein, at each sub-plan the pruning mechanism of PIK is applied and only the Pareto-optimal VMs of the node are forwarded to the higher plan-nodes. In a nutshell, by saving only the *Pareto-optimal sub-plans* the computation is reduced compared to the complete plan identification for all the Pareto-optimal VMs. Finally, to ensure that it never misses the knee VM, the dominated VMs are *efficiently* analyzed at the sub-plan levels for dominance.

We have prototyped SPIK inside Postgresql 9.3, and also implemented it as a Java wrapper program with the commercial engine. Our experimental results indicate that the total computation carried out by SPIK is within 40% of PIK approach. Further, the efficiency of the algorithm increases significantly when a relaxation factor of 20 to 30 % is permitted on the time axis.

1.6 Organization

The remainder of this thesis is organized as follows, survey of related work is given in Chapter 2, discussing recent advancements in database query processing to leverage benefits of the cloud platform. Chapter 3 updates the reader on prerequisites, and gives precise problem definition and required notations for the thesis. This is followed by a detailed study of the performance of a popular commercial database engine, **ComOpt** , on the Google cloud platform in Chapter 4. Subsequently, description of the plan-based pruning algorithm –PIK is given in Chapter 5. Following it is the empirical evaluation of PIK on TPC-DS benchmark database in Chapter 6.

To further increase the efficiency of the identification of the knee VM, we propose an algorithm – SPIK to apply pruning at sub-plan levels in Chapter 7. This chapter also discusses experimental results of the algorithm on **ComOpt** as well as on open-source engine – Postgresql 9.3. Finally, the thesis is concluded in Chapter 8, with discussion on some future avenues of the problem.

Chapter 2

Survey of Related Research

Advent of cloud framework encouraged a great deal of activity in database research to utilize benefits of this new platform. Unlike traditional setups where infrastructure is fixed, in cloud environment we can modify the infrastructure as per our requirements. This poses the problem of deciding the best infrastructural setup for given requirements. Additionally, this paradigm shift encourages rethinking of the database query optimizer model. Earlier, additional metrics used in query optimization were system throughput, power, robustness, etc., but when migrated to cloud, financial expenditure also surfaces as an important objective that is largely irrelevant for the traditional setups.

This chapter presents a survey of some recent work that propose approaches to leverage benefits of the cloud for database query processing. Thereafter, we discuss change in overall perspective in database systems with the emergence of cloud technology. We conclude this chapter, with a discussion on the problem of choosing the infrastructure giving best tradeoff between money and time, which is the central interest of this thesis.

2.1 Query Processing for Cloud Platforms

In traditional query processing [25], the objective is minimization of optimizer's cost, which is an approximation of response-time of the query. The plan with minimum estimated cost is identified from the exponential space of query plans, and used for query execution. However, when databases are ported to the cloud, consumers become interested in the minimization of monetary expenditure along with query response-time. Extension of traditional dynamic programming approach of query optimizer to this bi-objective problem is not straightforward. Because, with these additional objectives search space might bloat up, resulting in unreasonable optimization time, or these new objectives violate the basic dynamic programming property used in the query optimizers [17, 28]. Thus, presenting this challenging bi-criteria optimization problem for cloud databases.

Research done in query processing for cloud platforms can be divided into two main themes. One is to use the concepts of multi-objective optimization to consider response-time, money, etc., as additional objectives for query processing; other being the schemes for resource provisioning to minimize overall money and/or time consumption.

2.1.1 Multi-objective Query Optimization (MOQO)

The use of multi-objective optimization techniques is not new in database query optimization. Many approaches were developed for simultaneous optimization of other objectives such as system throughput [17], robustness [8], power-consumption [32] etc. Further, with developments in cloud technology, monetary cost, attained significance and approaches are proposed to include them into optimization.

Violation of Principle of Optimality in MOQO

Ganguly et al. proved in their work concerning query optimization problem to minimize the response-time of the queries with a constraint on the throughput, that multi-objective query optimization with conflicting objectives cannot be solved by a mere conversion to single objective [17]. The reason being pruning of plans rely on the single objective principle of optimality. According to this principle, among two sub-plans for the same output, the one with lower estimated response-time is better. On consolidation of all the objectives into one, this principle breaks down. Now, the sub-plans worse in one objective may be better for the other. Ganguly et al. suggested modifications to keep a set of incomparable sub-plans at each of the plan-node instead of only one optimal plan. Two plans are incomparable if neither is better than the other in all the objectives.

The primary challenge in such multi-objective techniques is that, with an increase in the number of objectives, cardinality of incomparable plans at each node increases, eventually requiring longer optimization times.

Approximation Schemes for MOQO

To increase efficiency of the algorithm, an approximate algorithm employing a user-defined approximation parameter (α_U) was proposed [28]. In this work, a user provides α_U and a weight vector as the inputs. The weight vector signifies the importance of respective objectives for the user. Their algorithm – RTA aims at finding the Pareto-optimal set of plans using the weighted cost of the objectives. Also, they only consider objectives whose cost functions satisfy the Principle Of Near-Optimality (PONO) that is an extension to the principle of optimality. The intuitive meaning of the Principle Of Near-Optimality is that, if the cost of the sub-plan increases by certain percentage then total cost of the plan cannot increase by more than that percentage. They show that this principle holds for the cost functions that uses sum, maximum, minimum, or multiplication by a constant, to calculate the cost of the plan. Observe that, in this thesis we take total money expenditure as the multiplication of query response-time and rental rate of the VM. Since, the rental rate changes with the VM, PONO would not hold for the given set of VMs.

Though it is guaranteed that the obtained plan is sub-optimal within the user-defined approximation factor, it is not ensured if this is the best plan for that cost. Also, it is not advisable to expect from users to know what value of α_U will give the balance between the latency of the algorithm and quality of obtained plans. Additionally, user would not know what approximation would keep the optimization time within desired time-frame, since, time taken by optimization algorithm depends on the size of the resultant set of plans. Furthermore, if the user tries to adapt the cost bounds for the given query, the algorithm starts from scratch for each such invocation.

Therefore, the same set of authors presented its extension to an incremental anytime algorithm, which starts with a coarse solution and keeps on improving the quality of plans with iterations, and reuses previous result for further refinement of the output [29].

Solving MOQO for the Query Workload

The aforementioned approaches give the Pareto-optimal plans for a given query, however, in a more practical setup the requirement would be to balance the two objectives for a given workload [15]. Additionally, instead of whole Pareto-front as the output, if one plan that minimizes both the objectives if possible i.e. *utopia solution* is given, it would suffice for most of the users. Since, money and time are mutually conflicting objectives, utopia point for them would not exist. Hence, they aim at finding the knee solution defined as the solution with minimum Euclidean distance from the utopia point.

They have formulated the problem for distributed databases without replication, where each sub-database is stored on a different and independent VM. The number and configuration of these VMs is variable depending on the number of large tables in the database, etc. The query workload is distributed among the independent copies of the database, depending on the tables in the copy and those required by the query. Now, among many choices of query plans pertaining to the tables and the configuration of the VMs, the one acting as the knee of the Pareto-front is selected. The authors have given a heuristic solution using genetic algorithms for multi-objective optimization to find the set of VMs which would balance the overall money and time for the given workload i.e. the knee solution.

The solution given in [15] cannot be used for the work in this thesis because, they extended the genetic algorithms for multi-objective optimization for their problem. These algorithms focus on finding the desired solution in the infinite space, without considering the total number of evaluated solutions. Since, they used distributed database setup, they expect an almost infinite set of solutions. Next, to evaluate a solution they used a simplified model to estimate response-time of the query. However, for the problem in this thesis we use the centralized setup and the actual cost-model of the database engine to get better estimates, thus, our aim is minimizing the number of optimizer calls without deteriorating the quality of the solution.

2.1.2 Resource Provisioning for Cloud Databases

In IaaS service model, vendors provide pre-configured Virtual Machines (VM) as per the requests of the consumer. These VMs are differentiated by the RAM, CPU cores, clock speed, etc., and users are charged for the resources employed and the duration of usage. In shared nothing and fully replicated database, likely concern is to select a set of heterogeneous VM configuration(s) that satisfies workload requirements with minimization of total financial budget [22]. Their approach take a representative workload that specifies query classes along with query distribution for each class. Server configuration is characterized by a performance vector for each pair of query class and rate of input for that class. Precisely, this performance vector consists of three values: input rate for a given query class, monetary cost per query, and cumulative distribution function of latency for given combination of query class and distribution.

They propose two techniques: *black-box* and *white-box* resource provisioning. The black-box approach profiles performance and monetary cost for different types of VMs for varying input queries, using sample executions. Among the set of VMs in this profile, one with minimum monetary cost under specified latency-bound is selected for database servers. There is a tradeoff between input rate of queries and latency, as input rate increases, financial investment per query goes down because of amortization of money. Similarly, with high input rate, resource contention increases that results in subsequent increase in latency of queries. Therefore, they select a server configuration for each query class and given input rate. In their white-box approach, resources are quantified by a profile on VM configurations. Next, requirements of the representative workload are studied using explain utility of the database optimizer for different configurations. Lastly, these two profiles are fed to constraint-solver of multi-dimensional bin-packing problem. The final output is a set of VMs for the given workload, ensuring each query class gets different server and maintaining user-defined latency bound.

2.2 Our Problem Focus

A summary of similarities and differences between the problem of interest for this thesis and those discussed in this chapter are as follows:

- **System model:** Similar to the works in [15, 22], the service model of cloud used in this work is IaaS. Unlike the other approaches that used replicated database [22], or distributed database [15], we used a centralized database without replication, wherein database is partitioned and all data disks attached to the VM of the desired configuration.

- **Objectives:** Unlike the formulation of Trummer et al. [28, 29], where buffer pool, cache size, number of cores, etc, were taken as different objectives, we consolidated them into one – money; considering importance of monetary investment in recent literature [16]. Therefore, we use only two objectives – money and time required by the user-query.

- **Input:** We solve optimization problem for a given query and not the workload, unlike the problem formulation in [15, 22]. We know that the formulation for workload is more practical and consider it as the future extension of this work.

  We have used an optional user-defined threshold on time, and do not expect any mandatory input as latency bounds [22], approximation parameter [28], or resolution value [29] from the users.

- **Output:** Similar to [15], but dissimilar to the interest of [22, 28, 29] our intended output is a VM configuration which gives the best tradeoff between money and time for the given query.

In short, within the scope of our knowledge there is no direct prior literature for the problem formulated in this thesis, these differences are summarized in Table 2.1. Though many approaches have same objectives for cloud databases but with different aim or for the different input set. However, some have common output but they solved it for a different database or cloud service model. Moreover, we are aiming at knee as the final output instead of the whole Pareto-front, which is considered a better preferred choice in literature [10, 13, 24, 27].

Reference	System Model	Objectives	Input	Output
[15]	Distributed Database	Time and Money	Workload	Knee set of VMs
[22]	Shared Nothing Replicated Database	Time and Money	Representative Workload	A set of VMs
[28]	Centralized Database	Nine objectives (satisfying PONO)	Query, α_U	Pareto-front
This thesis	Centralized Database	Time and Money	Query, λ_T	Knee VM

Table 2.1: Related work at a glance

Chapter 3

Problem Framework

As discussed in the last chapter, this thesis solves the problem of identifying the VM that balances the two objectives– money and time, for the given query. To acquaint the reader with the related terms, we provide a background description for the concepts of bi-objective optimization, Pareto-optimality and knee of the curve. Further, we present details of the framework including service model of cloud, etc. used in this work. Thereafter, we state precise problem definition, followed by the table of notations used in the remainder of this thesis.

3.1 Preliminaries

3.1.1 Bi-objective Optimization Problem

As the name suggests, in a bi-objective optimization problem, two objective functions are to be minimized[1]. The optimization problem statement for the set S $\subseteq \mathbb{R}^n$ of feasible solutions is as follows [13]:

$$\text{Minimize } \{ f_1(x), f_2(x) \}$$
$$\text{subject to } x \in S$$

The vector `x` is formed by `n` decision variables representing the quantities for which values are

[1]The maximization problem can also be dealt in a similar way.

to be chosen in the optimization problem.

3.1.2 Concept of Dominance and Pareto-Optimality

The notion of optimum is changed in bi-objective optimization problem when the two objective functions are conflicting in nature, as the problem studied in this thesis; there the commonly adopted notion of optimum is the one termed as Pareto-optimality. For such instances, the aim is to find a set of compromised solutions rather than a single optimal solution [11, 12, 18]. The related definitions are as follows [18]:

Definition 1 *Pareto-dominance*
A solution $\mathtt{x}$ is said to Pareto-dominate another solution $\mathtt{y}$, denoted by $\mathtt{x} \preceq_p \mathtt{y}$, if and only if

$$\forall \mathrm{i}\ \mathrm{f}_i\ (\mathrm{x}) \leq \mathrm{f}_i\ (\mathrm{y}) \text{ and } \exists \mathrm{j}\ \mathrm{f}_j\ (\mathrm{x}) < \mathrm{f}_j\ (\mathrm{y}) \text{ where i, j} \in \{\ 1,2\ \}$$

i.e. a solution Pareto-dominates another solution, if it either gives lower value in both the objectives, or in atleast one objective when value for other objective is equal.

Definition 2 *Pareto-optimality*
A solution $\mathtt{x}^* \in \mathrm{S}$ is Pareto-optimal if and only if

$$\nexists \mathrm{x} \in \mathrm{S} \text{ such that } \mathrm{x} \preceq_p \mathrm{x}^*$$

The definition says that $\mathtt{x}^*$ is Pareto-optimal if there does not exist any feasible solution which dominates $\mathtt{x}^*$, following Definition 1.

Definition 3 *Pareto-optimal set*
For a given bi-objective optimization problem, Pareto-optimal set $\mathtt{P}^*$ is defined as:

$$\mathrm{P}^* = \{\ \mathrm{x}^* \mid \mathrm{x}^* \in \mathrm{S} \text{ and } \nexists \mathrm{x} \in \mathrm{S}, \text{ such that } \mathrm{x} \preceq_p \mathrm{x}^*\ \}$$

In other words, $\mathtt{P}^*$ is the set of all mutually incomparable Pareto-optimal solutions.

Definition 4 *Pareto-optimal front*
For a given bi-objective optimization problem, Pareto-optimal front $\mathtt{PF}^*$ is defined as:

$$\mathrm{PF}^* = \{\ (\mathrm{f}_1(\mathrm{x}^*), \mathrm{f}_2(\mathrm{x}^*)) \mid \forall\ \mathrm{x}^* \in \mathrm{P}^*\ \}$$

In simpler words, the set of locations of all $\mathtt{x}^* \in \mathtt{P}^*$ in the objective space is termed as Pareto-front. Figure 3.1 shows the objective space, where each axis represents an objective function, and each point in the space denotes a solution. Here, $\mathtt{x}_1$ and $\mathtt{x}_2$ are Pareto-optimal solutions, and dominate all solutions in their respective first quadrants. Particularly, in Figure 3.1, $\mathtt{x}_1$, $\mathtt{x}_2$ $\in$ P* and $(\mathtt{f}_1(\mathtt{x}_1), \mathtt{f}_2(\mathtt{x}_1))$, $(\mathtt{f}_1(\mathtt{x}_2), \mathtt{f}_2(\mathtt{x}_2)) \in$ PF*.

Typically, Pareto-optimal set is given as the solution to the bi-objective optimization problem [11, 13]. Now, it is the task of the user to select one of these solutions as the final choice, depending on her requirements.

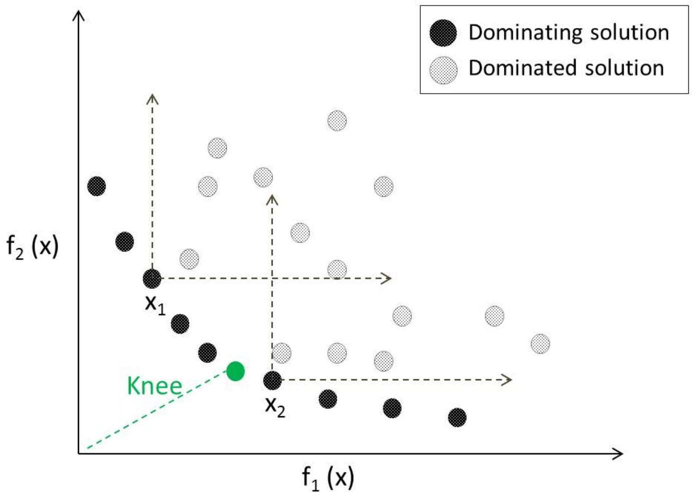


Figure 3.1: Pareto-front and knee solution in objective space

3.1.3 Concept of Knee

Definition 5 *Knee of Pareto-front*

A Pareto-optimal solution $\mathtt{x}_i$ will qualify as the knee if

$$\mathrm{d}_i := \sqrt{f_1(x_i)^2 + f_2(x_i)^2} \quad \text{is minimum among all the feasible solutions.}$$

Here $\mathtt{d}_i$ is the Euclidean distance of the solution $\mathtt{x}_i$ from the origin on the objective space. In Figure 3.1 the point in green color shows the pictorial representation of the knee for the given Pareto-front. On tracing left to right a non-increasing L-shaped curve, the points up to the knee provide significant improvement in $\mathtt{f}_2(\mathtt{x})$ with minuscule degradation of $\mathtt{f}_1(\mathtt{x})$, and the solutions after the knee show tiny improvement of $\mathtt{f}_2(\mathtt{x})$ with a substantial decay in

$\texttt{f}_1\texttt{(x)}$. Thus, intuitively it appears that such a point balances the two objectives. In case, there are more than one solution with equal d_i and it is the minimum value, all of them would be considered as the knee solutions.

Also, it is well argued that knee solution is a commonly preferred choice in the obtained Pareto-front [10, 12, 13, 24]. However, in spite of the established preference for knee solution there does not exist any standard definition for it in the current literature [24]. Based on the application or nature of the targeted Pareto-front, different definitions of knee were used [10, 12, 13, 24, 27]. For the purpose of this thesis we employ the definition used in [27], which is given as Definition 5.

3.2 Problem Formulation

Before we get into the precise problem definition, it is necessary to discuss the service model of the cloud, and other framework related details used in this work.

Infrastructure-as-a-Service model (IaaS): As discussed in Chapter 1, there are a number of different service models like IaaS, PaaS, SaaS, etc. currently available with a number of cloud providers [6, 7]; for this work, we are using IaaS provided by Google that is commonly referred as Google Compute Engine (GCE) [2]. GCE offers a pool of infrastructure in the form of resource parameters, viz., the user can choose number of cores, CPU speed, RAM size, operating system, hard disk size, speed, etc. A combination of these configuration parameters denotes a Virtual Machine (VM) which can be rented for the desired duration. There is flexibility of modifying these parameters as per the requirements. Also, every VM has a fixed rental rate given by Google, which depends on the configuration of the rented VM. Among multiple pricing mechanisms available we used per minute policy.

Monetary cost: Total money charged to the users is calculated using the rental rate of the resources and the duration of rent.

$$Total\ Money\ Expenditure = Time_i\ \times\ Rental\ Rate\ of\ VM_i, \tag{3.1}$$

where $\texttt{VM}_i$ is the VM reserved by the user and Time_i denotes the duration for which she rented it.

Certainly, as we configure a VM with more resources i.e. employ a CPU with higher processing speed, or more number of cores, or with a larger RAM, the associated rental rate of the VM increases. Like most applications, database queries also show a decrease in the query response-time with the increase in resources. Hence, it is likely that as we try to decrease query response-time by increasing resources, the total money required would increase. This leads to the need of minimizing two conflicting objectives – query response-time and total money expenditure.

Variable space: For this optimization problem, the variable space is the set of all the VMs available with the chosen cloud provider. As shown in Figure 3.2, each dimension of this space denotes a resource parameter, and every location in the space is an available VM. A VM Pareto-dominates other in the variable space if it is higher in all the resources than other, which is inverse to the Pareto-dominance in the objective space. For example, in Figure 3.2, all the other black colored VMs are Pareto-optimal, particularly, $\texttt{VM}_1$ and $\texttt{VM}_2$ dominate every VM in their respective third quadrants and are Pareto-optimal. In the remainder of this thesis we use Resource Space (RS) to denote the variable space and total money eXpenditure vs Time Space (XTS) for the objective space. The RS for GCE is given in Figure 3.3, where the dimensions are RAM size and the number of cores, the red points in the plot are the VMs available on GCE. For this work, we are using only two dimensional variable space, however the idea of dominance is extensible to any number of dimensions.

Problem definition: For a given set of Virtual Machines (VM), a query Q, and objective functions $\text{Money}_Q\ (\mathsf{VM}_i)$ and $\text{Time}_Q\ (\mathsf{VM}_i)$, the formal problem statement is

$$\text{Minimize } \{\ \text{Money}_Q\ (\mathsf{VM}_i),\ \text{Time}_Q\ (\mathsf{VM}_i)\ \}$$
$$\text{subject to } \mathsf{VM}_i \in \mathsf{VM}$$

The notation VM stands for the set of all Virtual Machines available with the chosen cloud

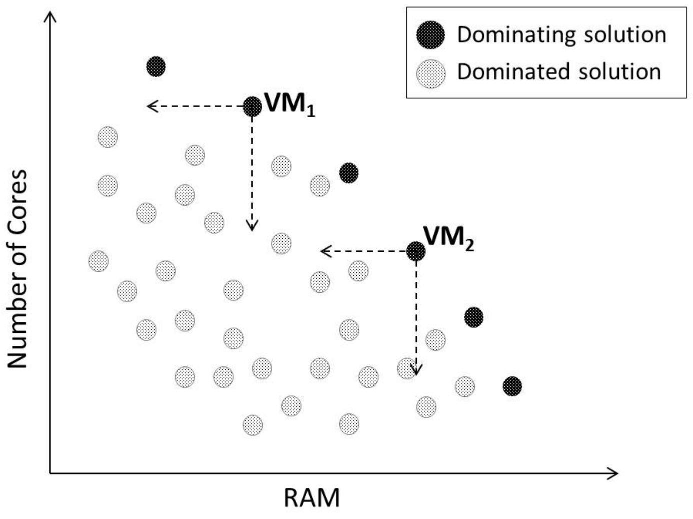


Figure 3.2: Concept of dominance in variable space

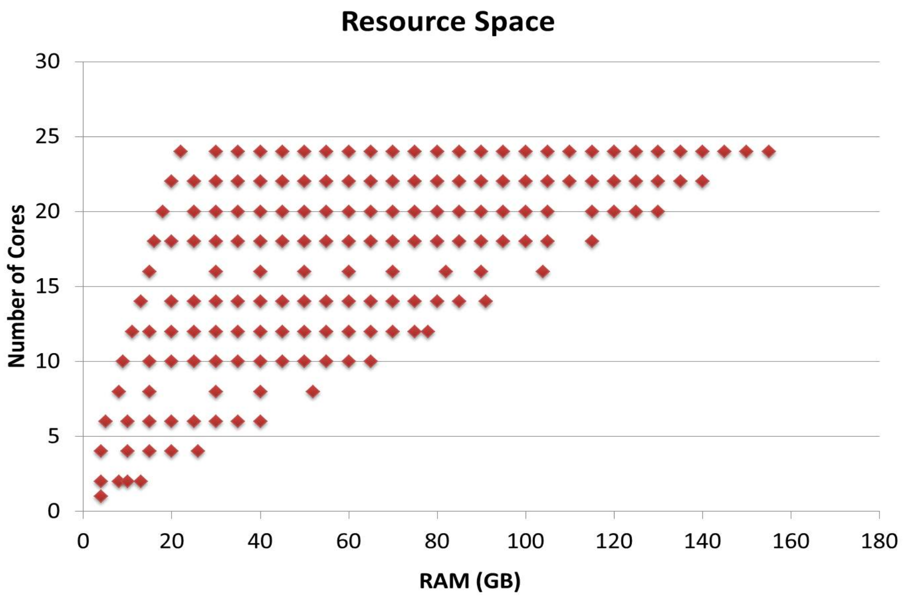


Figure 3.3: Variable space for the VMs available on GCE

provider. Each $\mathsf{VM}_i \in \mathsf{VM}$ is a vector of resources, i.e. $\text{VM}_i = [\text{RAM, Cores, ...}]$.

