HOSTED BY
ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/jides

A bi-objective cost model for optimizing database queries in a multi-cloud environment

*Anastasios Gounaris**, *Zisis Karampaglis*, *Athanasios Naskos*, *Yannis Manolopoulos*

Aristotle University of Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 23 October 2014
Received in revised form
19 January 2015
Accepted 15 February 2015
Published online 5 March 2015

Keywords:
Cost model
Cloud
Query optimization
Data flow

ABSTRACT