We know that objectives – time and money are mutually conflicting, hence, their simultaneous minimization is not possible. Therefore, we aim at identifying the knee VM as our final output.

Solution Approximation: Finding an exact knee VM may be computationally expensive [20], consequently, we also consider approximating it by employing a user-defined threshold on time, denoted by λ_T. In this approximation, we ensure that the knee solution satisfies

$$\mathrm{T}_a \leq (1 + \lambda_T)\mathrm{T}_e \text{ and } \mathrm{M}_a \leq \mathrm{M}_e$$

where T_a, M_a denote time and money, respectively, of VM_a which is the approximate knee, i.e when $\lambda_T > 0$. Likewise T_e and M_e represent time and money, respectively, for the VM_e which is the exact knee, i.e., with $\lambda_T = 0$.

Note that we are applying the threshold on time and not on money, this is because we can estimate the response-time of a query using the respective VM configuration. In short, if a VM is stronger than the other, then we know that the response-time of the query on stronger VM is upper-bounded by the weaker VM. However, the total money expenditure of the VMs cannot be estimated by their configurations alone. Since, total monetary cost is the product of rental-rate of the VM and its query response-time, therefore, total money expenditure of a cheaper VM may be more than its expensive counterpart. For example: $\mathrm{RR}_i = 1$, $\mathrm{RR}_j = 2$, $\mathrm{T}_i = 25$ and $\mathrm{T}_j = 10$, hence, $\mathrm{X}_i = 25$ and $\mathrm{X}_j = 20$. Here, although RR_i is lesser than VM_j, but the total money expenditure of VM_i is greater than VM_j.

Identifying Pareto-optimal Solutions: The techniques proposed in this thesis are extensible to output the Pareto-optimal VMs, in case user wants to know all the VMs in the Pareto-front of the query.

Time or Money Budgeted Solution: If the user wants her query to finish within a specified time and/or money budget, the extensions of our proposed technique to include these budgets are discussed later in the thesis.

3.3 Notations

Before we delve into the details of the algorithms for the identification of the knee VM in subsequent chapters, the notations used in the remainder of the thesis are given in Table 3.1:

Notations	
VM	Virtual Machine
Q	Given query
λ_T	User-defined threshold on time
VM	The set of Virtual Machines
R	The set of Resources ={ RAM, Cores, CPUSpeed, ...}
r_i	ith element of R
VM_i	ith element of VM , VM_i = [r_1, r_2, ...]
r_{ij}	jth resource of VM_i
T_i	Response time of Q on VM_i
X_i	Total Money expenditure for Q on VM_i
RR_i	Rental Rate of VM_i
XTS	Money eXpenditure versus Time space
RS	Resource Space
PPOS	Potential Pareto-Optimal Set
POS	Pareto-Optimal Set

Table 3.1: Notations used in this thesis

Chapter 4

Database Performance on the Cloud Platform

In this chapter, we discuss the performance of a popular commercial database engine **ComOpt** on Google cloud platform. We present our experiments on the standard decision-support benchmark database – TPC-DS, with its default size of 100 GB.

In our execution-time experiments, query-level details like variations in response-times and total money expenditure with changes in RAM size and number of cores is analyzed for a number of queries. Furthermore, we show experimentally that there is no one VM that acts as knee for all of the queries. We found that the knee VM for one query is a costlier and/or slower alternative for other queries compared to their respective knee VMs. These observations highlight the importance of selecting the right VM for a given query, cloud platform, and database engine. Additionally, the effects of these hardware related parameters on the different query-operators such as scan, join, etc. are studied in compile-time experiments.

Note that, the magnitude of money and time values could create a bias. For example, if the money is in the range of 1-10 cents and time in the range of 50-300 minutes, then the VM with lower response-time but higher monetary expenditure is likely to be selected as the knee. Therefore, to give equal importance to both the objectives we normalize the axes using the feature scaling technique - $\frac{X_i - X_{min}}{X_{max} - X_{min}}$, this brings all the values in the range [0,1] for both the axes. Specifically, actual units of money and time are there just to present a better understanding of the situation, like the range of variation in these values, etc., but the identification of the knee VM is carried out on the normalized axes.

4.1 Experimental Setup

This section gives a detailed description of the experimental framework comprising of cloud platform, database engine, and the database used in this work.

4.1.1 Cloud Platform Details

As discussed in the previous chapter, we are using IaaS model of cloud available as Google Compute Engine (GCE) [2]. While there are a number of options available for configuring a VM, we performed our experiments varying RAM size and number of cores only.

The pricing model we used for these experiments is per-minute, where all machines are charged a minimum of 10 minutes by Google. However, for simplification we discard the minimum quanta of 10 minutes. So, if a query runs for 2 minutes, we calculated the total money expenditure for 2 minutes only and not 10 minutes. The details of the per-minute price of VMs is available at [3].

On GCE, for a fixed number of cores there is a window within which RAM size can be varied. For instance if the number of cores is 4, then RAM could be any integer value between 4 GB to 26 GB. We maintained a granularity of 5 GB while configuring the VMs. Thus if core size is 4 then RAM sizes used are – 4 GB, 10 GB, 15 GB, 20 GB and 26 GB. Table 4.1 gives details of the combinations for configuring a VM, with total VMs used in these experiments summing to 186.

While performing the experiments, one base disk is kept which has necessary software including operating system and **ComOpt** installed on it. For creating a VM, first a combination of RAM and core size is selected from the provided cloud interface. Next, the base disk and the other disks containing the database are attached to this newly created VM.

Number of cores	Range of RAM size	Number of VMs
1	4-6	1
2	4-13	3
4	4-26	5
6	5-39	8
8	7-52	10
10	10-65	12
12	11-78	15
14	13-91	17
16	14-104	19
18	16-117	21
20	18-130	23
22	20-143	25
24	22-156	27
Total number of VMs = 186		

Table 4.1: VMs available on GCE

4.1.2 Database and DBMS Setup

We used 100 GB TPC-DS database which is partitioned across four disks, with partitioning handled by **ComOpt** itself, after specifying the disk and the partition size on each disk. Each of these disks are in the same region as the base disk and the VM.

Physical design: The experiments are performed on two physical schema of TPC-DS database – **Default Index(DI)** and **All Index (AI)**. In DI configuration, the default physical design of the database is available with clustered index on each primary key. AI, on the other hand is an index rich schema with indices available on every column, along with the clustered index on each primary key.

Query descriptors: In this chapter as well as in the remainder of this thesis, the queries are denoted in the format x-DSQ-n. Here, x stands for the physical design of the database, and n for the query number in the benchmark.

Interaction with database engine: To ensure that **ComOpt** reflects the configuration parameters of the chosen VM, system values of hardware related parameters based on the VM configuration are updated. After manually changing the values of these parameters in the system file, **ComOpt** is restarted so that it uses the updated values. Next, using auto tuner utility of **ComOpt**, different parameters such as database-memory, sort-memory, degree of parallelism, buffer size, etc. are set to their recommended values. Thereafter, the queries are executed sequentially with no other process running on the machine. Moreover, to ensure cold-cache environment the DBMS and OS cache are cleared after each query execution.

4.2 Empirical Results

We experimented on a number of TPC-DS queries for both execution-time as well as compile-time query processing. The observations regarding the knee VM, variations in time and money, query plans etc. are discussed next.

4.2.1 Execution Time Experiments

We executed a number of TPC-DS queries on the VMs available on GCE and profiled them on the XTS (money eXpenditure vs Time Space), with time on x-axis and money on y-axis. Because of long response-times of queries particularly on low-end machines, we could not run these queries on all of the 186 VMs. Instead, for each core size (2, 4, 8, etc.), we used the VMs with minimum and maximum RAM size available for that core number, as per GCE specifications given in Table 4.1. For the core size where the range of RAM size is large, we used some in between VMs also to cover the spectrum of the VMs.

The XTS plots of a few TPC-DS queries are shown in Figures 4.1, 4.2, 4.3, 4.4, and 4.5. Each of the points on these graphs represents a VM, the red points denote the Pareto-optimal VMs, and the knee VM is labeled green.

Significant variation in time and money across VMs: It is evident from these plots that the variation in time and money is substantial with the changes in the configurations of VM. For example, DI-DSQ-52 can be completed in 40 minutes on [60 GB, 16 core] but if run on [6 GB, 1 core] it would take several hours to complete. Similarly, AI-DSQ-24 completes in 30 minutes on [65 GB, 10 core], however it would require 90 minutes when the VM is [10 GB RAM, 4 core]. These variations in the response-time with the VM configurations are due to the resource requirement of the query plan.

Expensive but slow VMs: These plots also show that there are certain VMs which are expensive but do not improve response-time of the query. For example in Figure 4.1, the response-time of DI-DSQ-52 is similar on [104 GB, 16 core] and [60 GB, 16 core], but the former VM is twice as expensive as the latter. Thus, once the resources required by the query plan are attained, any further increase in resources does not decrease query response-time significantly. Similarly, there are VMs which require similar monetary investment but give materially different response-times. For instance in Figure 4.2 both [30 GB, 8 core] and [6 GB, 1 core] require around 10 cents to run DI-DSQ-55, but the former executes three times faster. Hence, renting an expensive VM may not decrease the response-time.

Knee VM is query specific: Another observation is that the knee VM is query specific. For queries DI-DSQ-52 (Figure 4.1) and DI-DSQ-55 (Figure 4.2), the VM [60 GB, 16 core] acts as the knee, but the same VM is around two times costlier compared to the knee VM of DI-DSQ-71 (Figure 4.3). Similarly, the knee VM of AI-DSQ-67 (Figure 4.5) is approximately two times slower to the knee VM of AI-DSQ-24 (Figure 4.4). Hence, there is no one VM that can be the knee for all of the queries.

In short, selecting the right VM for running the given user query is of paramount importance, otherwise one might end up paying too much in time and/or in money.

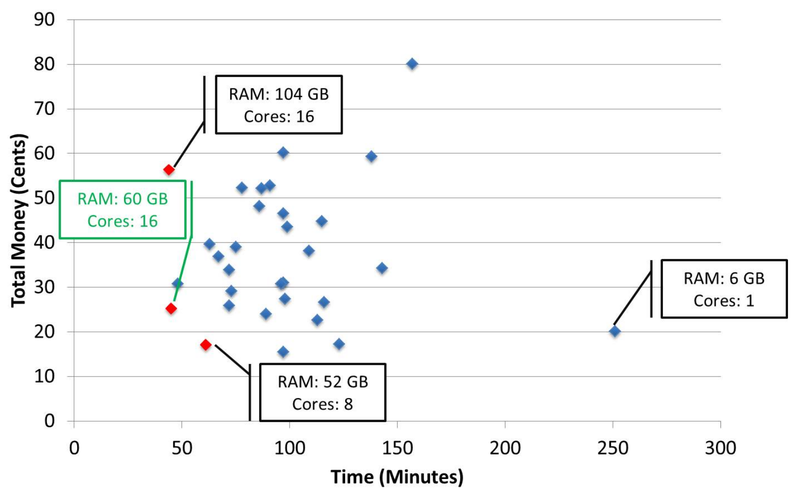


Figure 4.1: Execution time plot of DI-DSQ-52

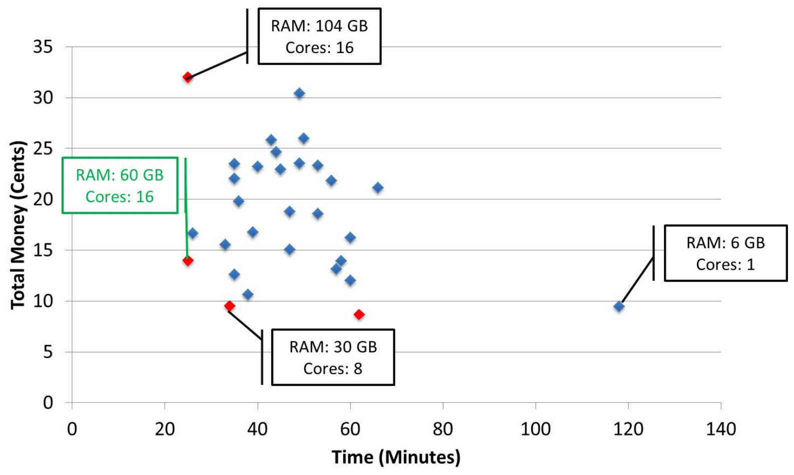


Figure 4.2: Execution time plot of DI-DSQ-55

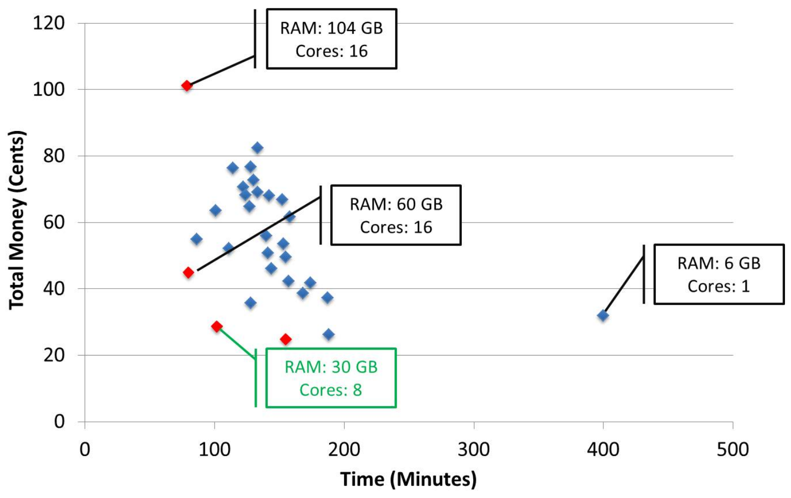


Figure 4.3: Execution time plot of DI-DSQ-71

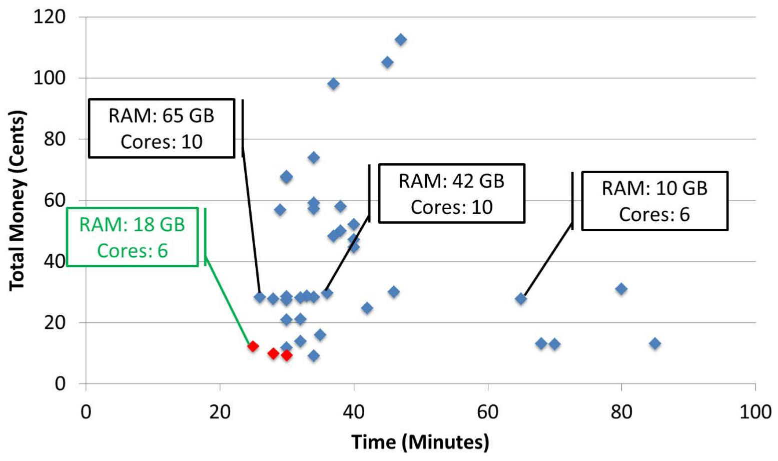


Figure 4.4: Execution time plot of AI-DSQ-24

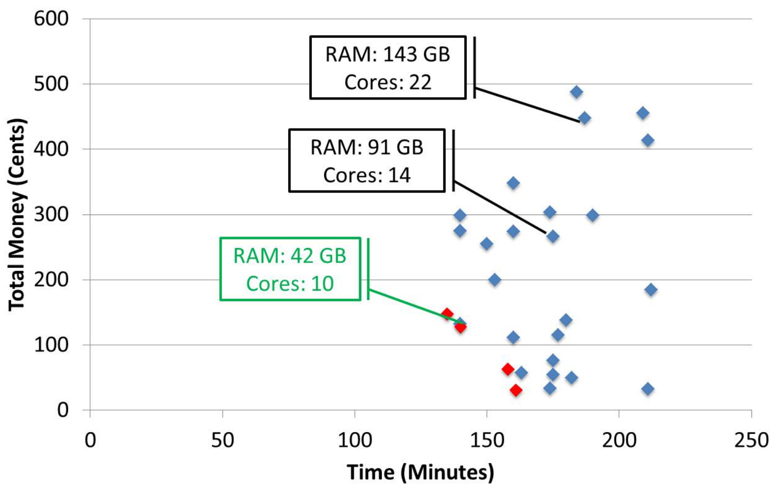


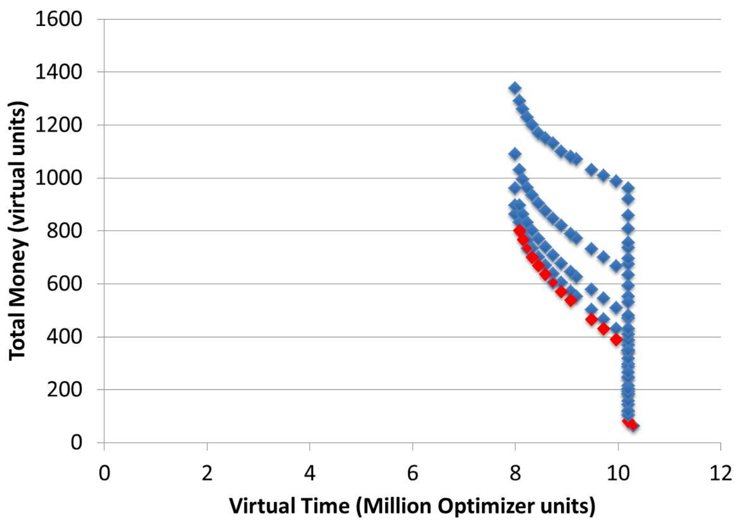
Figure 4.5: Execution time plot of AI-DSQ-67

4.2.2 Compile Time Experiments

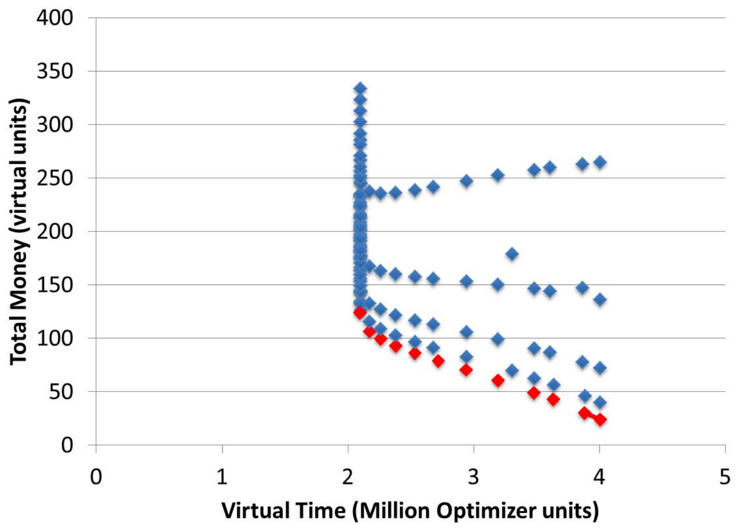
To analyze the performance of the queries on all of the available machines, we use compile-time plots for pragmatic reasons. We understand that compile-time plots do not give accurate approximation of execution-time performance, but the recent works on bridging the gap between the two, provides the hope of having similar compile-time and execution-time plots in coming years [30, 31].

For these experiments, explain utility of **ComOpt** is used to obtain the optimal query plan for each VM, and we use plan cost as the estimated response-time. The compile-time plots for a few benchmark queries are given in Figure 4.6 and Figure 4.7. In these plots, x-axis represents optimizer's estimated time and y-axis is the calculated total money. The VMs on the Pareto-front of the query are shown as red dots and the dominated ones are in blue.

Effects of configuration on the query-operators: On observing the behavior of individual operators with changes in configuration, we found that with every small increase in per node memory, cost of nested loop join is reduced; however cost of hash join decreases only when per node memory is doubled.



(a) Compile time plot of DI-DSQ-6

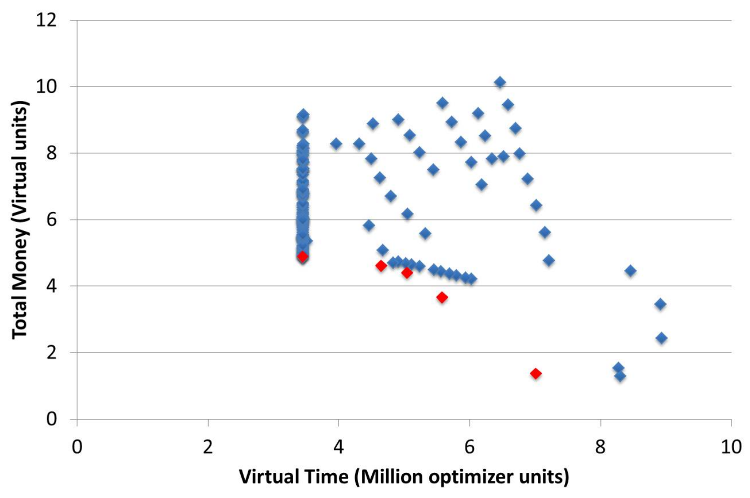


(b) Compile time plot of DI-DSQ-71

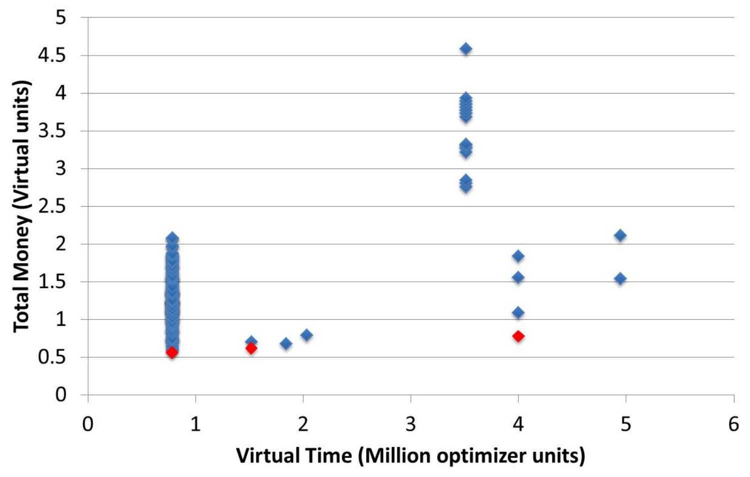
Figure 4.6: Compile time performance of TPC-DS queries on DI

Similarly, cost of parallel operators in plans decreases with added parallelism and gives the arc like patterns exhibited by queries given in Figure 4.6, where each arc corresponds to a different core size.

It is visible from Figure 4.8 and Figure 4.9 that along with join algorithms, join orders and degree of parallelism also change with per node memory. The reason being that when available



(a) Compile time plot of AI-DSQ-24



(b) Compile time plot of AI-DSQ-59

Figure 4.7: Compile time performance of TPC-DS queries on AI

memory is low, indexed nested loop or sort-merge join with index is cheap; however, hash join is preferred when available memory is high.

The general behavior of the query-operators is summarized in Table 4.2. The optimizer's cost for Hash-join and unclustered indexes is reduced when the size of node per memory is doubled. However, sort and nested-loop joins show a smooth variation with every small increase in available memory. Similarly, the change in the costs of parallel operators is smooth with the changes in core-size of the system.

Parameter	Behavior	Operators
Memory	Bursty	Hash-join, Unclustered indexes
	Smooth	Sort, Nested-loop join, Set operators
Core-size	Smooth	Parallel join, Parallel scan

Table 4.2: General behavior of operators with variations in hardware parameters

From execution-time to compile-time plots: Some notable observations pertaining to the comparison of compile-time and execution-time plots include: Pareto-fronts in compile-time plots are nearly linear and difference in their extreme virtual times is also low. However, Pareto-fronts of execution-time plots are more like L-shaped and the difference in extreme values is large. The number of elements in the Pareto-front of execution-time plots is lesser than the number of Pareto-optimal solutions in compile-time plots. These differences in the plots are owed to the cost model of **ComOpt** query optimizer. The point to note is that the definition of knee (Definition 5) we are using in this thesis, gives the solution with best tradeoff for any L-shaped Pareto-front. Therefore, although we are using compile-time plots to evaluate our approaches, later in the thesis, it is only for practical concerns. The techniques will work fine even with execution-time plots or with compile-time plots that are accurate approximation of the execution-time performance.

4.2.3 Conclusions

In this chapter, we discussed the effects of VM configurations on the query plans, response-time, and the monetary expenditure for a commercial database engine on an actual cloud platform. The gist of these experiments is as follows:

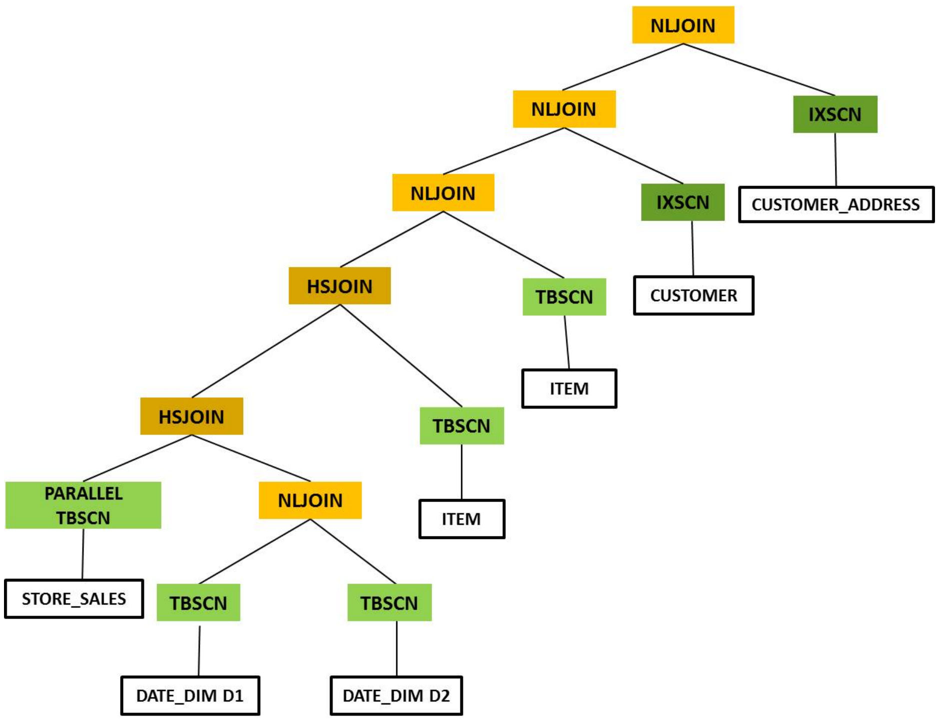
- Renting an expensive VM may not decrease the response-time of the query. Specifically, there may be a cheaper alternative with similar query response-time.

- For a given query, the total money expended by two VMs may be similar even if their configuration and query response-times are significantly different. For example, the total money expenditure for $\texttt{VM}_i$ and $\texttt{VM}_j$ might be similar, i.e.,

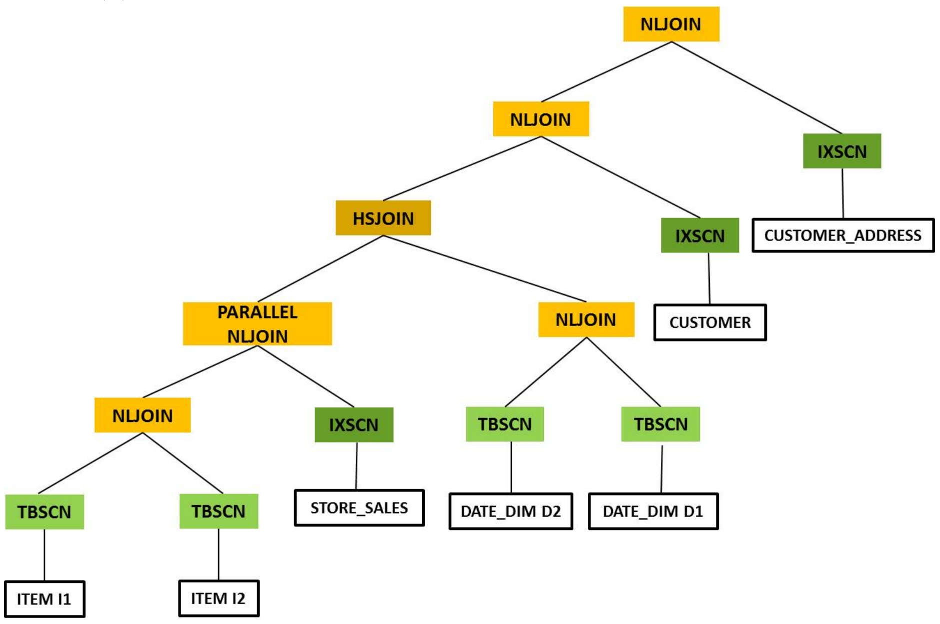
  $$RR_i \times T_i \approx RR_j \times T_j \tag{4.1}$$

  Where, $\texttt{VM}_j$ is a richer VM than $\texttt{VM}_i$, with $\texttt{RR}_i < \texttt{RR}_j$ and $\texttt{T}_i > \texttt{T}_j$.

- The query plans may or may not change with the alterations in VM configuration depending on the query requirements.

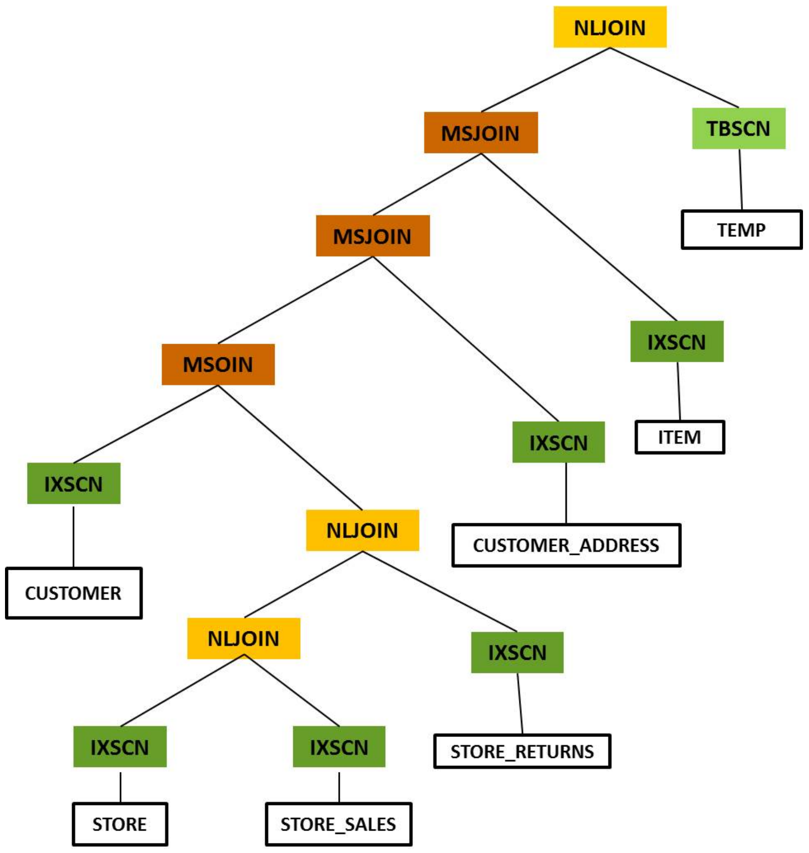


(a) Query plan for DI-DSQ-6 on VMs with low parallelism

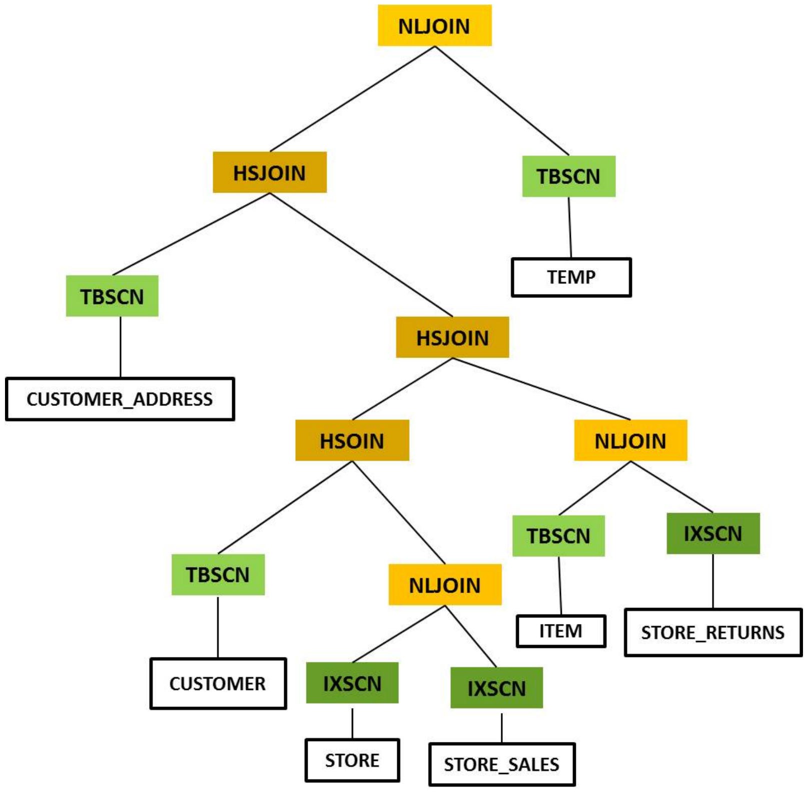


(b) Query plan for DI-DSQ-6 on VMs with high parallelism

Figure 4.8: Plan structure for DI-DSQ-6 on different VMs



(a) Query plan for AI-DSQ-24 on VM with 4GB RAM and no parallelism



(b) Query plan for AI-DSQ-24 on VM with 21GB RAM and parallelism

Figure 4.9: Plans of AI-DSQ-24 on different VMs

Chapter 5

A Plan-based Approach to Identify the Knee VM

In the previous chapter, we have seen that the response-times of the queries vary significantly with the change in VMs, and so does the total money. Thus, the choice of VM for a given query and database system is a crucial step, while migrating to the cloud environment. A straightforward approach to identify the knee VM from among a large pool of VM configurations is to exhaustively enumerate the behavior of the individual VMs. However, this can prove to be inefficient, especially since the process has to be carried out afresh for each new query, given that the knee VM is query-specific, as highlighted in the previous chapter. We can do better if the VMs can be compared among themselves for query performance.

Configurations of the VMs and query response-times: Observe that the computation power of a VM is a function of its resources, such as RAM size, number of cores, etc. Let there be $\texttt{VM}_w$ and $\texttt{VM}_s$, where $\texttt{VM}_w \preceq_p \texttt{VM}_s$ in RS (Resource Space) with their corresponding query response-time be $\texttt{T}_w$ and $\texttt{T}_s$ respectively. Now, every $\texttt{VM}_i$ bounded by these two VMs in RS would have query response-time $\texttt{T}_i \in \{ \texttt{T}_x \mid \texttt{T}_s \leq \texttt{T}_x \leq \texttt{T}_w \}$. In other words, the response-time of the query for a VM is within the range of response-times of its bounding VMs.

Specifically, query response-time is an antitone function of the hardware resources, i.e., response-time of the query shows monotonic decrease with the increase in resources. We exploit this behavior by ordering the VMs on RS and subsequently using this order to locate them on XTS (money eXpenditure versus Time Space) to get the knee VM for the query.

Notion of similarity between VMs on query response-times: For a given query $\mathtt{Q}$ and the VMs $\mathtt{VM}_w$, $\mathtt{VM}_s$ such that $\mathtt{VM}_w \preceq_p \mathtt{VM}_s$. The response-times of $\mathtt{Q}$ on $\mathtt{VM}_w$, $\mathtt{VM}_s$ are said to be similar, if

$$\mathtt{T_w} \leq \mathtt{T_s} + \epsilon \text{ i.e. } \frac{\mathtt{T_s}}{\mathtt{T_w}} \approxeq 1$$

where ϵ is a very small number, and $\mathtt{T}_w$ and $\mathtt{T}_s$ are the response-times of $\mathtt{Q}$ on $\mathtt{VM}_w$ and $\mathtt{VM}_s$ respectively.

Algorithm for efficient identification of knee VM: We propose the algorithm – PIK (Plan-based Identification of Knee), to identify the knee VM for the given query and the set of VMs. The algorithm uses partial ordering on the set of VMs to locate a VM on XTS. It evaluates the query execution plans on the minimal and maximal VMs of each poset for their estimated query response-times – if the response-times are estimated to be similar, then all the VMs bounded by these extreme VMs are pruned. Otherwise, the already processed VMs are set aside, and the minimal and maximal VMs of the remaining unprocessed VMs are evaluated for their response-times. Finally, the knee VM is identified from the processed VMs as the one with the minimum Euclidean distance from the origin on the money-time space. To estimate the query response-time for different VMs, the query execution plan available from the database engine's API is used.

Later, we theoretically prove that PIK always identifies the knee VM; further, if it is acceptable to find a "near-optimal" knee by providing a relaxation-factor on the response-time distance from the optimal knee, then PIK is also capable of finding even more efficiently a satisfactory knee under these relaxed conditions.

Note that we are identifying the knee VM at the compile-time and we have an additional assumption that the resources available at run-time to be at least commensurate with that expected at compile-time.

5.1 Partial Order on the Virtual Machines in RS

A virtual machine is represented as a tuple of resources, e.g. $\text{VM}_i = [\text{r}_1, \text{r}_2, ...]$. To compare different VMs and establish a relation between them, we define the relation $\preceq$ on the set of VMs. Later, we prove that this relation forms a poset on the set of available VMs, denoted by VM.

Weaker and stronger VM: For VM_i, $\text{VM}_j \in \mathsf{VM}$, VM_i is the weaker VM if

$$\forall\ \text{r}_s \in \mathsf{R},\ \text{r}_{is} \leq \text{r}_{js} \text{ and } \exists\ \text{r}_t \in \mathsf{R},\ \text{r}_{it} < \text{r}_{jt}$$

E.g. for $\mathsf{R}= \{\text{RAM, Cores}\}$, if $\text{RAM}_i \leq \text{RAM}_j$ and $\text{Cores}_i < \text{Cores}_j$ then $\text{VM}_i \preceq \text{VM}_j$. Similarly, VM_i is referred as the stronger VM if

$$\forall\ \text{r}_s \in \mathsf{R}\ \ \text{r}_{is} \geq \text{r}_{js} \text{ and } \exists\ \text{r}_t \in \mathsf{R},\ \text{r}_{it} > \text{r}_{jt}$$

In short, if $\text{VM}_i \preceq \text{VM}_j$ then $\text{VM}_j \succeq \text{VM}_i$.

Incomparable VM: If VM_i and VM_j are such that neither is weaker or stronger than the other, then they are incomparable.

Bounded VM: $\text{VM}_k \in \mathsf{VM}$, is said to be bounded by VM_i and VM_j with $\text{VM}_i \preceq \text{VM}_j$ if,

$$\text{VM}_i \preceq \text{VM}_k \preceq \text{VM}_j$$

Figure 5.1 shows the notion and corresponding locations of weak and strong VMs pictorially. For the given machine $\texttt{VM}_1 \in \mathsf{VM}$, all $\text{VM}_j \succeq \texttt{VM}_1$ will fall in the region labeled as stronger VMs. Similarly, all $\text{VM}_j \preceq \texttt{VM}_1$ will fall in the region marked as weaker VMs and rest of the VMs are incomparable to $\texttt{VM}_1$.

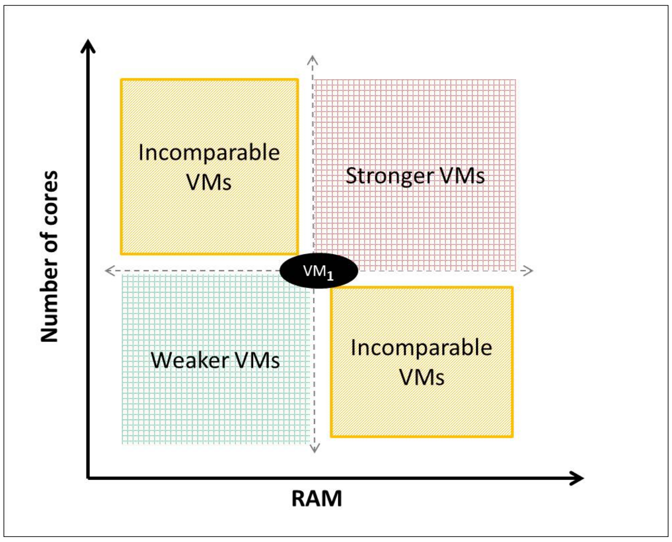


Figure 5.1: Concept of weaker and stronger VMs than a given VM

Claim: *Any set of VMs with relation $\preceq$ forms a poset.*

Proof: Each property of a poset on the set of VMs for relation $\preceq$ can be proved as follows. For virtual machines VM_i, VM_j, $VM_k \in \mathsf{VM}$ and $\forall$ $r_s \in \mathsf{R}$,

Reflexivity: We know that $\forall$ r_s, $r_{is} \leq r_{is}$ therefore, $VM_i \preceq VM_i$. Hence, $VM_i \preceq VM_i$.

Antisymmetry: If $VM_i \preceq VM_j$ and $VM_j \preceq VM_i$, then it implies that $\forall$ r_s, $r_{is} \leq r_{js}$ and $r_{js} \leq r_{is}$. It can only be possible if $r_{is} = r_{js}$. Thus, $VM_i = VM_j$.

Transitivity: If $VM_i \preceq VM_j$ and $VM_j \preceq VM_k$, then $\forall$ r_s, $r_{is} \leq r_{js}$ and $r_{js} \leq r_{ks}$, which implies that $r_{is} \leq r_{ks}$, thus $VM_i \preceq VM_k$.

Hence, $\preceq$ forms the poset on VM .

Hasse diagram of the VMs available on GCE is given in Figure 5.2. It shows that for the VMs available on GCE as per Table 4.1, the minimal VM is [4 GB, 1 core] and the maximal VM is [156 GB, 24 core] for the entire set of VMs. Also, there are certain VMs which are incomparable among themselves like [4 GB, 2 core], [6 GB, 1 core] at the first level and [10 GB, 2 core], [4 GB, 4 core] at the second level, so on and so forth.

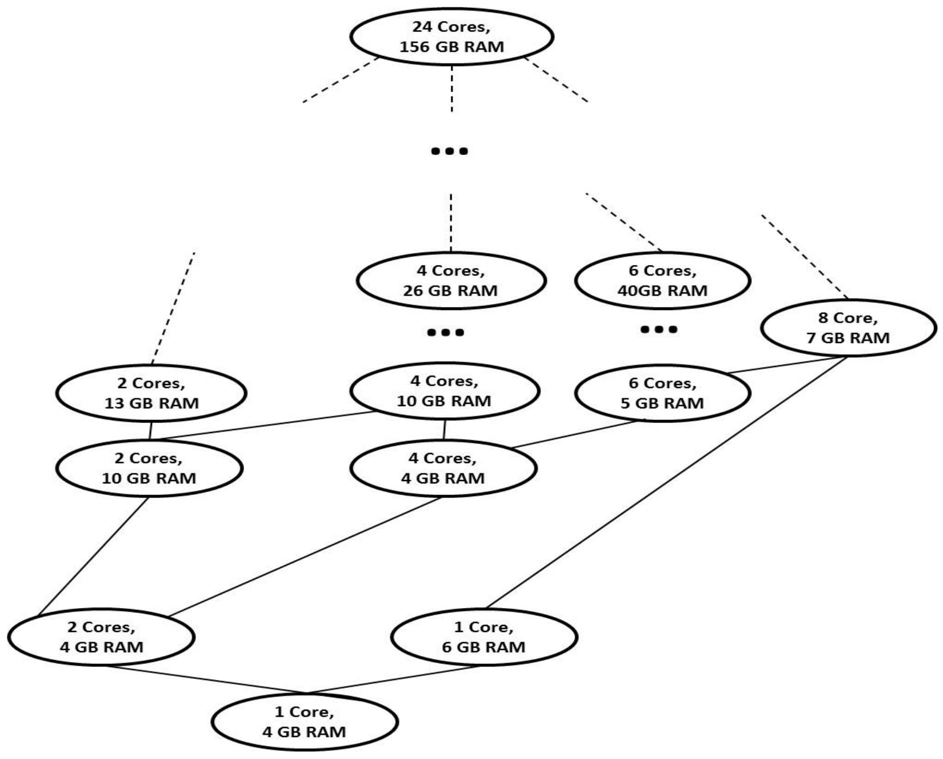


Figure 5.2: Hasse diagram of the VMs available on GCE

5.2 Locating Virtual Machines on XTS

In this section, we will see how to use the aforementioned partial ordering on the VMs to locate them on the XTS. From the previous section, we know that $\mathtt{VM}_i \preceq \mathtt{VM}_j$ implies that $\mathtt{VM}_j$ has atleast as much resources as $\mathtt{VM}_i$. Now, either the query benefits from these additional resources of $\mathtt{VM}_j$, or these are superfluous to the requirements of the query, which gives no further improvement in the performance of the query on $\mathtt{VM}_j$. Therefore, we can say that the response-time of the query on $\mathtt{VM}_j$ would be atmost that on $\mathtt{VM}_i$ i.e. $\mathtt{T}_j \leq \mathtt{T}_i$.

Now, using the response-time of the query on a VM (say $\mathtt{VM}_i$), we can calculate the total money required by the query on $\mathtt{VM}_i$ and can locate it on XTS. Once the location of $\mathtt{VM}_i$ on XTS is known, following can be concluded:

1. From the antitonic nature of response-time with resources, we know that all the $\mathtt{VM}_j \succeq \mathtt{VM}_i$

will have $T_j \leq T_i$. Therefore, all such VM_j will lie in the second or the third quadrant of VM_i on the XTS. Figure 5.3 enumerates this pictorially, for the given location of VM_1 on XTS, all the stronger VMs are to the left of the VM_1, shown in red color.

2. Similarly, for all the $VM_j \preceq VM_i$, $T_j \geq T_i$, thus, all such VM_j will be in the first or fourth quadrant of VM_i. In Figure 5.3, blue colored region shows the location of all the VMs weaker than VM_1.

Observe that we are making this conclusion by comparing the VMs on RS, hence, we cannot conclude anything for the VMs incomparable on RS . Although rental-rates of VMs would be comparable for all VMs, but total money expenditure depends on both the query response-time and the rental rate of the VM. Particularly, a VM with lesser rental-rate but higher response-time may require more money consumption than the one with higher rental rate but lesser response-time and vice-versa. Thus, the domination of VMs on RS does not always follow on the XTS.

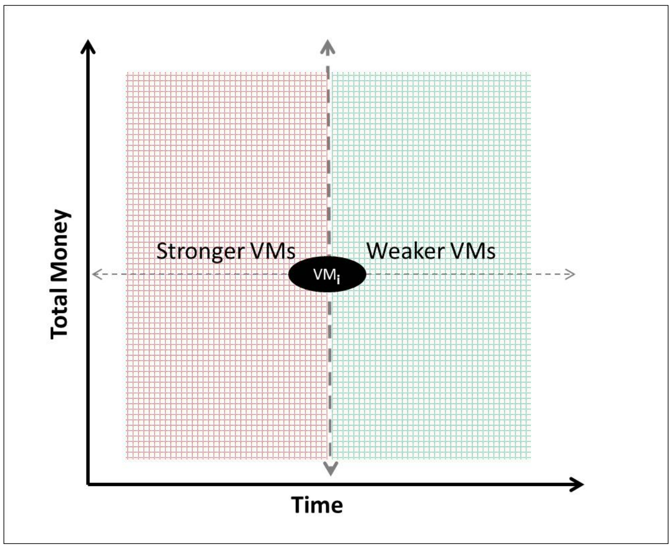


Figure 5.3: Regions for the stronger/weaker VMs on the XTS for a given VM

5.3 Plan-based Identification of Knee (PIK)

This section describes the algorithm to *pick* the knee VM among hundreds of VMs, for the given query. The algorithm is divided into three steps; firstly, the VMs are arranged in poset order of their resources. This is a one time preprocessing and does not change with the query or the database engine used. Secondly, using this arrangement of VMs, they are processed for their estimated query response-times. Lastly, among the processed VMs, the knee VM is characterized as the one with minimum Euclidean distance from origin on the XTS.

5.3.1 Preprocessing

In this step, we arrange all the VMs in the poset order of their resources. This is done by a simple algorithm for sorting ordered pairs, with each resource parameter taken at an iteration of sorting. For example, if R= {Core, RAM}, at first iteration take r_0 = Core and sort all the VMs on their core size. Subsequently, take r_1 = RAM as the next sort parameter. The second sorting iteration maintains the sort of previous iteration and sorts on r_1, i.e., all the VMs with same core size are now sorted on their RAM sizes. The order in which sort parameters are selected does not matter. Note that this phase only depends on the configuration of the VMs available on the cloud platform and not on the query, database, or database engine.

The sorting of this kind would ease in the subsequent identification of minimal and maximal elements, while identifying the knee VM. As shown in Figure 5.2, the bottom-most VM is the minimal of the poset and is least in resources, similarly top-most VM is the maximal.

5.3.2 Identifying Potential Pareto-optimal VMs

This step creates the PPOS (Potential Pareto-Optimal Set) of VMs, one among them is the knee VM of the query. It processes the minimal and maximal VMs of the poset (VM, $\preceq$), and compares their query response-times. If the response-times are similar then all the VMs bounded by them are pruned. Otherwise, identify the next pair of minimal and maximal VMs from the remaining unprocessed VMs and process them. This continues till we get a pair of VMs with similar response-times or the number of VMs is exhausted.

The reason for pruning the sandwiched VMs without worrying about loosing the knee VM is that, from the previous discussion we know that, for all these VMs the response-time will be similar, and the rental-rate of the minimal VM among them is least. Hence, the minimal VM is the one with minimum monetary expenditure and similar response-time, thus, it Pareto-dominates all the sandwiched VMs. Again, knee VM is the one among the dominating VMs, and cannot be dominated by other VMs.

Finding Minimal VM

The algorithm `FindMinimal` identifies the set of minimal VM(s), its complete routine is given in Algorithm 1. We know that preprocessing ensures that the VMs that appear later in the set VM will have $\mathtt{r}_0$ atleast that of their predecessor, so the function `FindMinimal` compares the VMs on their second resource – $\mathtt{r}_1$. If a later VM has smaller $\mathtt{r}_1$ then it means that these two are mutually incomparable and minimal. Such VMs are added to the sets MinimalVM. It also uses a set XcludeVM which contains all the processed as well as pruned VMs. To ensure that a VM is not processed more than once, it is checked if it is not in XcludeVM. Finally, MinimalVM is given as the output of this sub-routine.

Finding Maximal VM for a Given VM

The function `FindMaximal` given in Algorithm 2 finds the set of maximal VMs. The input arguments are VM , XcludeVM and the $\mathtt{VM}_m$ that is the chosen minimal VM. After preprocessing, the VMs are sorted in the increasing order of resources, and we need to find the VM with highest resources or the maximal VM. So, we process VM in reverse order of resources which is denoted by VM^{rev}. At first, we check if the concerned VM is comparable to $\mathtt{VM}_m$ and maximal in the available set of VMs. Finally, we add it in the sets MaximalVM and XcludeVM after ensuring that it is not already processed.

Algorithm 1: FindMinimal

```
Initialization: MinimalVM = NULL
$\mathrm{VM}_i$ = first VM $\in$ VM such that $\mathrm{VM}_i \notin$ XcludeVM
MinimalVM $\cup$ $\mathrm{VM}_i$
currVM= $\mathrm{VM}_i$
j = 0
while $j <$ | VM | do
    if $r_{j1} \leq r_{i1}$ and $VM_j \notin$ XcludeVM then
        MinimalVM $\cup$ $\mathrm{VM}_j$
        currVM = $\mathrm{VM}_j$
    end
    j = j + 1
end
return MinimalVM
```

Algorithm 2: FindMaximal

```
Initialization: MaximalVM = NULL
$\mathrm{VM}_i$ = first VM $\in$ VM $^{rev}$ such that $\mathrm{VM}_i \notin$ XcludeVM
MaximalVM $\cup$ $\mathrm{VM}_i$
currVM= $\mathrm{VM}_i$
j = | VM |
while $j \geq 0$ do
    if $VM_j.r_1 \geq currVM.r_1$ and $VM_j \notin$ XcludeVM then
        MaximalVM $\cup$ $\mathrm{VM}_j$
        currVM = $\mathrm{VM}_j$
    end
    j = j - 1
end
return MaximalVM
```

Populating Potential Pareto-Optimal Set (PPOS)

The pseudocode of the complete routine to populate PPOS is given in Algorithm 3, the inputs to this algorithm are the set of preprocessed VMs – VM and the query. It also accepts an optional input λ_T , which is a user-defined threshold on time. The algorithm uses the aforementioned routines – Algorithm 1 and Algorithm 2, to identify the minimal and maximal VMs respectively.

Once the minimal and maximal VMs are identified, query response-time on them are obtained using the sub-routine `GetQTime (VM`$_i$`, Q)`. The function `GetQTime (VM`$_i$`, Q)` makes a call to the optimizer after setting parameter values that reflect $\texttt{VM}_i$ to the optimizer. These processed VMs are then added to the PPOS and XcludeVM. Note that the set XcludeVM contains the pruned and processed VMs, also, it is common in all the three algorithms. Its purpose is to ensure that each VM is processed only once and pruned VMs are never processed. Before processing any VM it is checked if it is not already in this set. Next, if response-times on the minimal and maximal VMs are significantly different, then search for the knee VM is continued in the set of unprocessed VMs. As soon as a pair of minimal and maximal VMs that give similar query response-times is identified, all the VMs that are bounded by these two VMs are added to the set XcludeVM. The algorithm continues till the size of XcludeVM is equal to that of VM, implying that the number of VMs is exhausted. The set PPOS is the final output of this algorithm.

On the availability of λ_T, the notion of similar response times changes to $\texttt{T}_i \leq (1+\lambda_T)\ \texttt{T}_j$, with $\texttt{VM}_i \preceq \texttt{VM}_j$ and $\texttt{T}_i$, $\texttt{T}_j$ be the respective response-times.

Note that there could be more than one maximal VM, but in Algorithm 3 we use only first element of the set `MaximalVM`. In case, a VM is the last or the only minimal VM with more than one maximal VM, we pair it again with the remaining maximal VMs.

5.3.3 Characterizing the Knee VM

Once the set PPOS is populated, no further calls to the query optimizer are required. The total money required by each of the VM in PPOS is calculated using respective rental rates and response-times.

Normalizing the money and time axes: While calculating the Euclidean distance of the VMs from origin to identify the knee VM, the magnitude of the money and time values could create a bias. For example, if the money is in the range of 1-10 cents and time in the range of 50-300 minutes, then the VM with lower response-time but higher monetary expenditure is likely to be selected as the knee. Therefore, to give equal importance to both the objectives we

Algorithm 3: Populating PPOS for the given VM and query Q

```
Initialization: XcludeVM = NULL, PPOS = NULL
while |XcludeVM| < |VM| do
    MinimalVM = FindMinimal (VM , XcludeVM)
    foreach $VM_i \in MinimalVM$ do
        XcludeVM ∪ $VM_i$
        MaximalVM = FindMaximal (VM , XcludeVM, $VM_i$)
        XcludeVM ∪ MaximalVM[0]
        GetQTime ($VM_i$ , Q)
        GetQTime (MaximalVM[0], Q)
        PPOS ∪ $VM_w$ ∪ MaximalVM[0]
        if $T_i \approx T_{MaximalVM[0]}$ then
            Add all VMs bounded by $VM_i$ and MaximalVM[0] to XcludeVM
        end
        if $VM_{i+1} = NULL\ and\ MaximalVM_1 \neq NULL$ then
            i = i - 1
            continue
        end
    end
end
```

normalize the axes using the *feature scaling* technique – $\frac{X_i - X_{min}}{X_{max} - X_{min}}$. It brings all the values in the range [0,1] for both the axes.

Finally, the knee VM of the query is identified by calculating Euclidean distance of each VM in PPOS from the origin in the XTS. The VM with minimum Euclidean distance is selected as the preferred choice following Definition 5 in Chapter 3.

Obtaining all the Pareto-optimal VMs: If the user asks for all the Pareto-optimal VMs, then PPOS is filtered to get the final set of VMs which are Pareto-optimal. We used standard algorithm of Kung et al.[19] for this filtering. This approach first sorts the solutions of PPOS in the ascending order of time. Thereafter, they are recursively halved into two subsets as Top (T, say) and Bottom (B, say). Knowing that the solutions in T are better (lesser) in time, bottom-half is checked with top-half for domination. The solutions of B that are not dominated by any members of T are combined with members of T to give a final set – POS . The check for domination and merging starts with the innermost subset and proceeds in bottom-up manner.

Solution for constrained time and/or money budget: In case, user has provided some fixed budget on time and/or money, the VMs falling out of the budget are simply ignored. The knee VM is then identified from the remaining set of the VMs.

5.4 Guarantees on the Knee VM

In this section, we will show that the aforementioned algorithm never misses an optimal knee VM when the value of λ_T is zero i.e. no time threshold is given. Also when a relaxation is provided with $\lambda_T > 0$, then PIK finds a sub-optimal VM as mentioned in Chapter 3, which has higher response-time than the optimal knee VM but is within λ_T , and its money requirement is lesser. This way, when time threshold is provided we get the knee VM, worse in one objective but within the provided threshold and better on other objective. The proof for the claim is as follows.

Claim: *The knee VM obtained by PIK satisfies:*

$$T_i \leq (\ 1 + \lambda_T\ )T_O \text{ and } M_i \leq M_O$$

where T_O, M_O denote time and money respectively, of the knee VM on the actual Pareto-front and T_i, M_i are time and money respectively, of the knee VM on the Pareto-front obtained by PIK .

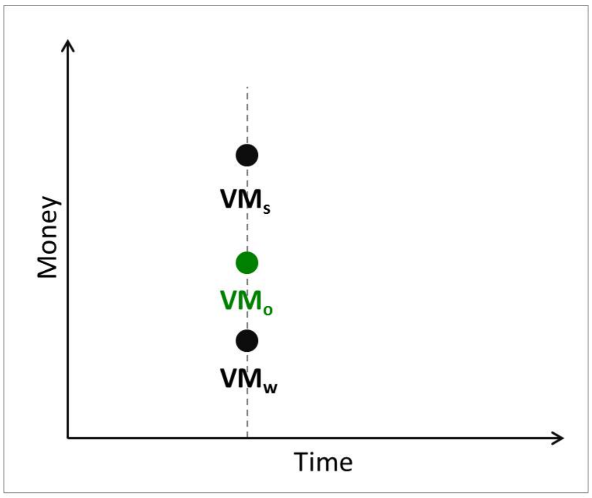
Proof: Let there be a $\mathtt{VM}_O$ on the Pareto-front of an *oracle* algorithm and missed by PIK. Let $\mathtt{VM}_O$ be in between some $\mathtt{VM}_w$ and $\mathtt{VM}_s$ which are processed by PIK . Now, to investigate further we divide the proof in two cases – one with $\lambda_T = 0$ and other with non-zero value of λ_T.

Case a: $\lambda_T = 0$: $\mathtt{VM}_O$ can be missed by PIK if the VMs in between $\mathtt{VM}_w$ and $\mathtt{VM}_s$ are not processed. This could happen, either if there is no VM bounded by them (i.e. there is no such $\mathtt{VM}_O$ possible) or they have similar response-times as shown in Figure 5.4a.

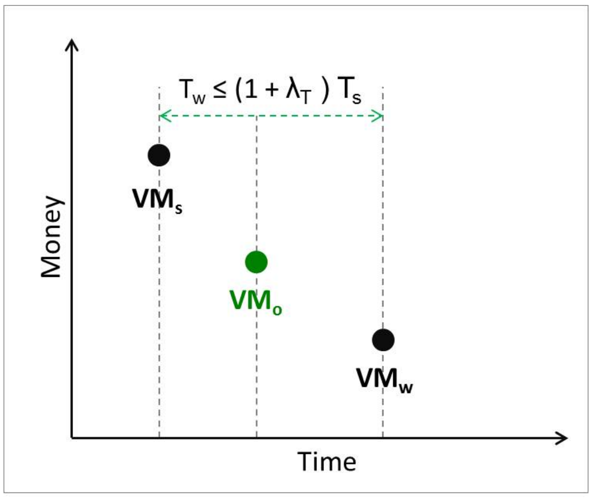
In case $\mathtt{T}_w \approx \mathtt{T}_s$, from the discussion of Section 5.2 we know that $\mathtt{T}_O \approx \mathtt{T}_s$. Since $\mathtt{VM}_w \preceq \mathtt{VM}_s$, $\mathtt{RR}_w < \mathtt{RR}_O$ therefore, $\mathtt{M}_w < \mathtt{M}_O$. Thus, contradicting the claim that $\mathtt{VM}_O$ is given by *oracle* as it is clearly dominated by $\mathtt{VM}_w$.

Case b: $\lambda_T > \mathbf{0}$: Again, VMs between $\texttt{VM}_s$ and $\texttt{VM}_w$ remain unprocessed by PIK, if either there is no VM bounded by them (i.e. there is no such $\texttt{VM}_O$ possible) or $\texttt{T}_w \leq (\ 1 + \lambda_T\ )\ \texttt{T}_s$. Since, $\texttt{VM}_O$ is bounded by these two VMs, therefore $\texttt{T}_s \leq \texttt{T}_O \leq \texttt{T}_w$ as shown in Figure 5.4b and $\texttt{M}_w \leq \texttt{M}_O$, hence VM_w is the VM satisfying the claim.

Since, knee VM $\in$ PPOS, it will also satisfy the above claim. Thus, the proposed algorithm will find the knee, with either the same performance as the optimal or with response time $\leq (1 + \lambda_T\ )$ of the optimal and lesser total money, if such a VM exists.



(a) Case a: No time threshold is given



(b) Case b: Non-zero time threshold is given

Figure 5.4: Quality of the VM obtained by the algorithm

5.5 Summary

In this chapter, we described our plan-based algorithm to identify the knee VM among the given set of VMs, for a given query. To start with, the available VMs were arranged in the poset order of their resources. Subsequently, in every iteration the complete plan was obtained for the minimal and maximal VMs of the poset. Now, if their estimated query response-times were similar then all the sandwiched VMs were pruned, since none of them could qualify for knee, as they were dominated by the minimal VM. Otherwise the next pair of minimal and maximal VMs was identified among the unprocessed VMs. This continues till all the VMs were either processed or pruned. The knee VM was then identified among the processed VMs.

Lastly, we proved that PIK identifies the optimal knee VM when no relaxation in response-time is given, otherwise it finds a near-optimal knee VM within the user-defined threshold as explained in Chapter 3. The algorithm is empirically evaluated in the following chapter.

Chapter 6

Empirical Evaluation of PIK

This chapter gives a detailed account of the experimental performance of – PIK. The performance results of PIK are given, after detailing the experimental setup. Experiments are done for the queries on the standard decision-support benchmark – TPC-DS. It was found that most of the time PIK identifies the knee VM by processing only 20% of the total VMs.

Next, we delve into the performance details of PIK for a few queries to understand in-depth working of the algorithm. Later, the effect of user-defined threshold – λ_T on the efficiency of the algorithm is studied. Empirical results show that often, giving a value of 10-20% to λ_T gives material improvement in the efficiency of the algorithm.

6.1 Experimental Framework

To evaluate PIK, we implemented it as a Java program that uses JDBC calls to get the execution-plans of queries for different VMs on **ComOpt**. The testbed for these experiments is a GCE VM comprising of 24 core Intel Ivy Bridge processor with 155 GB main memory, all the other required VMs are virtualized over it. Since, the experiments are done using the query execution-plans available from explain utility of **ComOpt**, the required VMs can be easily reflected by updating the different parameter values of **ComOpt** query optimizer.

Experiments are performed on the standard benchmark TPC-DS of size 100 GB. the database is partitioned into four different hard disks which are in the same physical region as the VM. We experimented on two physical schema of the database – Default Index(DI) and All Index (AI) that are explained in Chapter 4.

The number of queries in our experiments is limited, because not many queries show notable difference in optimizer's plan cost with change in VM configurations. This is due to coarse cost modeling in **ComOpt** with respect to hardware related parameters. We have seen its evidence in the experiments of Chapter 4 also. There are many VMs giving small difference in response-time for compile-time plots but are significantly apart on execution-time plots. Hence, the following evaluation of PIK can be expected to be coarser than its evaluation on actual execution-time. Therefore, we also present the performance of PIK on actual response-times for a few queries only, due to pragmatic constraints.

6.2 Performance of PIK

The performance of PIK for some benchmark queries of TPC-DS is presented in Table 6.1a for DI schema and in Table 6.1b for AI schema, with $\lambda_T = 0$. The leftmost column specifies the query, next column gives the cardinality of PPOS, which is essentially the number of VMs for which PIK queried the database query optimizer, and the last column gives the selected knee VM. It is clear from Table 6.1 that PIK never performs as bad as the exhaustive enumeration of the VMs.

Variations in PPOS and knee VM with the schema: Observe that the cardinality of PPOS and the knee VM changes with schema. E.g. for query DI-DSQ-4, minimal VM of the poset (VM , $\preceq$) is selected as the knee, since there is no significant variation in the response-time of minimal and maximal VMs. However, same query on AI-schema, processed 12 VMs to find the knee. This is due to the changes in query plans with physical schema.

Specifically, as seen in Chapter 4 also, query plans are more sensitive to available memory when the indexes are available. Since, these indexes are non-clustered and can be used only if enough main memory is available.

However, for some queries e.g. AI-DSQ-6, AI-DSQ-19, the cost of query plans on AI schema show no significant variation for the minimal and the maximal VMs. In these queries the cardinality of scan nodes is low and the memory available in minimal VM is sufficient for the chosen query plan. Hence, no variation in query response-time across VMs is visible. However, without indexes, query plans use sort nodes which are more sensitive to memory availability, leading to varying response-time across VMs.

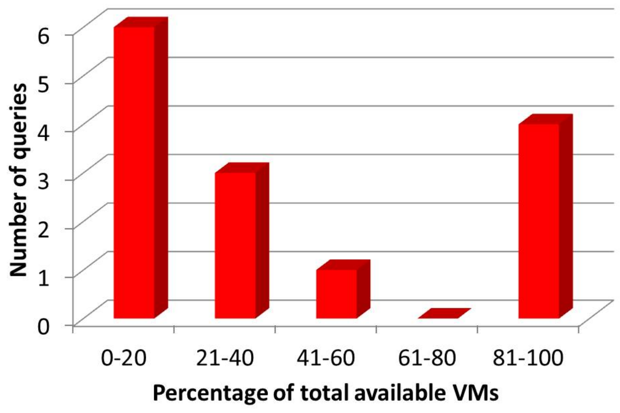
Minimal VM as the knee: Another notable point is that, for some queries, size of PPOS is high, still the minimal VM is chosen as the knee, e.g. DI-DSQ-14 and DI-DSQ-24. Queries exhibit such behavior when variation in response-times is low with respect to the rental rates of virtual machines. Hence, there is no significant reduction in the monetary investment for the VMs of higher configuration, and the minimal VM qualifies as the knee.

High cardinality of PPOS: Table 6.1 shows that for some queries e.g. DI-DSQ-19, DI-DSQ-52, DI-DSQ-55, and DI-DSQ-71, 85% of the total VMs are processed. The number of tried VMs is high for these queries because of operators like nested-loop joins and sort. As discussed in Chapter 4, presence of these operators in the query plans lead to response-time variation even with small increase in available memory, even though it is a small variation.

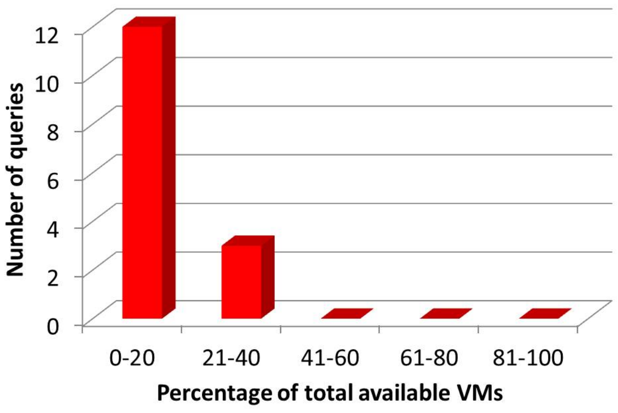
The other factor which plays a role in increasing the cardinality of PPOS, is the construction of PIK itself. All the minimal and maximal VMs are tried at every iteration. Therefore, as the number of iterations increases more incomparable VMs are found, resulting in more processed VMs. The required per-node memory is high for these queries, which requires more iterations to get the VM with enough required memory, hence, higher cardinality of PPOS.

However, when using PIK on actual response-times for DI-DSQ-19 and DI-DSQ-55, the size of PPOS is 18 and 36 respectively. This indicates that the high cardinality of PPOS for estimated response-times is because of coarse cost-modeling of **ComOpt** .

Performance summary: Figure 6.1 summarizes the performance of PIK , indicating that for most of the queries PIK finds the Pareto-front by trying only 20% of the total VMs for both AI as well as DI schema. The cardinality of PPOS for AI is within 40% of the total VMs, for all the tested queries. On the other hand, for DI there are a few queries requiring 85% of total VMs to be processed to identify the knee VM. The reasons for these differences with schema are already explained.



(a) Performance summary for DI



(b) Performance summary for AI

Figure 6.1: Performance summary of PIK

6.2.1 Performance Microanalysis

This section discusses the in-depth performance of PIK in detail. There are three main issues we want to highlight:

- **Issue 1:** increase in cardinality of PPOS with number of required iterations
- **Issue 2:** trying incomparable VMs even after identifying a pair with similar response-times
- **Issue 3:** different minimal-maximal pair for same iterations across queries

To explain the reasons for each of these cases, we picked queries exhibiting atleast one of these behaviors.

Query	|PPOS|	Knee VM
DI-DSQ-3	12	[8 core, 7 GB]
DI-DSQ-4	2	[1 core, 4 GB]
DI-DSQ-6	94	[2 core, 10 GB]
DI-DSQ-14	76	[1 core, 4 GB]
DI-DSQ-19	158	[6 core, 10 GB]
DI-DSQ-24	12	[1 core, 4 GB]
DI-DSQ-47	2	[1 core, 4 GB]
DI-DSQ-55	158	[2 core, 13 GB]
DI-DSQ-59	68	[2 core, 10 GB]
DI-DSQ-67	6	[2 core, 5 GB]
DI-DSQ-71	158	[2 core, 10 GB]
DI-DSQ-74	2	[1 core, 4 GB]

(a) DI schema and $\lambda_T = 0$

Query	|PPOS|	Knee VM
AI-DSQ-3	12	[6 core, 5 GB]
AI-DSQ-4	12	[2 core, 5 GB]
AI-DSQ-6	2	[1 core, 4 GB]
AI-DSQ-14	32	[4 core, 15 GB]
AI-DSQ-19	2	[1 core, 4 GB]
AI-DSQ-24	60	[10 core, 30 GB]
AI-DSQ-47	6	[2 core, 5 GB]
AI-DSQ-55	2	[1 core, 4 GB]
AI-DSQ-59	67	[20 core, 18 GB]
AI-DSQ-67	6	[2 core, 5 GB]
AI-DSQ-71	2	[1 core, 4 GB]
AI-DSQ-74	30	[2 core, 10 GB]

(b) AI schema and $\lambda_T = 0$

Total Number of available VMs = 186

Table 6.1: Performance of PIK on TPC-DS benchmark queries

Issue 1: Table 6.2 gives per iteration breakdown of AI-DSQ-14. As the number of iterations increase, more incomparable minimal VMs are found, eventually increasing the size of PPOS. This increase in the number of incomparable VMs is owed to the configurations of VMs available with GCE. We have already seen in Chapter 5 the Hasse diagram of GCE which explains it. However, if the VMs are in total order or with lesser incomparable VMs then the cardinality of PPOS might decrease.

Issue 2: In Table 6.3, analysis of AI-DSQ-3 is given where total number of VMs processed is 12. This is an aggregate query with join of three tables – date_dim, store_sales, and item. The query plan is same across VMs and have simple join operators without parallelism. Thus available memory is the only significant factor in plan cost reduction. Since, two tables out of three are small, per-node memory requirement of plan is low. Also, from Table 6.3 it is evident that, once RAM size reaches 10 GB, any further increase in resources does not reduces plan cost. Therefore, all the VMs bounded by [2 core, 10 GB] and [24 core, 140 GB] are pruned. But, the algorithm does not terminate at this iteration, as there are more incomparable pairs

AI-DSQ-14			
Iteration	Weakest VM	Strongest VM	Number of VMs pruned
1	[1 core, 4GB]	[24 cores, 155GB]	2
2	[1 core, 6GB]	[24 cores, 150GB]	2
	[2 cores, 5GB]	[24 cores, 145GB]	2
3	[2 cores, 10GB]	[24 cores, 140GB]	2
	[4 cores, 5GB]	[24 cores, 135GB]	2
4	[2 cores, 13GB]	[24 cores, 130GB]	2
	[4 cores, 10GB]	[24 cores, 125GB]	2
	[6 cores, 5GB]	[24 cores, 120GB]	2
5	[4 cores, 15GB]	[24 cores, 115GB]	154
	[6 cores, 10GB]	[22 cores, 143GB]	2
	[8 cores, 7GB]	[22 cores, 140GB]	2
6	[8 cores, 10GB]	[22 cores, 135GB]	2
7	[10 cores, 10GB]	[22 cores, 130GB]	2
8	[12 cores, 11GB]	[22 cores, 125GB]	2
9	[14 cores, 13GB]	[22 cores, 120GB]	2
10	[16 cores, 14GB]	[20 cores, 130GB]	2
|PPOS| = 32			
Knee VM =[2 cores, 13GB]			

Table 6.2: Microanalysis of AI-DSQ-14 with $\lambda_T = 0$

of minimal-maximal VMs.

Based on the construction of PIK , all of these pairs are tried irrespective of query response-time on the incomparable VMs. Also, at every iteration atleast two VMs that constitute the minimal-maximal pair of that iteration are pruned.

Issue 3: The minimal-maximal VM pair at same iteration number is different for AI-DSQ-3 and AI-DSQ-14. At iteration 4 of AI-DSQ-3 the minimal-maximal pair is [6 cores, 5 GB] and [8 cores, 7 GB] but for AI-DSQ-14 it is a set of three incomparable VMs. The reason is that in AI-DSQ-3 at iteration 3 we found a pair with same cost which pruned all the VMs in between them, so the next minimal-maximal pair is selected from the remaining subset of VMs.

All the VMs are covered efficiently: Further, from the third column in each of the aforementioned tables it is evident that the algorithm covers all the VMs by either trying or pruning them, since it always sums up to the total number of available VMs – 186.

AI-DSQ-3			
Iteration	Weakest VM	Strongest VM	Number of VMs pruned
1	[1 core, 4 GB]	[24 core, 155 GB]	2
2	[1 core, 6 GB]	[24 core, 150 GB]	2
	[2 core, 5 GB]	[24 core, 145 GB]	2
3	[2 core, 10 GB]	[24 core, 140 GB]	176
	[4 core, 5 GB]	[22 core, 143 GB]	2
4	[6 core, 5 GB]	[8 core, 7 GB]	2
|PPOS| = 12			
Knee VM = [1 core, 6 GB]			

Table 6.3: Microanalysis of AI-DSQ-3 with $\lambda_T = 0$

6.2.2 Effect of Time Threshold

The effect of time threshold (λ_T) on the size of PPOS for different queries is shown in Figure 6.2 for DI schema and in Figure 6.3 for AI schema. The value of λ_T is varied from 10% to 50%. It is evident that for most of the queries, cardinality of PPOS reduces with increase in λ_T.

No effect of λ_T on the cardinality of PPOS: However, for some queries, e.g., AI-DSQ-67 the number of processed VMs remains constant. The reason can be explained using Table 6.4, which gives microanalysis of the query when no time threshold is provided. Table 6.4 illustrates that further reduction in the size of PPOS can be seen only when the difference between the response-times for the minimal-maximal pair of VMs at first iteration is within the provided threshold.

AI-DSQ-67			
Iteration	Weakest VM	Strongest VM	Number of VMs pruned
1	[1 core, 4GB]	[24 cores, 155GB]	2
2	[1 core, 6GB]	[24 cores, 150GB]	181
	[2 cores, 5GB]	[6 cores, 5GB]	3
|PPOS| = 6			
Knee VM = [1 core, 6GB]			

Table 6.4: Microanalysis of AI-DSQ-67 with $\lambda_T = 0$

Effect of λ_T on the knee VM: For query AI-DSQ-14, effect of λ_T on the size of PPOS can be

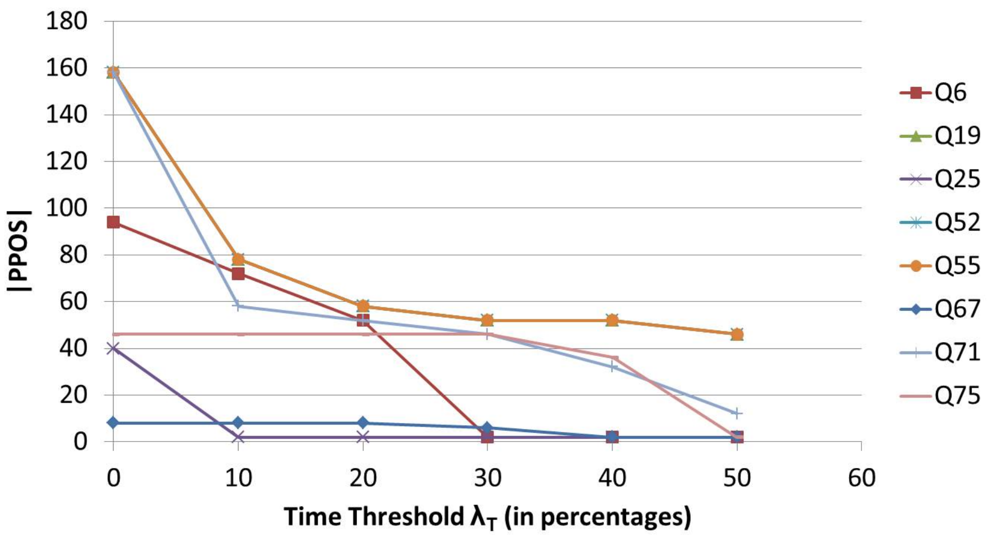


Figure 6.2: Effect of λ_T on DI

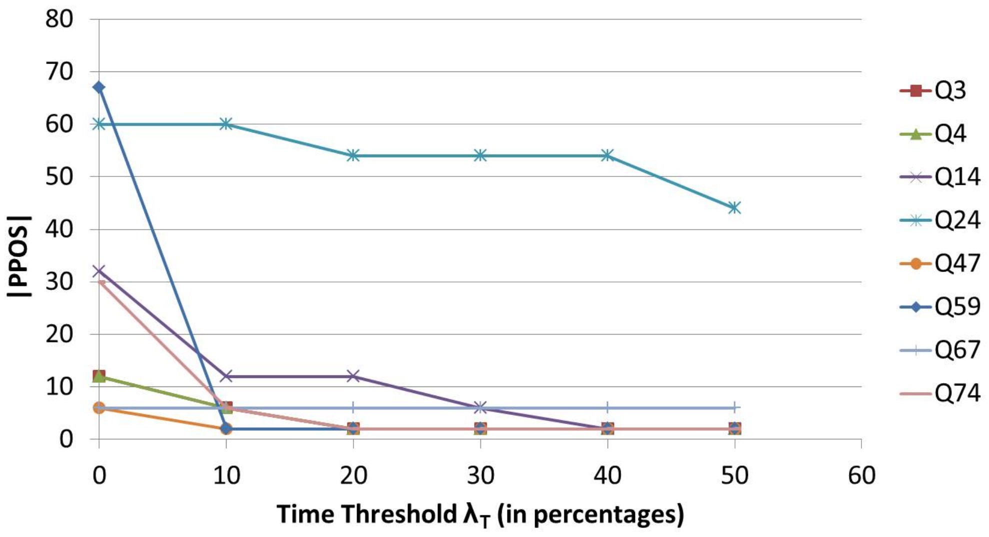


Figure 6.3: Effect of λ_T on AI

seen in Table 6.6 and Table 6.7. The VM selected as the knee also changes with the threshold value. Further, Table 6.5 shows that the knee VM remains same as when no time threshold is provided (given in Table 6.3). The reason being low resource requirements of the query, hence, no reduction in query response-time with additional resources.

Additionally, from Figure 6.3 it is clear that when λ_T=20% the total number of tried VMs for AI-DSQ-3 is reduced to 2, which means that the difference in the response-times of the minimal and maximal VM of first iteration are within 20%. Hence, even though more VMs are tried at lower values of λ_T, now the difference is low enough to select same VM as the knee.

AI-DSQ-3, λ_T =10%			
Iteration	Weakest VM	Strongest VM	Number of VMs pruned
1	[1 core, 4GB]	[24 cores, 155GB]	2
2	[1 core, 6GB]	[24 cores, 150GB]	181
	[2 cores, 5GB]	[6 cores, 65GB]	3
|PPOS| = 6			
Knee VM = [1 core, 6GB]			

Table 6.5: Microanalysis of AI-DSQ-3 with λ_T = 10%

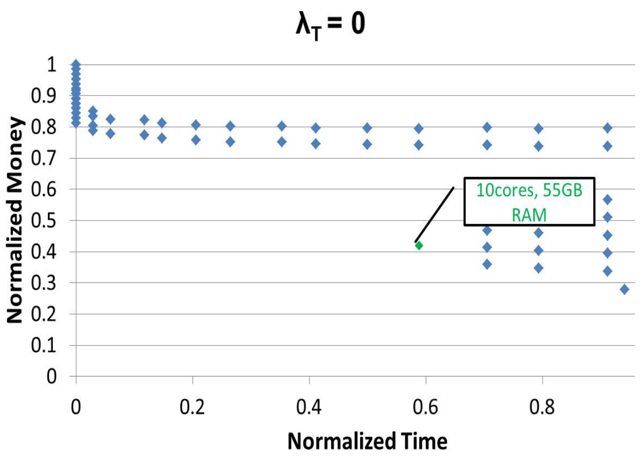
AI-DSQ-14, λ_T =10%, 20%			
Iteration	Weakest VM	Strongest VM	Number of VMs pruned
1	[1 core, 4GB]	[24 cores, 155GB]	2
2	[1 core, 6GB]	[24 cores, 150GB]	2
	[2 cores, 5GB]	[24 cores, 145GB]	2
3	[2 cores, 10GB]	[24 cores, 140GB]	176
	[4 cores, 5GB]	[22 cores, 143GB]	2
4	[6 cores, 5GB]	[8 cores, 7GB]	2
|PPOS| = 12			
Knee VM = [2 cores, 10GB]			

Table 6.6: Microanalysis of AI-DSQ-14 with λ_T = 10%, 20%

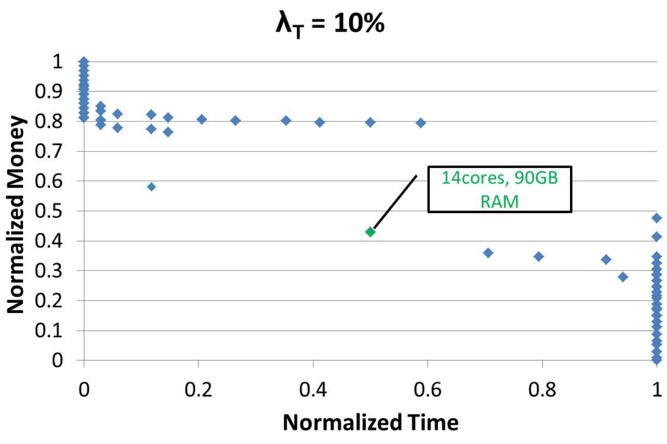
AI-DSQ-14, λ_T =30%			
Iteration	Weakest VM	Strongest VM	Number of VMs pruned
1	[1 core, 4GB]	[24 cores, 155GB]	2
2	[1 core, 6GB]	[24 cores, 150GB]	181
	[2 cores, 5GB]	[24 cores, 145GB]	3
|PPOS| = 6			
Knee VM = [1 core, 6GB]			

Table 6.7: Microanalysis of AI-DSQ-14 with λ_T = 30%

Location of omitted VMs with λ_T : Furthermore, Figure 6.4 shows the effect of λ_T on the size of PPOS along with corresponding locations of VMs in XTS , for DI-DSQ-6. Similarly, Figure 6.5 gives the plots of DI-DSQ-52 for different values of λ_T. Evidently, the VMs which are omitted when higher threshold is given, are the dominated VMs for lower values of λ_T. Observe that when a non-zero value of threshold is provided the knee VM selected is always to the right of the ideal knee VM (when $\lambda_T = 0$), as mentioned in last chapter about the near-optimal solution. Note that the threshold is always applied on non-normalized values.



(a) DI-DSQ-6 with $\lambda_T = 0$

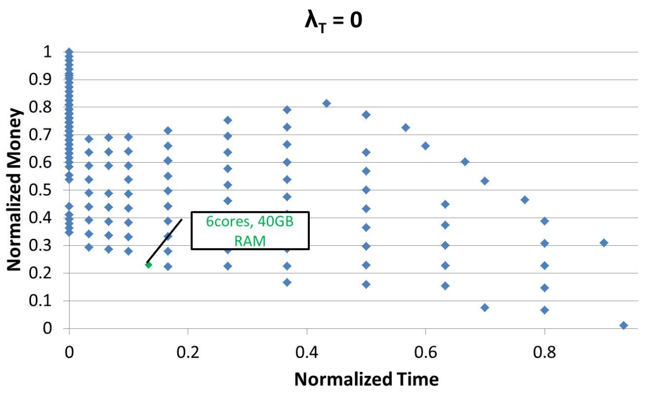


(b) DI-DSQ-6 with $\lambda_T = 10\%$

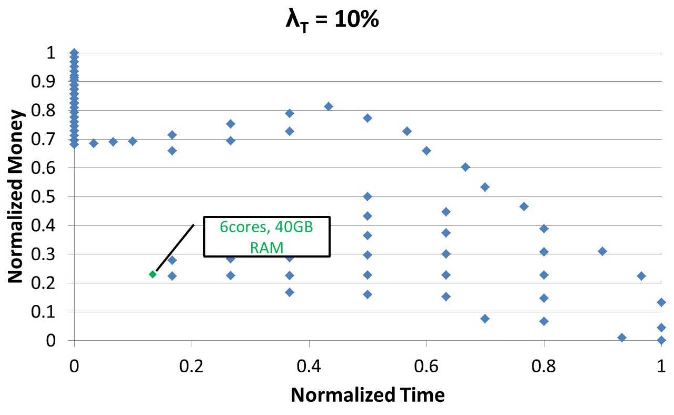
Figure 6.4: Effect of λ_T for DI-DSQ-6

6.3 Summary

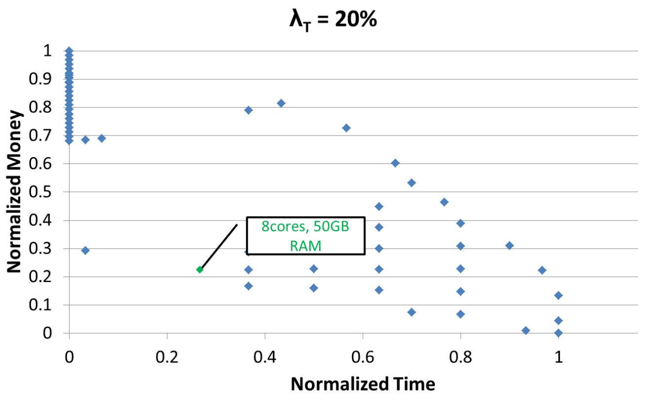
To summarize, the size of PPOS remained within 20% of the total VMs, for the most of the queries. However, for some queries, upto 85% of the total VMs were processed to find the knee VM. The query plans for these queries have operators that were highly sensitive to available memory e.g. sort operators and nested-loop joins. Further, the efficiency of the algorithm increased significantly when user-defined threshold of 20-30% is applied on time. However, the effect of time-threshold was found negligible for few queries; the notable point was that these queries already had small PPOS.



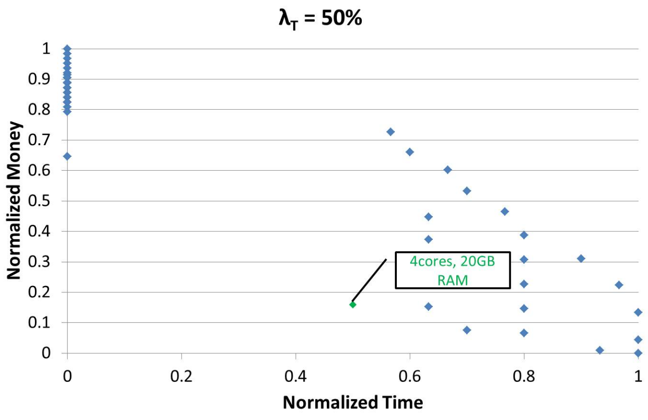
(a) DI-DSQ-52 with $\lambda_T = 0$



(b) DI-DSQ-52 with $\lambda_T = 10\%$



(c) DI-DSQ-52 with $\lambda_T = 20\%$



(d) DI-DSQ-52 with $\lambda_T = 50\%$

Figure 6.5: Effect of λ_T for DI-DSQ-52

Chapter 7

Identification of the Knee VM: A Sub-plan based Approach

In the previous chapter, we have seen that for most of the queries, PIK identifies the knee VM by processing only 20% of the total VMs. We know that, it receives the complete query plan given by the database engine's API for every required VM. However, while observing these query plans, we found that often there are repetitive sub-plans across VMs. In that case, same computation is performed by the query optimizer for many sub-plans for different VMs. However, if the query optimizer is modified, such that it prunes the VMs at sub-plan levels also to find the knee VM, then this extra computations could be reduced. This motivated us to augment the optimizer algorithm for the optimization of money expenditure alongwith response-time.

We use the concept of partial ordering of the VMs on their resources as discussed in Chapter 5, to devise a Sub-Plan-based Identification of Knee (SPIK). This algorithm prunes the VMs at the sub-plan levels and retains only the Pareto-optimal sub-plans and their corresponding VMs. An important point to note here is that a VM whose sub-plan is not Pareto-optimal at a lower node may turn out to be so at higher nodes. To ensure that we never miss such a VM and its plan, for all the non-Pareto-optimal VMs, we check if their minimum possible monetary expenditure is lesser than their stronger alternative. If yes, then we explicitly obtain the sub-plan on that VM, otherwise the VM is discarded from consideration.

We have prototyped SPIK inside Postgresql 9.3, and also implemented it as a Java wrapper program with the commercial engine. Our experimental results indicate that the total computation carried out by SPIK is within 40% of PIK. Further, the efficiency of the algorithm increases significantly when a relaxation factor of 20 to 30 % is permitted on the time axis.

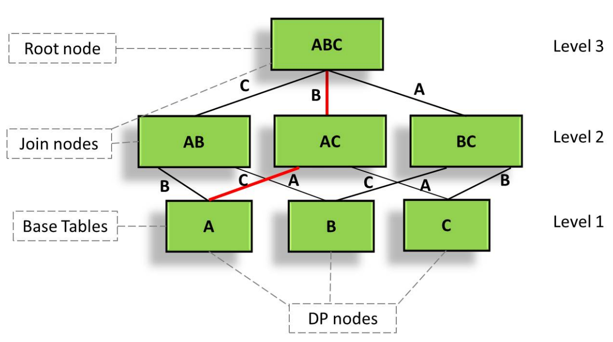
Since, this algorithm aims at pruning the VMs at sub-plan levels, it requires modifications in the query optimizer's selection process of the query-plan. Therefore, before discussing our algorithm, we present a brief background of the traditional query optimizer approach. Also, we discuss the challenges in the adaptation of this approach for the twin objectives of money and time.

7.1 Traditional Query Optimizer

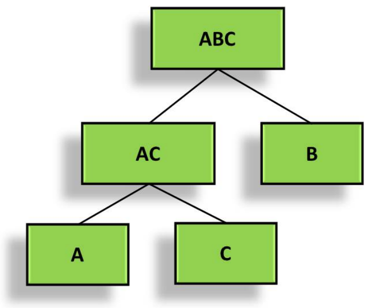
In the traditional optimizer algorithm, a Dynamic Programming (DP) based approach is used to find the plan with minimum response-time [25]. To explain the general terminology of this approach, an abstract three level DP-tree is given in Figure 7.1a. The base tables are at the leaf nodes of the DP tree, while the join nodes feature at the higher levels. Now, at each node in the tree the sub-plan with the least response-time is selected and forwarded to the higher nodes to incrementally construct the complete plan tree. This way at each node only the sub-plan with the least response-time is retained. For instance the plan shown in Figure 7.1b, i.e, the selected join order is ((A$\bowtie$C)$\bowtie$B).

Challenges for Cloud Query Optimizer

Unlike the conventional query optimizer, the goal for cloud databases is to consider both the response-time and total money expense. More importantly, instead of finding the query-plan for the fixed infrastructure, here hundreds of VMs are given as the input. Therefore, the



(a) DP tree for a sample query



(b) Selected query-plan with minimum response-time

Figure 7.1: DP based approach

information saved at each node is a quadruple – (VM_i, P_i, T_i, X_i), where P_i is the plan with time T_i and X_i monetary expenditure on VM_i. Since, in this quadruple a VM is associated with a single sub-plan, we will use the terms VM and sub-plan interchangeably hereafter. Now, if a VM requires more time but lesser money, then it has to be saved at that node. Thus, increasing the number of sub-plans saved at each node. Consequently, the total number of VMs to be tried at the nodes of higher levels may lead to unreasonable optimization time. On the other hand, naive pruning may miss out the knee VM.

In short, the main challenges in the modification of this approach are – (i) keeping the plan information for too many VMs would cause an exponential blow-up at higher nodes of the DP tree, and (ii) deciding which VMs to keep is not straightforward, as a VM may be in the POS (Pareto-Optimal Set) at the root node but absent in the POS of some intermediate nodes.

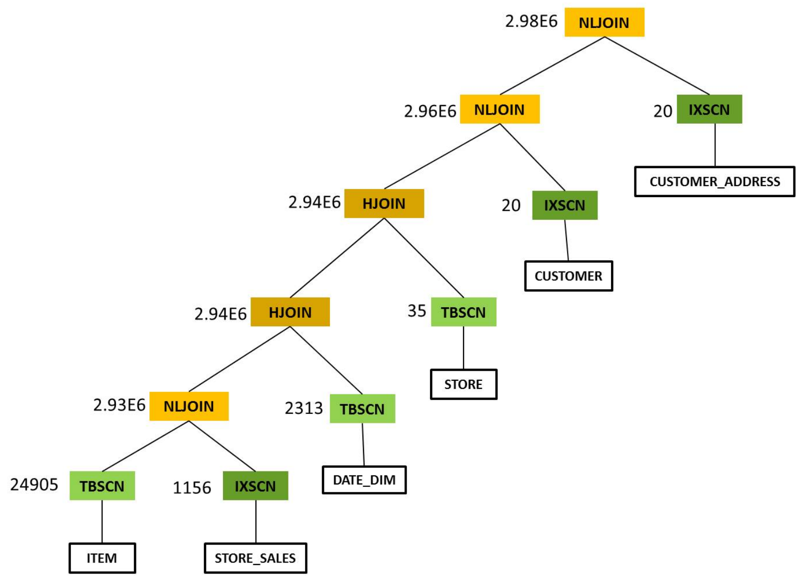
7.2 Repetitive Sub-plans Across VMs

On observing the query plans, we found that the change in operator algorithm and/or cost, etc. in the query plans with respect to the changes in hardware related parameters is fairly slow. In Figure 7.2, the query plans which are repeated across several VMs are shown. Note that the plans are same in structure as well as in optimizer's cost. These query plans remain unchanged when the provided resources are superfluous to their requirements. For example, in Figure 7.2a, the cost of the query plan is not influenced by the parallelism provided by the different VMs. Hence, the plan remains the same for given RAM size, irrespective of the number of cores available in the VMs. Similarly, in Figure 7.2b the maximum memory required by the nodes of the plan is attained when the RAM size is 60 GB, henceforth the provision of additional memory does not alter the plan.

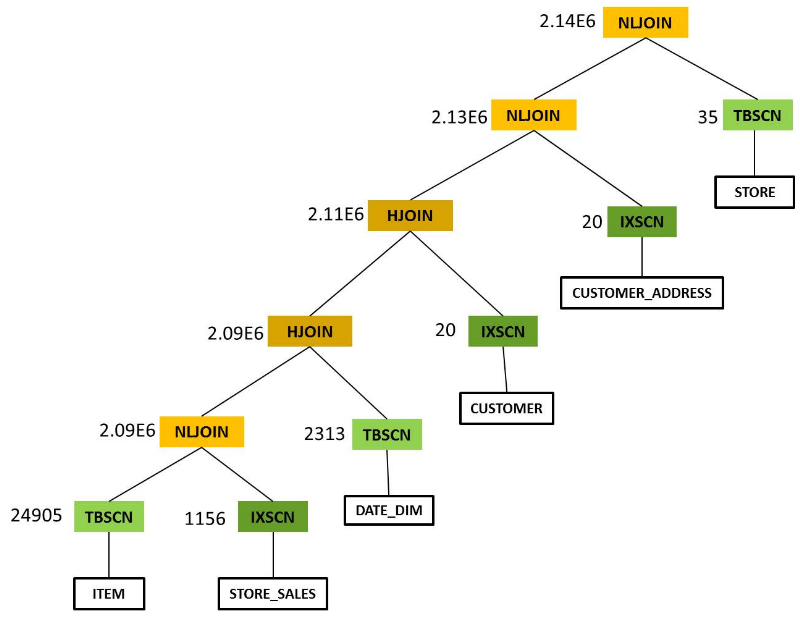
Similarly, for the query plans shown in Figure 7.3, the repeated sub-plans are shown within the dashed box. The highlighted sub-plans remain unchanged with the change in VMs. Again, it is because the resources required by the highlighted sub-plan are already attained.

Thus, if the query plans for all these VMs are computed, then at the nodes with same sub-plans query optimizer does redundant computation. This extraneous computation can be omitted, if we can control the query optimization process to prune the VMs at the sub-plans.

Overall, the objective is to include an array of VM configurations with their respective rental rates, as the additional inputs to the query optimization phase, and select the knee VM with its corresponding plan.

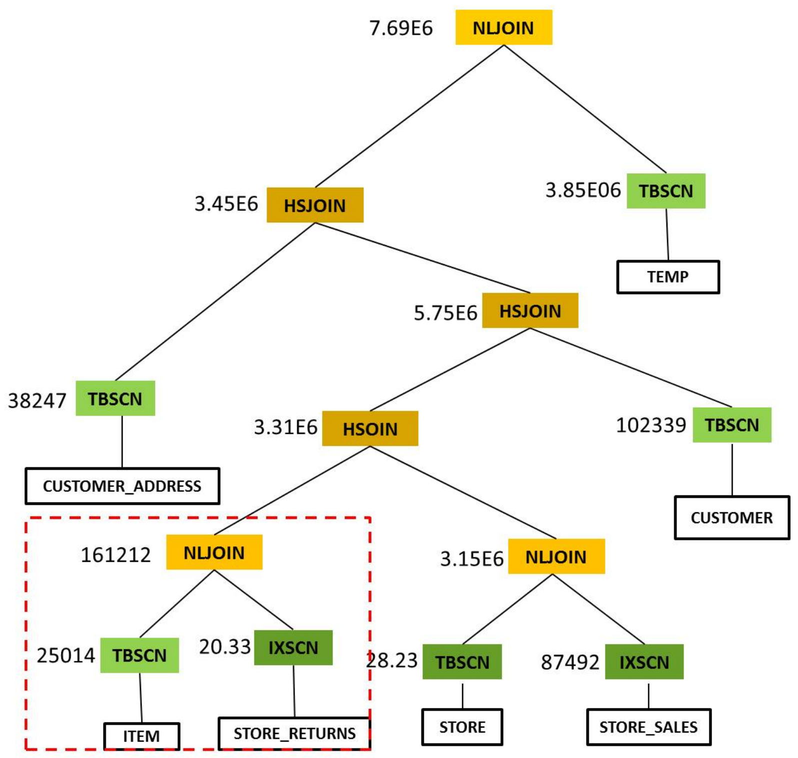


(a) For all VMs with 20 GB RAM, irrespective of number of cores

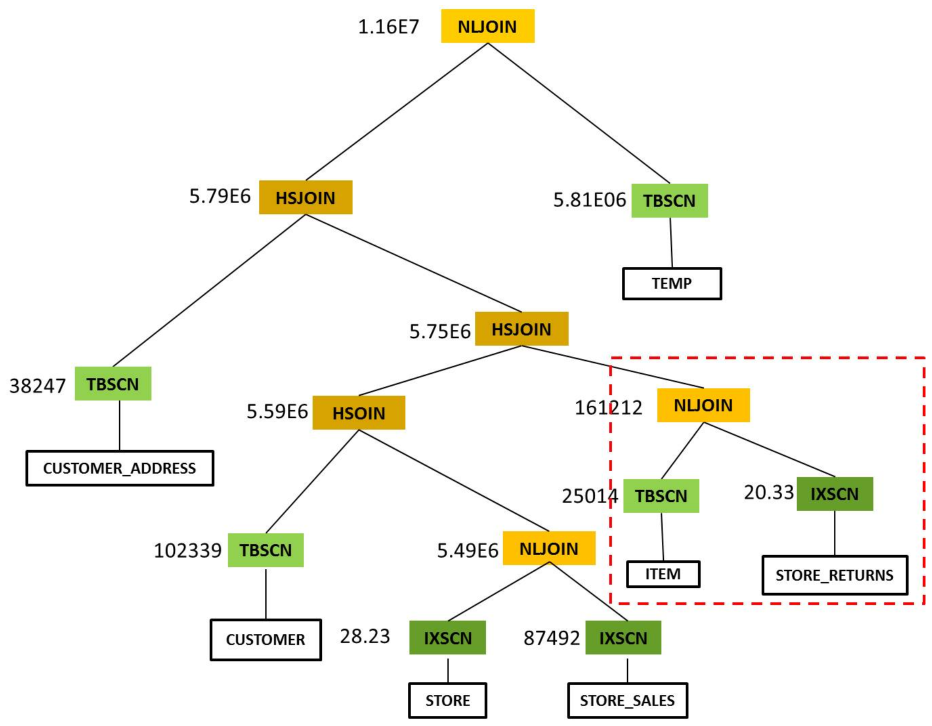


(b) For all VMs from 60 GB to 104GB RAM and 16 cores

Figure 7.2: Query plans for DI-DSQ19 across VMs



(a) For VM with 21 GB RAM and parallelism



(b) For VM with 10 GB RAM and no parallelism

Figure 7.3: Query plans for AI-DSQ-24 across VMs

7.3 Sub-Plan-based Identification of Knee (SPIK)

We present our algorithm – SPIK that *spans* over the entire space of VMs and *picks* the knee VM and its corresponding query plan. In this algorithm, at each node, the VMs are processed in the poset order of their resources as explained previously in Chapter 5. Specifically, at each DP node, it obtains the query response-time for the minimal and maximal VMs only. If these response-times differ significantly, then it identifies the next pair of minimal and maximal VMs from among the unprocessed VMs. Otherwise, it prunes all the VMs bounded by these extreme VMs. This continues this till a pair of minimal-maximal VMs with similar response-times at that node are identified or the number of VMs is exhausted.

Next, total money expenditure is calculated for all the processed VMs at a node. Now, at every node only Pareto-optimal sub-plans in XTS are saved alongwith their VMs. Further, they are forwarded to the higher nodes for the incremental plan tree construction.

7.3.1 Challenges in the Sub-Plan-based Approach

Currently only Pareto-optimal VMs of a node are selected and forwarded to higher nodes, it may lead to following anomalies:

- **Missing VMs:** Plan information required for some VMs might be missing. Specifically, when the plan is to be constructed at a node for a VM which is not in the POS of one or both of the children node(s). For example in Figure 7.4, the plan of node B for $\texttt{VM}_m$ is required at node AB.

- **Returning VMs:** Constructing the sub-plans only for the VMs in the POS of the children nodes, might lead to the omission of VMs which are Pareto-optimal in XTS at higher but not in the lower nodes of the plan-tree. For example in Figure 7.5, the $\texttt{VM}_m$ is dominated at node A but Pareto-optimal at ABC.

A detailed discussion on the occurrences and solution for each of these problems, and their proposed solutions is followed next.

Providing Missing Plan Information for VM(s)

Since, at every node we forward only the plans which are Pareto-optimal in the XTS at that node, this may lead to the unavailability of some descendant sub-plans. Particularly, this is problematic when we want to construct a plan for the VM that is not in the POS of the children node(s). Without these sub-plans further plan construction for the VM(s) could not be completed. For example in Figure 7.4, $\mathtt{VM}_m$ is not Pareto-optimal at node `B`, hence, it is omitted. However, to know the Pareto-optimal VMs at node `AB`, the plan for $\mathtt{VM}_m$ at `AB` is required, which in turn requires the sub-plan of node `B`.

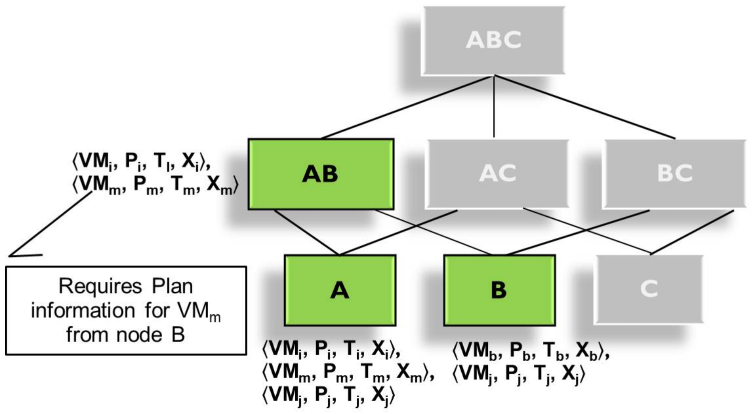

ABC
AB
AC
BC
A
B
C
⟨VM_i, P_i, T_i, X_i⟩, ⟨VM_m, P_m, T_m, X_m⟩
Requires Plan information for VM_m from node B
⟨VM_i, P_i, T_i, X_i⟩, ⟨VM_m, P_m, T_m, X_m⟩, ⟨VM_j, P_j, T_j, X_j⟩
⟨VM_b, P_b, T_b, X_b⟩, ⟨VM_j, P_j, T_j, X_j⟩

Figure 7.4: Plan information of VMs required at higher nodes might be missing

We know that, the VMs for which sub-plans are stored at a node are Pareto-optimal in the XTS at that node. This confirms that the missing VM(s) is dominated by some VM in the POS of that node. In simpler words, at a node the time as well as money expenditure for the missing VM(s) is greater than some VM in the POS of that node. Hence, it is dominated and subsequently removed from the POS of that node.

Now, let there be two VMs VM_m, VM_p such that $\text{VM}_p \preceq \text{VM}_m$ and VM_m be dominated by VM_p at the node of level n (say). Now, from the discussion of Section 5.2, we know that,

$\mathrm{T}_m \leq \mathrm{T}_p$. Therefore, $\max(\mathrm{T}_m) = \mathrm{T}_p$, thus, for each missing VM, we can use the sub-plan of a weaker VM as the upper-bound sub-plan.

To ensure that we always have a weaker VM for any given VM at any node, we save the plan information for the minimal VM(s) at every node. Also, for each missing VM, we opt for the VM that is weaker and have minimum Euclidean distance from the concerned VM on RS. If two such incomparable VMs are found, then pick any.

Identifying the Dominated VMs that may Dominate Later

In SPIK, at each node, only the VMs that are Pareto-optimal in the XTS are saved. This might cause the omission of the VMs that are Pareto-optimal at higher nodes but not at the lower nodes. An example for such a case is given in Figure 7.5, where $\mathtt{VM}_m$ is dominated at lower node (A) but is Pareto-optimal at the root node (ABC).

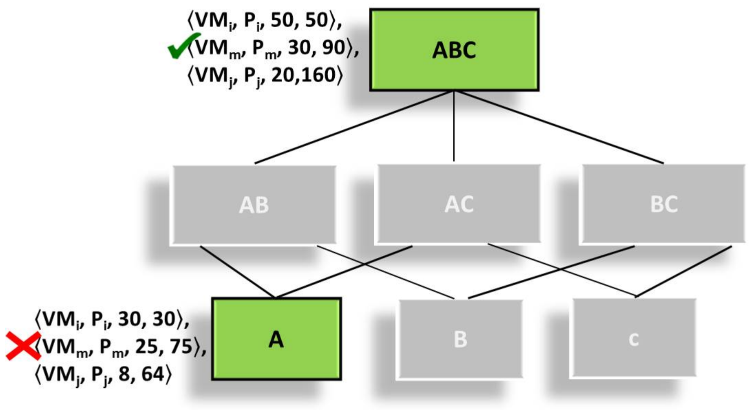


Figure 7.5: A VM dominated at lower levels may be Pareto-optimal at higher nodes

Let there be two VMs such that $\mathtt{VM}_m \preceq_p \mathtt{VM}_i$. Now at every level following will hold,

$$\mathtt{RR_m} < \mathtt{RR_i} \text{ and } \mathtt{T_i} \leq \mathtt{T_m} \tag{7.1}$$

At level x: Let $\mathtt{VM}_i$ be a Pareto-optimal and $\mathtt{VM}_m$ be a dominated VM at level x. Therefore,

$$\mathtt{X_i} \leq \mathtt{X_m} \Rightarrow \frac{\mathtt{RR_i}}{\mathtt{RR_m}} < \frac{\mathtt{T_m}}{\mathtt{T_i}}, \tag{7.2}$$

At level $\texttt{x+k}$: We will use subscript $\texttt{x+k}$ to denote the quantities of this level. Now, $\texttt{VM}_m$ can be Pareto-optimal at this level, only if $\texttt{X}_m^{x+k} \leq \texttt{X}_i^{x+k}$. Since, from equation 7.1, $\texttt{T}_i \leq \texttt{T}_m$. But, to calculate $\texttt{X}_m^{x+k}$ the value of $\texttt{T}_m^{x+k}$ is required, rather we check the following condition,

$$\min\,(\texttt{X}_\texttt{m}^\texttt{x+k}) < \texttt{X}_\texttt{i}^\texttt{x+k} \tag{7.3}$$

$$= \texttt{RR}_\texttt{m} \times \min(\texttt{T}_\texttt{m}^\texttt{x+k}) < \texttt{RR}_\texttt{i} \times \texttt{T}_\texttt{i}^\texttt{x+k}$$

Where, $\min(\texttt{T}_m^{x+k}) = \texttt{T}_m + (\texttt{T}_i^{x+k} - \texttt{T}_i)$. Note that, we are not taking $\min(\texttt{T}_m^{x+k}) = \texttt{T}_i^{x+k}$, to ensure that $\texttt{T}_m < \texttt{T}_m^{x+k}$, because, response-time is cumulative in nature. Since, we are comparing minimum value of $\texttt{X}_m^{x+k}$ with $\texttt{X}_i^{x+k}$, it may happen that the actual value of $\texttt{X}_m^{x+k}$ is greater than $\texttt{X}_i^{x+k}$ and $\texttt{VM}_m$ remains dominated. But, by checking condition 7.3 for the dominated VMs, we will never miss any VM that can be Pareto-optimal at the current level. In short, condition 7.3 may give false positives but not false-negatives.

Observe that we do not need actual value of $\texttt{T}_m^{x+k}$ to check condition 7.3, instead we can use the a lower-bound of it, which is the response-time of the weaker VM ($\texttt{VM}_p \preceq \texttt{VM}_m$) at level $\texttt{y}$ (say) such that $\texttt{y < x}$, as discussed in the previous section.

Also, to check this condition we need response-time of a stronger VM ($\texttt{VM}_i$). To ensure that plan information for such a VM is available, we save the plan of the maximal VM of the poset at every node. For each missing VM, we opt for the VM that is stronger and have minimum Euclidean distance from the concerned VM on RS. If two such incomparable VMs are found, then pick any.

7.3.2 Complete SPIK Algorithm

The complete algorithm to modify the planning process of the query optimizer is given in Algorithm 4. At the leaf level, it obtains the sub-plans for minimal-maximal VMs of the poset in that order, till either all the VMs are processed or pruned. Next, filter the Pareto-optimal VMs of the node and forward them to the higher nodes (set ForwardVM constitute such VMs). For the remaining nodes, it constructs the sub-plans for the VMs received from lower nodes and those satisfying condition 7.3. If the response-times are found to be similar for any two VMs, then all the VMs sandwiched between them are pruned. Finally, at the root node it identifies the knee VM by calculating their Euclidean distance on the XTS , after applying normalization as given in Section 5.3.3.

Algorithm 4: Sub-Plan based Identification of Knee

```
for Each node n in DP tree do
    Initialization: XcludeVM = NULL, PPOS = NULL, ForwardVM = NULL,
     ReceivedVM = Union of the $ForwardVM_i$ set of each child
    MinimalVM = FindMinimal (VM , XcludeVM)
    MaximalVM = FindMaximal (VM , XcludeVM, $MinimalVM_0$)
    GetQTime ($MinimalVM_0$, Q)
    GetQtime ($MaximalVM_0$, Q)
    $ForwardVM_n \cup MinimalVM_0$
    $ForwardVM_n \cup MaximalVM_0$
    if optimizer's cost on minimal and maximal VMs are similar then
        Add all the VMs in between the minimal and maximal VMs to the set XcludeVM
    else
        for each $VM_i \in ReceivedVM$ and $VM_i \notin XcludeVM$ do
            GetQTime ($VM_i$, Q)
            $PPOS \cup VM_i$
            if any two VMs give similar optimizer's cost then
                Add all the VMs in between these two VMs to the set XcludeVM
            end
        end
        for all the $VM_i \notin XcludeVM$ and PPOS do
            if $min(X_i) < X_j$ then
                GetQTime ($VM_i$, Q)
                $PPOS \cup VM_i$
            end
            if any two VMs give similar optimizer's cost then
                Add all the VMs in between these two VMs to the set XcludeVM
            end
        end
    end
    Filter the Pareto-optimal VMs from the set PPOS and add them to set $ForwardVM_n$
end
```

7.4 Empirical Evaluation

This section details the empirical performance of SPIK on Postgresql and **ComOpt**. To start with, we specify the performance metric used for this evaluation. Next, the implementation details as well as the performance results for both of these engines are discussed. We prototyped SPIK for Postgresql 9.3 and since, **ComOpt** is a commercial engine and does not provide access to its source, therefore, we implemented SPIK as a Java wrapper program with it.

7.4.1 Performance metric

Unlike PIK , this algorithm processes the VMs at a sub-plan level, thus, the number of VMs processed changes at each node of the plan. Hence, evaluating it on the previously used metric – cardinality of PPOS would not be fair. Instead, we propose the following performance metric to evaluate SPIK.

Total Costing Calls: The total computation done by SPIK for a given query is the sum total of the processing done at each DP-node of the query. We refer to the computation done at a node as one costing call. Therefore, to evaluate the performance of SPIK we have used the total number of costing calls as our metric, denoted by σ. It is the summation of the number of VMs in the set PPOS for every DP node of the query, i.e.

$$\sigma = \sum_{i=1}^{n} |\mathsf{PPOS}| \text{ } \textit{at node i} \tag{7.4}$$

where n is total number of DP-nodes in query. For the brute-force approach, value of $|\mathsf{PPOS}|$ at every node is equal to the total number of available VMs.

7.4.2 Results on Postgresql

Implementation Details on Postgresql 9.3

Postgresql 9.3 has a single threaded architecture, which means that for a single query it does not use the parallelism provided by the underlying system. So, the only system related parameter considered in query optimization for our model, is memory. The most influential configuration parameters in Postgresql are `work_mem` and `effective_cache_size` [14]. We use an array of main-memory values to simulate different VM configurations to the optimizer, which determine the corresponding values of `work_mem` and `effective_cache_size`. The alterations done in Postgresql 9.3 to implement SPIK are explained next.

Modifications in Data-structures: Unlike traditional query optimization, in SPIK we store plans for a set of VMs, and along with usual plan information it requires money expenditure of plans. In the work flow of Postgresql 9.3, firstly query paths are constructed, and then they are converted to query plans, which are used in `explain` utility to output cheapest cost query plan. We made following changes to *Path* and *Plan* data structures to implement SPIK .

- ***Path:*** this data-structure is defined in *plannodes.h*, and it saves the access paths for join or base relations. To distinguish the access paths for different VMs, we add variables for `work_mem` and `effective_cache_size`. Now, for each path, its corresponding memory parameters are also stored, which identifies the corresponding VM.

- ***Plan:*** This is defined in *plannodes.h* and saves the final plan. SPIK requires comparison on money as well as response-time of plans, so we add *money* variable to *Plan.* Similar to *Path*, to distinguish the plans for different VMs we add variables for `work_mem` and `effective_cache_size`, these values are passed from the respective *paths* while conversion of *path* to *plan.*

Saving Pareto-optimal Set of Plans: In Postgresql 9.3, plans are filtered such that only the interesting order or cheapest costs plans are saved. For SPIK, we modified the plan filtering mechanism such that they are pruned only when dominated in terms of both time as well as money. Thus, at each node, all Pareto-optimal paths are saved and forwarded to higher nodes. To implement this mechanism, we have to make changes in the following parts of the optimizer:

- Costing for the given set of VMs
- Modify the plan selection mechanism

The costing functions are modified to calculate the total money expenditure of the operator. Now, each operator has a cost and money expense associated with it. The plans that have more cost as well as money than some other plan of that node are pruned. We modified the pruning routine of Postgresql 9.3, to prune only when the plan is dominated in terms of both optimizer cost and money or it is an interesting order plan.

1. For every node, plans for a number of different VMs are created. Firstly we change the values of `work_mem` and `effective_cache_size` using the sub-routine *SetConfigOption*, to reflect the configuration of the chosen VM. Now, while costing of operators these new values of parameters are used. We calculate the total money expenditure for each operator and save it in the respective *Path* structure.

2. When creating the plans for a node, we use all the saved Pareto-optimal plans of the children nodes. It is ensured that considered sub-plans are for weaker or equivalent of the current VM.

3. The plans with more optimizer cost and money expenditure than some other plan at that node are pruned. All the remaining plans are saved since they form the Pareto-optimal set of the node. The plans with interesting order are also saved irrespective of their cost and money values compared to their non-interesting counterparts. However, among the interesting order plans the dominated ones are pruned. Finally, at the root node, the one

with minimum Euclidean distance on XTS is selected as the knee plan, and given as the output of the explain utility. Note that if user wants to know all the Pareto-optimal plans for the query, modifications can be done to output this set of plans.

Amendments were made in the costing functions of operators in *costsize.c*, in *add_path* routine of *pathnode.c*, *make_join_rel* function of *joinrels.c*, and in modules of *createplan.c* to convert knee path to plan.

Performance Results

The testbed for these experiments is a SUN Utra 20, AMD-Opteron workstation with dual core 2.5 GHz, 4GB RAM and two 240GB hard disks, running Ubunutu 12.04. We have evaluated the algorithm for both DI as well AI schema on TPC-DS benchmark queries, with database size being 100 GB. We have also calculated the computation overheads for SPIK in terms of memory and time.

An array of 40 different memory values ranging from 1 GB to 200 GB is given as the input to optimizer, along with query. A difference of 5 GB is maintained between consecutive memory values. The parameter `effective_cache_size` is given 50% and `work_mem` 5% of the given memory size.

As mentioned before, Postgresql uses a single threaded architecture for a given query, thus, there are no interesting variations in the behavior of the queries on multi-core machines. Hence, we compared PIK with exhaustive approach on Postgresql. However, on **ComOpt** we compared the performance of PIK and SPIK which is given later in Section 7.4.3.

Comparison of σ values for SPIK and brute-force is given in Figure 7.6. It is evident from the chart that the value of σ is at most 30% of the brute-force. To give a clearer picture of the optimization phase, we present cardinality of POS at the root and its maximum size in the plan tree in Table 7.1. This table illustrates that only a small number of VMs are Pareto-optimal and with our pruning mechanism, only few reach the root of the plan tree. Thus, avoiding exponential breakout of plans at higher nodes. For instance in AI-DSQ-17 and AI-DSQ-29, although there are 12 plans in the POS at intermediate nodes, but only half of them make it to the POS of the root node, rest are pruned in between.

Variations in the selected query-plan: The changes in the selected plan with the introduction of monetary expenditure metric is given in Figure 7.7 for DI-DSQ-45 and in Figure 7.8 for AI-DSQ-61. To show the difference between the two plans, the nodes in Figures 7.7b and 7.8b are outlined by red if they are different from their knee plan counterparts.

It is clear from these figures that when response-time alone is not the deciding metric, join algorithms as well as join order change. Also, hash join appears to be the cheapest alternate when only the optimizer's cost is to be minimized. However, when both money and time are taken into consideration, nested-loop join surfaces as a better option though not the fastest, because it requires lesser memory and thus a cheaper VM. Note that the VM corresponding to the knee plan gives a different value of memory parameters than default values of Postgresql. The plans we present in Figures 7.7b and 7.8b are with the parameter values of the knee VM and not the default parameter values.

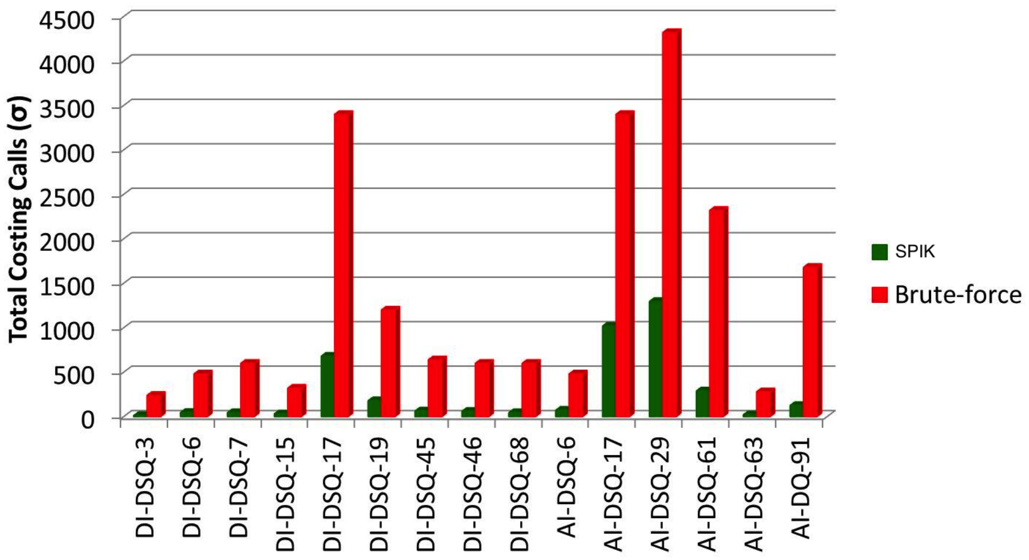
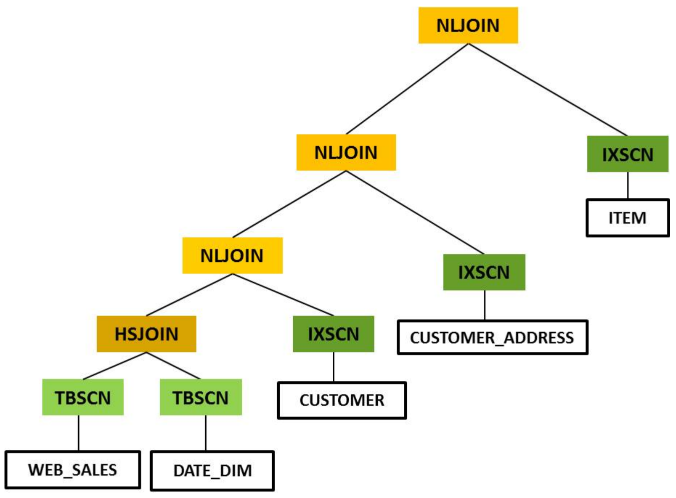


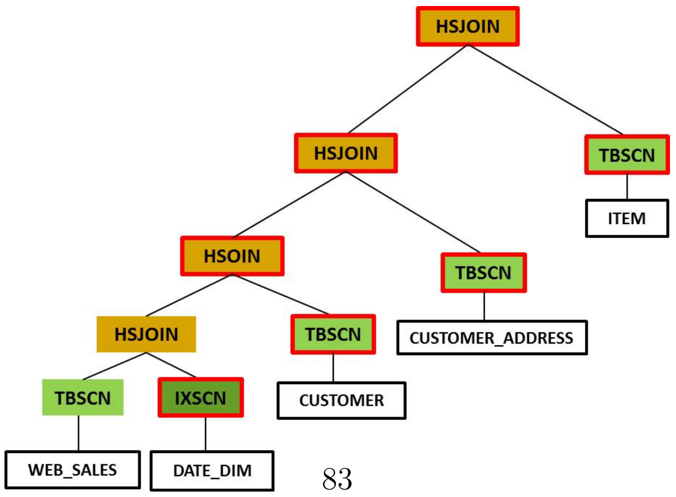
Figure 7.6: Performance of SPIK on Postgresql 9.3

Query	$\vert$POS$\vert_{max}$	$\vert$POS$\vert_{root}$
DI-DSQ-6	4	4
DI-DSQ-7	3	2
DI-DSQ-15	4	2
DI-DSQ-17	8	3
DI-DSQ-19	6	4
DI-DSQ-45	4	2
DI-DSQ-46	4	3
DI-DSQ-68	3	2
AI-DSQ-6	6	4
AI-DSQ-17	12	6
AI-DSQ-29	12	7
AI-DSQ-61	5	2
AI-DSQ-63	3	2
AI-DSQ-91	3	2

Table 7.1: Cardinality of POS for TPC-DS queries on Postgresql 9.3

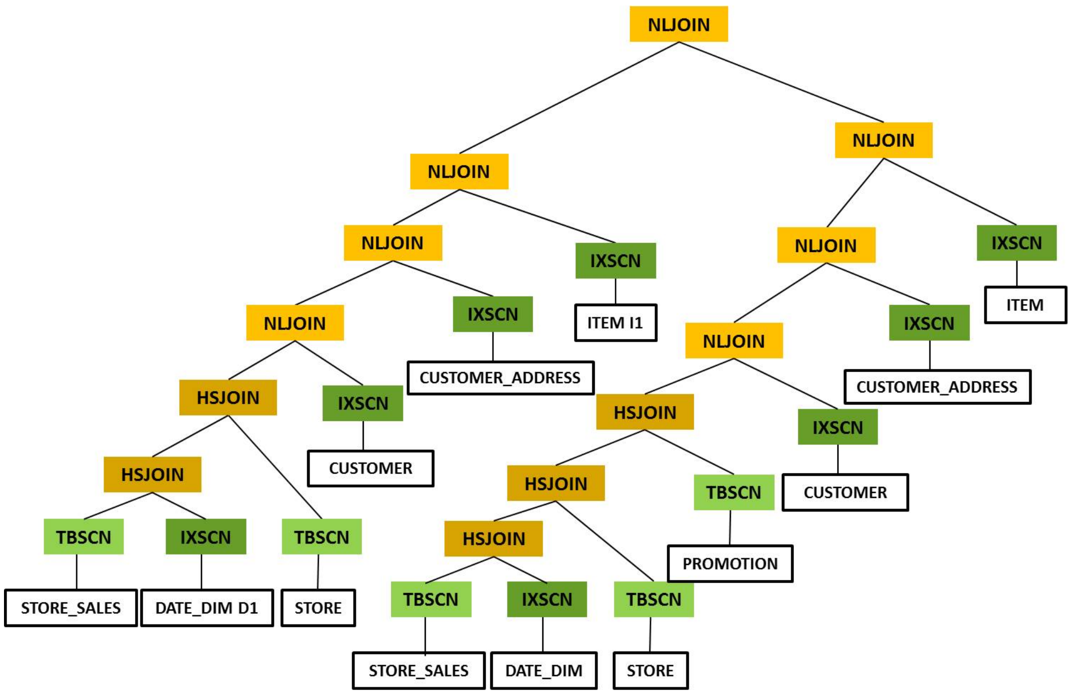


(a) Knee query plan for DI-DSQ-45 by SPIK

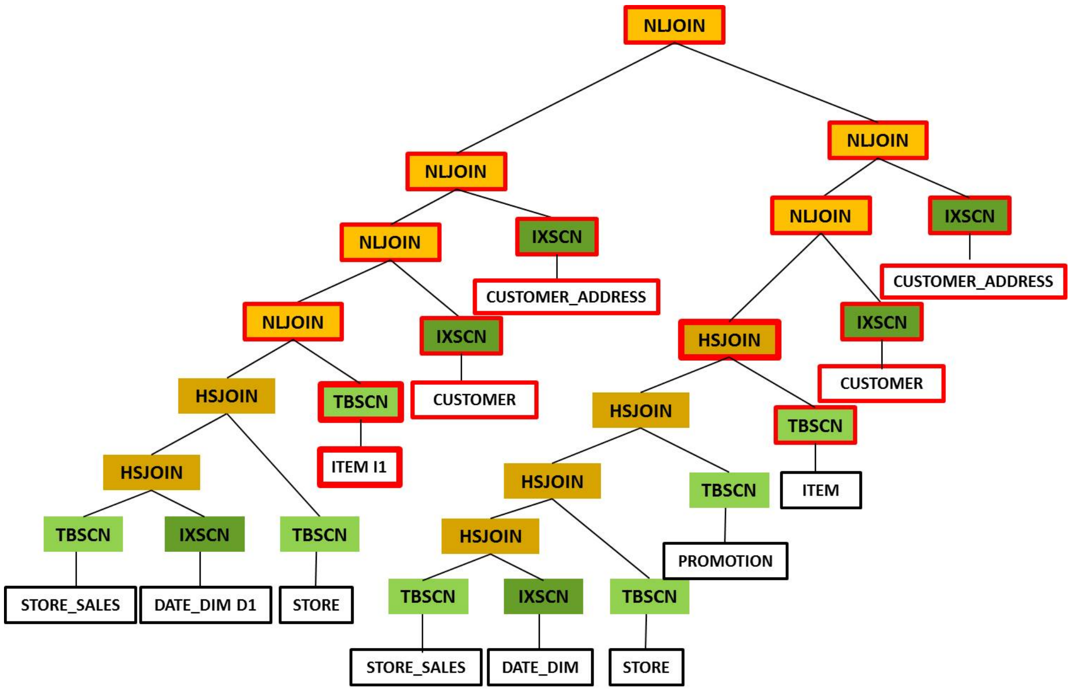


(b) Query plan for DI-DSQ-45 by native Postgresql 9.3

Figure 7.7: Query plans of DI-DSQ-45



(a) Knee query plan for AI-DSQ-61 by SPIK



(b) Query plan for AI-DSQ-61 by native Postgresql 9.3

Figure 7.8: Query plans of AI-DSQ-61

Computation overheads

Now, we take a look at the overheads paid for identifying the knee VM for the given set of configurations in terms of increased optimization time and memory. Time aspect of SPIK compared to native DP algorithm is captured in Table 7.2a, and Table 7.2b gives the account of extra memory required by SPIK during optimization. Clearly, optimization time taken by this new algorithm, which takes tens of VM configurations as input and identifies knee among them is around one second for the majority of queries. Similarly, peak memory expenditure is less than 10 MB, which appears acceptable overhead for todays' rich computing systems leveraging the benefits of cloud platform.

However, for some queries e.g. DI-DSQ-17, AI-DSQ-17, and AI-DSQ-61, the optimization time as well as peak memory is high because they have a large number of DP nodes. Specifically, DI-DSQ-17 and AI-DSQ-17 have 85 and AI-DSQ-61 has 58 DP nodes. When the number of DP nodes is this large, storing Pareto-optimal plans at each node is costly in terms of both time and memory, and the overall optimization is slow for such queries.

Query	SPIK (ms)	Native DP (ms)
DI-DSQ-6	200	14
DI-DSQ-7	150	10
DI-DSQ-15	80	12
DI-DSQ-17	750	45
DI-DSQ-19	200	18
DI-DSQ-45	150	15
DI-DSQ-46	80	15
DI-DSQ-68	100	15
AI-DSQ-6	300	30
AI-DSQ-17	1500	120
AI-DSQ-61	2000	90
AI-DSQ-63	150	25
AI-DSQ-91	700	40

(a) Time overheads of SPIK

Query	SPIK (KB)	Native DP (KB)
DI-DSQ-6	285	280
DI-DSQ-7	280	220
DI-DSQ-15	285	220
DI-DSQ-17	7200	250
DI-DSQ-19	330	220
DI-DSQ-45	250	220
DI-DSQ-46	6500	2600
DI-DSQ-68	290	220
AI-DSQ-6	240	220
AI-DSQ-17	600	280
AI-DSQ-61	550	260
AI-DSQ-63	230	220
AI-DSQ-91	300	220

(b) Memory overheads of SPIK

Table 7.2: Computation overheads of SPIK

7.4.3 Results on ComOpt

Since, **ComOpt** is a commercial engine, therefore it is not possible for us to modify its query optimizer. Hence, we implemented it as a Java wrapper program around **ComOpt** , to evaluate the performance of SPIK. This program obtains the complete query plans given by the explain utility of **ComOpt** for the required VMs and then compares their sub-plans. The comparison of sub-plans is not possible when the DP-nodes are different across VMs, since, that information is missing in the optimizer's chosen plan. Also, only those DP nodes are considered by this program that are in the optimizer's chosen plan, the information for the rest of the nodes is missed by this implementation. Hence, certain coarseness is expected in these empirical results.

To compare the performance of our two algorithms on **ComOpt** we calculated σ values for PIK as well. To calculate σ value for PIK, we summed up the cardinality of PPOS for each of the DP-nodes in the query execution plan. The comparative numbers for a few queries is given in Table 7.3. It is clear from this table that SPIK is within 40% of the PIK for most of the queries. However, for query DI-DSQ-3, the performance of PIK and SPIK are close to each other, it is because the number of Pareto-optimal VMs is small for this query and the number of nodes in the execution-plan is also low. Hence, the total processing done by PIK is also low for this query. Observe that this comparison is not exact, the ideal comparison could be done by comparing the optimization times for the two approaches after implementing SPIK inside **ComOpt** query optimizer.

Effect of time-threshold

The effect of time threshold (λ_T) on the efficiency of SPIK is given in Figure 7.9. The figure illustrates that often allowing a threshold of 20% on time, decreases the value of σ by upto 40%. Additionally, it is visible that for some queries the effect of λ_T is almost negligible. Note that this behavior is shown by the queries that already have a very low value of σ. According to the construction of SPIK , at every node atleast minimal and maximal VMs are tried. So,

Query	PIK (σ)	SPIK (σ)
DI-DSQ-3	24	14
DI-DSQ-6	376	140
DI-DSQ-19	790	223
DI-DSQ-25	280	90
DI-DSQ-75	460	65
AI-DSQ-4	120	40
AI-DSQ-24	300	60
AI-DSQ-67	469	22
AI-DSQ-74	240	42

Table 7.3: Comparison of σ for PIK and SPIK on **ComOpt**

the minimum value of σ is two times the number of nodes in the DP lattice and this remains unaffected by the value of λ_T .

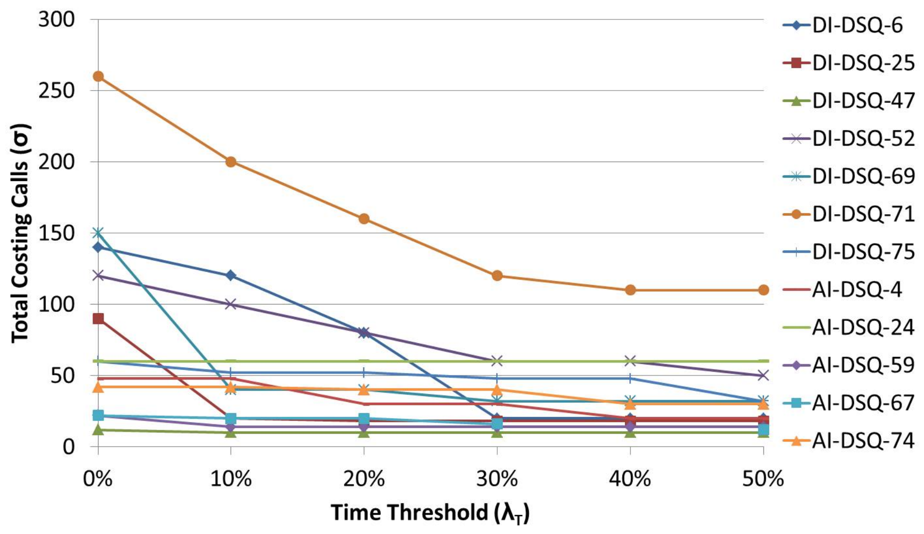


Figure 7.9: Effect of λ_T on SPIK for **ComOpt**

7.4.4 Summary

We prototyped SPIK on Postgresql 9.3, and found that it required only 30% of the processing as required by the exhaustive enumeration. To evaluate SPIK on **ComOpt**, we implemented it as a Java wrapper program, since modification in the **ComOpt** is not possible because of its commercial nature. On comparing the two algorithms on **ComOpt** , we found that SPIK requires only 40% of the computation as compared to PIK. Thus, the pain of intrusion in the query optimizer seems worth the gain in the performance of the knee identification.

Chapter 8

Conclusions and Future Work

The flexibility of infrastructure provided by the cloud platform encouraged migration of RDBMSs to cloud and opened many interesting arenas in the database system research. One of them is to optimize monetary investment along with query response-time. In the traditional multi-objective database query optimization, money is commonly ignored objective [16]. Since in traditional setups money consumption is more or less static, as infrastructural framework is worked up once and for all. However, IaaS model of cloud provides the capability of renting a different machine configuration for each of our queries, this raises the question of identifying the best configuration for a given query and database.

In this thesis, we considered the problem of balancing money and time investment for a given query and database. We used the concept of Pareto-optimality, which is the standard notion of optimality in multi-objective optimization. But, instead of overwhelming the user with too many details of all the query-plans, etc. on the Pareto-front, we chose to output just the knee VM and corresponding plan.

We analyzed the performance of a popular commercial database engine **ComOpt** on the Google cloud platform using benchmark queries of standard decision support benchmark database TPC-DS. We found that the execution-time of queries vary greatly with the changes in VM configuration, and so does the monetary expenditure. Additionally, this experimentation confirmed that there was no one VM that could act as the knee VM for all the queries. Hence, choosing the right VM for executing queries when migrating to the cloud framework is of paramount importance.

To identify the knee VM among hundreds of available configurations, we proposed an approach (PIK) that uses the partial ordering on the resources of the VMs. The algorithm was based on two main observations:

- the set of VMs for the relation $\preceq$ forms a poset
- if a VM dominates another on RS and both have similar response-times, then all the VMs bounded by them would also have same response-times

Therefore, once the maximal and minimal VMs of the poset with similar response-time were identified, all the VMs bounded by them were pruned. However, a noteworthy difference in the response-times on the two boundary VMs signals the presence of the knee VM in the poset. We then created a subset of VMs by excluding previously selected minimal and maximal VMs, and processed this subset for the knee VM. We continued this search till all the VMs were exhausted or we reached a smaller subset with no difference in response-times. Finally, we calculated Euclidean distance of each processed VM from origin on the XTS , and selected the one with minimum distance as the knee VM.

The empirical evaluation of PIK was done on **ComOpt** for the benchmark queries of TPC-DS database of size 100 GB. It was found that for the majority of the queries, PIK identified the knee VM by processing only 20-30% of the available VMs. To further decrease the number of processed VMs, a user-defined threshold (λ_T) on time was used. Giving a mere value of 10-20% to λ_T , the number of processed VMs reduced significantly.

However, PIK was relying on the response-time of the queries alone without considering the query specific details as the query plan. On observing the query plans across the VMs for a given query, it was found that often sub-plans were repeated. Thus, while computing the optimal plan for each required VM, redundant processing was done for the same sub-plan(s). This propelled us to make changes in the query optimizer module such that it can omit this redundant computation by pruning the VMs at sub-plan levels also. The additional input to this new query optimizer was the set of VM configurations, and it selected the knee VM and the corresponding plan as the final output.

The proposed sub-plan based algorithm – SPIK , used the concept of partially order set on the VMs and saved only Pareto-optimal VMs at each node of the query plan. Also, it ensure that no interesting VM went missing, it checks if the minimum possible monetary expenditure of the dominated VM is lesser than some stronger and dominating VM.

To empirically evaluate SPIK , we prototyped SPIK for Postgersql 9.3, and also implemented it as a Java wrapper program with **ComOpt** . Our experimental results indicate that the total computation carried out by SPIK is within 40% of PIK . Further, the efficiency of the algorithm increases significantly when a relaxation factor of 20 to 30 % is permitted on the time axis.

Therefore, from an overall perspective, this thesis facilitates the desired migration of enterprise databases to cloud platforms, by identifying the VM(s) that offer competitive tradeoffs between money and time for the given query.

8.1 Future Work

It would be interesting to extend this work for the query workload, which would be useful in pragmatic setups. Using the algorithms given in this thesis, we can identify the knee VM for individual queries of the workload, and execute them accordingly. However, if the database used by the queries is same and the size of workload is large then keeping multiple copies of the database might be cumbersome. In other words, if we aim at minimizing the number of VMs also, then we can identify the near-optimal knee VMs for some of the queries and execute them accordingly.

However, ideal solution for the workload is to find a knee VM that provides the best tradeoff between the money and time required by the workload. Similar to our approach, there might be two threads for this extension. The plan-based approach for can be extended for workload by assigning weights to the queries. The user-provided query weights would symbolize the importance of the respective query and the solution would give higher priority to the queries with greater weights. The objectives could be weighted too, signifying the worth of each, to the user. For example, if more weight is given to the response-time then solution VMs would complete queries as fast as possible, giving more importance to the queries with higher weight.

The balance between time and money can be attained by controlling the query optimization process by an extension in the multi-query optimization [23, 26]. It is a challenging task to add objective in the multi-query optimization for money. However, the motivation for the multi-query optimization was that many queries share the sub-plans. In Chapter 7, we have witnessed that often query plans across VMs have common sub-plans. Thus, it is a viable option to extend multi-query optimization for this problem. In that case, the query-plans would be chosen such that queries can leverage from this repetitive computation.

Furthermore, we have considered the problem only for the IaaS model of cloud, however it would be more challenging to study it for preemptive or spot instances on cloud [1, 4]. Instead of fixed rental rate given by cloud vendors, these VMs are available to users with highest bid, and the user has to preempt it as soon as someone else bids a higher value. The simultaneous optimization of monetary expenditure and query execution-time is tricky for such instances. It would be an interesting problem to decide not only the VM configuration but also time of the day for OLAP query executions. If the query is not executed in the stipulated time, the VM might be preempted resulting in failed query execution, hence some penalty function for incorrect estimation could be used.

Bibliography

[1] Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot/. 5, 92

[2] Compute engine - iaas google cloud platform. https://cloud.google.com/compute/. 4, 23, 29

[3] Google cloud pricing. https://cloud.google.com/pricing/. 7, 29

[4] Preemptible vm instances - compute engine google cloud platform. https://cloud.google.com/compute/docs/instances/preemptible. 5, 92

[5] Pricing - cloud services — microsoft azure. https://azure.microsoft.com/en-in/pricing/details/cloud-services/. 4, 7

[6] Types of cloud computing. https://aws.amazon.com/types-of-cloud-computing/. 2, 4, 23

[7] Understanding the cloud computing stack: Saas, paas, iaas. https://support.rackspace.com/white-paper/understanding-the-cloud-computing-stack-saas-paas-iaas/. 2, 23

[8] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. On the stability of plan costs and the costs of plan stability. *Proc. of 36th International Conference on Very Large DataBase (VLDB)*, pages 1137–1148, September 2010. 14

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, pages 164–177, October 2003. 4

[10] Jürgen Branke, Kalyanmoy Deb, Henning Dierolf, and Matthias Osswald. Finding knees in multi-objective optimization. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 722–731. Springer, 2004. 19, 23

[11] Massimiliano Caramia and Paolo Dell'Olmo. *Multi-objective management in freight logistics: Increasing capacity, service level and safety with optimization algorithms.* Springer Science & Business Media, 2008. 21, 22

[12] Indraneel Das and John E Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657, 1998. 21, 23

[13] Kalyanmoy Deb and Shivam Gupta. Understanding knee points in bicriteria problems and their implications as preferred solution principles. *Engineering optimization*, 43(11):1175–1204, 2011. 19, 20, 22, 23

[14] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *IEEE 24th International Conference on Data Engineering Workshop, 2008. ICDEW 2008*, pages 11–18. IEEE, 2008. 79

[15] Tansel Dokeroglu, Seyyit Alper Sert, and Muhammet Serkan Cinar. Evolutionary multiobjective query workload optimization of cloud data warehouses. *The Scientific World Journal*, 2014, 2014. 16, 18, 19

[16] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *ACM Sigmod Record*, 38(1):43–48, 2009. 18, 89

[17] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 9–18, New York, NY, USA, 1992. ACM. 14

[18] Antonio López Jaimes, Saúl Zapotecas Martınez, and Carlos A Coello Coello. An introduction to multiobjective optimization techniques. *Optimization in Polymer Processing*, pages 29–57, 2009. 21

[19] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975. 52

[20] Julien Legriel, Colas Le Guernic, Scott Cotton, and Oded Maler. Approximating the pareto front of multi-criteria optimization problems. In *TACAS*, pages 69–83. Springer, 2010. 26

[21] Peter Mell and Tim Grance. Draft nist working definition of cloud computing. *Referenced on June. 3rd*, 15:32, 2009. 2

[22] Jennie Rogers, Olga Papaemmanouil, and Ugur Cetintemel. A generic auto-provisioning framework for cloud databases. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW), 2010*, pages 63–68. IEEE, 2010. 6, 17, 18, 19

[23] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 249–260, New York, NY, USA, 2000. ACM. 92

[24] Ville Satopää, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *31st International Conference Distributed Computing Systems Workshops (ICDCSW), 2011*, pages 166–171. IEEE, 2011. 19, 23

[25] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34. ACM, 1979. 13, 68

[26] Timos K. Sellis. Multiple-query optimization. *ACM Transaction on Database Systems, ACM TODS 1988*, (1):23–52, March 1988. 92

[27] Pradyumn Kumar Shukla, Marlon Alexander Braun, and Hartmut Schmeck. Theory and algorithms for finding knees. In *Evolutionary Multi-Criterion Optimization*, pages 156–170. Springer, 2013. 19, 23

[28] Immanuel Trummer and Christoph Koch. Approximation schemes for many-objective query optimization. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1299–1310. ACM, 2014. 6, 14, 15, 18, 19

[29] Immanuel Trummer and Christoph Koch. An incremental anytime algorithm for multi-objective query optimization. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1941–1953, New York, NY, USA, 2015. ACM. 6, 16, 18

[30] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings on 39th International Conference on Very Large DataBase (VLDB)*, 6(10):925–936, August 2013. 35

[31] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigumus, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *IEEE 29th International Conference on Data Engineering (ICDE), 2013*, pages 1081–1092. IEEE, 2013. 35

[32] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. Exploring power-performance tradeoffs in database systems. In *IEEE 26th International Conference on Data Engineering (ICDE), 2010*, pages 485–496. IEEE, 2010. 14

Appendix

Query Text (based on benchmark queries)

```
select dt.d_year, item.i_brand_id brand_id, item.i_brand brand,
        sum(ss_ext_discount_amt) sum_agg
from date_dim dt, store_sales, item
where dt.d_date_sk = store_sales.ss_sold_date_sk
        and store_sales.ss_item_sk = item.i_item_sk
        and item.i_manufact_id = 783
        and dt.d_moy=11
group by dt.d_year, item.i_brand, item.i_brand_id
order by dt.d_year, sum_agg desc, brand_id;
```

TPC-DS Query 3

```
with year_total as
        (select c_customer_id customer_id, c_first_name customer_first_name,
                c_last_name customer_last_name, c_preferred_cust_flag
                customer_preferred_cust_flag, c_birth_country customer_birth_country,
                 c_login customer_login, c_email_address customer_email_address,
                d_year dyear, sum(((ss_ext_list_price-
                ss_ext_wholesale_cost- ss_ext_discount_amt)+ss_ext_sales_price)/2)
                year_total, 's' sale_type
        from customer, store_sales, date_dim
        where c_customer_sk = ss_customer_skand ss_sold_date_sk = d_date_sk
        group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag,
                   c_birth_country, c_login, c_email_address, d_year
        union all
        select c_customer_id customer_id, c_first_name customer_first_name, c_last_name
                customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
                c_birth_country customer_birth_country, c_login customer_login,
                 c_email_address customer_email_address, d_year dyear,
                sum((((cs_ext_list_price - cs_ext_wholesale_cost-cs_ext_discount_amt) +
                cs_ext_sales_price)/2) )
                year_total, 'c' sale_type
        from customer, catalog_sales, date_dim
        where c_customer_sk = cs_bill_customer_sk and cs_sold_date_sk = d_date_sk
        group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag,
                c_birth_country, c_login, c_email_address, d_year,
        union all
        select c_customer_id customer_id, c_first_name customer_first_name, c_last_name
            customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
            c_birth_country customer_birth_country, c_login customer_login,
            c_email_address customer_email_address, d_year dyear,
            sum((((ws_ext_list_price-ws_ext_wholesale_cost-ws_ext_discount_amt) +
            ws_ext_sales_price)/2) ) year_total, 'w' sale_type
        from customer, web_sales, date_dim
        where c_customer_sk = ws_bill_customer_skand ws_sold_date_sk = d_date_sk
        group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag,
            c_birth_country, c_login, c_email_address, d_year)
select t_s_secyear.customer_login
from year_total t_s_firstyear, year_total t_s_secyear, year_total t_c_firstyear,
        year_total t_c_secyear, year_total t_w_firstyear, year_total t_w_secyear
where t_s_secyear.customer_id = t_s_firstyear.customer_id
        and t_s_firstyear.customer_id = t_c_secyear.customer_id
        and t_s_firstyear.customer_id = t_c_firstyear.customer_id
        and t_s_firstyear.customer_id = t_w_firstyear.customer_id
        and t_s_firstyear.customer_id = t_w_secyear.customer_id
        and t_s_firstyear.sale_type = 's' and t_c_firstyear.sale_type = 'c'
        and t_w_firstyear.sale_type = 'w' and t_s_secyear.sale_type = 's'
        and t_c_secyear.sale_type = 'c' and t_w_secyear.sale_type = 'w'
        and t_s_firstyear.dyear = 1999 and t_s_secyear.dyear = 1999+1
        and t_c_firstyear.dyear = 1999 and t_c_secyear.dyear = 1999+1
        and t_w_firstyear.dyear = 1999 and t_w_secyear.dyear = 1999+1
        and t_s_firstyear.year_total > 0 and t_c_firstyear.year_total > 0
        and t_w_firstyear.year_total > 0 and case when t_c_firstyear.year_total > 0
```

```
          then t_c_secyear.year_total / t_c_firstyear.year_total else null end > case
          when t_s_firstyear.year_total > 0 then t_s_secyear.year_total /
          t_s_firstyear.year_total else null end
      and case when t_c_firstyear.year_total > 0 then t_c_secyear.year_total /
          t_c_firstyear.year_total else null end > case when t_w_firstyear.year_total > 0
          then t_w_secyear.year_total / t_w_firstyear.year_total else null end
order by t_s_secyear.customer_login ;
```

TPC-DS Query 4

```
select a.ca_state state, count(*) cnt
from customer_address a, customer c, store_sales s , date_dim d, item i
where a.ca_address_sk = c.c_current_addr_sk
        and c.c_customer_sk = s.ss_customer_sk
        and s.ss_sold_date_sk = d.d_date_sk
        and s.ss_item_sk = i.i_item_sk
        and d.d_month_seq = (select distinct (d_month_seq)
                            from date_dim
                            where d_year = 1998
                                    and d_moy = 5 )
        and i.i_current_price > 1.2 * (select avg(j.i_current_price)
                                    from item j
                                    where j.i_category = i.i_category)
group by a.ca_state
having count(*) >= 10
order by cnt ;
```

TPC-DS Query 6

```
select i_brand_id brand_id, i_brand brand, i_manufact_id, i_manufact,
        sum(ss_ext_sales_price) ext_price
from date_dim, store_sales, item,customer,customer_address,store
where d_date_sk = ss_sold_date_sk
        and ss_item_sk = i_item_sk
        and i_manager_id=91
        and d_moy=12
        and d_year=2002
        and ss_customer_sk = c_customer_sk
        and c_current_addr_sk = ca_address_sk
        and substr(ca_zip,1,5) ¡¿ substr(s_zip,1,5)
        and ss_store_sk = s_store_sk
group by i_brand, i_brand_id, i_manufact_id, i_manufact,
order by ext_price desc, i_brand, i_brand_id, i_manufact_id, i_manufact;
```

TPCDS Query 19

```
with ssales as
        (select c_last_name, c_first_name, s_store_name, ca_state, s_state, i_color,
             i_current_price, i_manager_id, i_units,
             i_size, sum(ss_ext_sales_price) netpaid
        from store_sales, store_returns, store, item, customer, customer_address
        where ss_ticket_number = sr_ticket_number and ss_item_sk = sr_item_sk
             and ss_customer_sk = c_customer_sk and ss_item_sk = i_item_sk
             and ss_store_sk = s_store_sk and c_birth_country = upper(ca_country)
             and s_zip = ca_zip and s_market_id=8
        group by c_last_name, c_first_name, s_store_name, ca_state,
             s_state, i_color, i_current_price,
             i_manager_id, i_units, i_size)
select c_last_name, c_first_name, s_store_name, sum(netpaid) paid
from ssales
where i_color = 'lawn'
group by c_last_name, c_first_name, s_store_name
having sum(netpaid) > (select 0.05*avg(netpaid) from ssales);
```

TPCDS Query 24

```
select i_item_id, i_item_desc, s_store_id, s_store_name, max(ss_net_profit) as store_sales_profit,
        max(sr_net_loss) as store_returns_loss,max(cs_net_profit) as catalog_sales_profit
from store_sales, store_returns, catalog_sales, date_dim d1, date_dim d2,
        date_dim d3, store, item
where d1.d_moy = 4 and d1.d_year = 2001 and d1.d_date_sk = ss_sold_date_sk
        and i_item_sk = ss_item_sk and s_store_sk = ss_store_sk
        and ss_customer_sk = sr_customer_sk
        and ss_item_sk = sr_item_sk and ss_ticket_number = sr_ticket_number
        and sr_returned_date_sk = d2.d_date_sk and d2.d_moy between 4 and 10
        and d2.d_year = 2001 and sr_customer_sk = cs_bill_customer_sk
        and sr_item_sk = cs_item_sk and cs_sold_date_sk = d3.d_date_sk
        and d3.d_moy between 4 and 10 and d3.d_year = 2001
group by i_item_id, i_item_desc, s_store_id, s_store_name
order by i_item_id, i_item_desc, s_store_id, s_store_name;
```

TPCDS Query 25

```
with v1 as (select i_category, i_brand, s_store_name, s_company_name, d_year, d_moy,
                   sum(ss_sales_price) sum_sales, avg(sum(ss_sales_price)),
                   over (partition by i_category, i_brand, s_store_name,
                   s_company_name, d_year), avg_monthly_sales, rank()
                   over (partition by i_category, i_brand, s_store_name, s_company_name
                   order by d_year, d_moy) rn
          from item, store_sales, date_dim, store
          where ss_item_sk = i_item_sk and ss_sold_date_sk = d_date_sk
                  and ss_store_sk = s_store_sk and ( d_year = 2000 or
                  ( d_year = 2000-1 and d_moy = 12) or ( d_year = 2000+1 and d_moy = 1))
          group by i_category, i_brand, s_store_name, s_company_name, d_year, d_moy),
v2 as
          (select v1.s_store_name, v1.s_company_name, v1.d_year, v1.avg_monthly_sales,
                  v1.sum_sales, v1_lag.sum_sales psum, v1_lead.sum_sales nsum
          from v1, v1 v1_lag, v1 v1_lead
          where v1.i_category = v1_lag.i_category and v1.i_category = v1_lead.i_category
                  and v1.i_brand = v1_lag.i_brandand v1.i_brand = v1_lead.i_brand
                  and v1.s_store_name = v1_lag.s_store_name
                  and v1.s_store_name = v1_lead.s_store_name
                  and v1.s_company_name = v1_lag.s_company_name
                  and v1.s_company_name = v1_lead.s_company_name
                  and v1.rn = v1_lag.rn + 1 and v1.rn = v1_lead.rn - 1)
select *
from v2
where d_year = 2000 and avg_monthly_sales > 0
```

```
        and case when avg_monthly_sales > 0 then
        abs(sum_sales - avg_monthly_sales) avg_monthly_sales
        else null end > 0.1
order by sum_sales - avg_monthly_sales, 3;
```

TPCDS Query 47

```
select dt.d_year, item.i_brand_id brand_id, item.i_brand brand, sum(ss_ext_sales_price) ext_price
from date_dim dt, store_sales, item
where dt.d_date_sk = store_sales.ss_sold_date_sk and store_sales.ss_item_sk = item.i_item_sk
        and item.i_manager_id = 1 and dt.d_moy=12 and dt.d_year=2000
group by dt.d_year, item.i_brand, item.i_brand_id
order by dt.d_year, ext_price desc, brand_id;
```

TPC-DS Query 52

```
select i_brand_id brand_id, i_brand brand, sum(ss_ext_sales_price) ext_price
from date_dim, store_sales, item
where d_date_sk = ss_sold_date_sk and ss_item_sk = i_item_sk
        and i_manager_id = 40 and d_moy = 12 and d_year = 2001
group by i_brand, i_brand_id
order by ext_price desc, i_brand_id;
```

TPC-DS Query 55

```
with wss as (select d_week_seq, ss_stor e_sk, sum(case when (d_day_name='Sunday')
        then ss_sales_price else null end) sun_sales, sum(case when (d_day_name='Monday')
        then ss_sales_price else null end) mon_sales, sum(case when (d_day_name='Tuesday')
        then ss_sales_price else null end) tue_sales, sum(case when (d_day_name='Wednesday')
        then ss_sales_price else null end) wed_sales, sum(case when (d_day_name='Thursday')
        then ss_sales_price else null end) thu_sales, sum(case when (d_day_name='Friday')
        then ss_sales_price else null end) fri_sales, sum(case when (d_day_name='Saturday')
        then ss_sales_price else null end) sat_sales
        from store_sales,date_dim
        where d_date_sk = ss_sold_date_sk
        group by d_week_seq,ss_store_sk)
select s_store_name1,s_store_id1,d_week_seq1, sun_sales1/sun_sales2, mon_sales1/mon_sales2,
      tue_sales1/tue_sales1, wed_sales1/wed_sales2, thu_sales1/thu_sales2,
      fri_sales1/fri_sales2, sat_sales1/sat_sales2
from (select s_store_name s_store_name1,wss.d_week_seq d_week_seq1, s_store_id s_store_id1,
            sun_sales sun_sales1, mon_sales mon_sales1, tue_sales tue_sales1,
            wed_sales wed_sales1, thu_sales thu_sales1, fri_sales fri_sales1, sat_sales sat_sales1
      from wss,store,date_dim d
      where d.d_week_seq = wss.d_week_seq and ss_store_sk = s_store_sk
            and d_month_seq between 1184 and 1184 + 11) y,
      (select s_store_name s_store_name2,wss.d_week_seq d_week_seq2, s_store_id s_store_id2,
            sun_sales sun_sales2, mon_sales mon_sales2, tue_sales tue_sales2,
            wed_sales wed_sales2, thu_sales thu_sales2, fri_sales fri_sales2, sat_sales sat_sales2
      from wss,store,date_dim d
      where d.d_week_seq = wss.d_week_seqand ss_store_sk = s_store_sk
```

```
                and d_month_seq between 1184+ 12 and 1184 + 23) x
where s_store_id1=s_store_id2 and d_week_seq1=d_week_seq2-52
order by s_store_name1,s_store_id1,d_week_seq1;
```

TPC-DS Query 59

```
select *
from (select i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy, s_store_id,
            sumsales, rank() over (partition by i_category order by sumsales desc) rk
     from (select i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy,
                s_store_id, sum(coalesce(ss_sales_price*ss_quantity,0)) sumsales
           from store_sales, date_dim, store, item
           where ss_sold_date_sk = d_date_sk and ss_item_sk=i_item_sk
                and ss_store_sk = s_store_sk and d_month_seq between 1214 and 1214+11
           group by rollup (i_category, i_class, i_brand, i_product_name,
                                d_year, d_qoy, d_moy,s_store_id))dw1) dw2
where rk ≤ 100
order by i_category, i_class, i_brand, i_product_name, d_year,
         d_qoy, d_moy, s_store_id, sumsales, rk;
```

TPC-DS Query 67

```
select i_brand_id brand_id, i_brand brand,t_hour, t_minute, sum(ext_price) ext_price
from item, (select ws_ext_sales_price as ext_price, ws_sold_date_sk as sold_date_sk,
                   ws_item_sk as sold_item_sk, ws_sold_time_sk as time_sk
            from web_sales, date_dim
            where d_date_sk = ws_sold_date_sk and d_moy=12 nd d_year=2002
            union all
            select cs_ext_sales_price as ext_price, s_sold_date_sk as sold_date_sk,
                   cs_item_sk as sold_item_sk, cs_sold_time_sk as time_sk
            from catalog_sales,date_dim
            where d_date_sk = cs_sold_date_sk and d_moy=12 and d_year=2002
            union all
            select ss_ext_sales_price as ext_price, ss_sold_date_sk as sold_date_sk,
                   ss_item_sk as sold_item_sk, ss_sold_time_sk as time_sk
            from store_sales,date_dim
            where d_date_sk = ss_sold_date_sk and d_moy=12 and d_year=2002)
                    as tmp,time_dim
where sold_item_sk = i_item_sk and i_manager_id=1and time_sk = t_time_sk
       and (t_meal_time = 'breakfast' or t_meal_time = 'dinner')
group by i_brand, i_brand_id,t_hour,t_minute
order by ext_price desc, i_brand_id;
```

TPC-DS Query 71

```
with year_total as (select c_customer_id customer_id, c_first_name customer_first_name,
                           c_last_name customer_last_name, d_year as year,
                           max(ss_net_paid) year_total, 's' sale_type
                    from customer, store_sales, date_dim
                    where c_customer_sk = ss_customer_sk and ss_sold_date_sk = d_date_sk
                          and d_year in (1998,1998+1)
                    group by c_customer_id, c_first_name, c_last_name, d_year
                    union all
                    select c_customer_id customer_id, c_first_name customer_first_name,
                           c_last_name customer_last_name, d_year as year,
                           max(ws_net_paid) year_total, 'w' sale_type
                    from customer, web_sales, date_dim
                    where c_customer_sk = ws_bill_customer_sk
                          and ws_sold_date_sk = d_date_sk and d_year in (1998,1998+1)
                    group by c_customer_id, c_first_name, c_last_name, d_year)
select top 100 t_s_secyear.customer_id, t_s_secyear.customer_first_name,
         t_s_secyear.customer_last_name
from year_total t_s_firstyear, year_total t_s_secyear, year_total t_w_firstyear,
     year_total t_w_secyear
where t_s_secyear.customer_id = t_s_firstyear.customer_id
      and t_s_firstyear.customer_id = t_w_secyear.customer_id
      and t_s_firstyear.customer_id = t_w_firstyear.customer_id
      and t_s_firstyear.sale_type = 's' and t_w_firstyear.sale_type = 'w'
      and t_s_secyear.sale_type = 's' and t_w_secyear.sale_type = 'w'
      and t_s_firstyear.year = 1998 and t_s_secyear.year = 1998+1
```

```
        and t_w_firstyear.year = 1998 and t_w_secyear.year = 1998+1
        and t_s_firstyear.year_total > 0 and t_w_firstyear.year_total > 0
        and case when t_w_firstyear.year_total > 0 then t_w_secyear.year_total /
        t_w_firstyear.year_total else null end > case when t_s_firstyear.year_total > 0 then
        t_s_secyear.year_total / t_s_firstyear.year_total else null end
order by 1,2,3;
```

TPC-DS Query 74

```
with all_sales as (select d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id,
                   sum(sales_cnt) as sales_cnt, sum(sales_amt) as sales_amt
                       from (select d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id,
                              cs_quantity - coalesce(cr_return_quantity, 0) as sales_cnt,
                              cs_ext_sales_price - coalesce(cr_return_amount, 0.0) as sales_amt
                              from catalog_sales join item on i_item_sk=cs_item_sk
                              join date_dim on d_date_sk=cs_sold_date_sk
                              left join catalog_returns on (cs_order_number=cr_order_number
                                                        and cs_item_sk=cr_item_sk)
                              where i_category='Shoes'
                  union
                  select d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id,
                         ss_quantity - coalesce(sr_return_quantity,0) as sales_cnt,
                         ss_ext_sales_price - coalesce(sr_return_amt,0.0) as sales_amt
                  from store_sales join item on i_item_sk=ss_item_sk
                       join date_dim on d_date_sk=ss_sold_date_sk
                       left join store_returns on (ss_ticket_number=sr_ticket_number
                                                and ss_item_sk=sr_item_sk)
                  where i_category='Shoes'
                  union
                  select d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id,
                         ws_quantity - coalesce(wr_return_quantity,0) as sales_cnt
                         ws_ext_sales_price - coalesce(wr_return_amt,0.0) as sales_amt
                  from web_sales join item on i_item_sk=ws_item_sk
                         join date_dim on d_date_sk=ws_sold_date_sk
```

```
                              left join web_returns on (ws_order_number=wr_order_number
                                                          and ws_item_sk=wr_item_sk)
                        where i_category='Shoes') sales_detail
                   group by d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id)
select top 100 prev_yr.d_year as prev_year, curr_yr.d_year as year,
             curr_yr.i_brand_id, curr_yr.i_class_id, curr_yr.i_category_id, curr_yr.i_manufact_id,
             prev_yr.sales_cnt as prev_yr_cnt, curr_yr.sales_cnt as curr_yr_cnt,
             curr_yr.sales_cnt-prev_yr.sales_cnt as sales_cnt_diff,
             curr_yr.sales_amt-prev_yr.sales_amt as sales_amt_diff
from all_sales curr_yr, all_sales prev_yr
where curr_yr.i_brand_id=prev_yr.i_brand_id
       and curr_yr.i_class_id=prev_yr.i_class_id
       and curr_yr.i_category_id=prev_yr.i_category_id
       and curr_yr.i_manufact_id=prev_yr.i_manufact_id
       and curr_yr.d_year=2000
       and prev_yr.d_year=2000-1
       and cast(curr_yr.sales_cnt as decimal(17,2))/
               cast(prev_yr.sales_cnt as decimal(17,2)) < 0.9
order by s_sales_cnt_diff;
```

TPC-DS Query 75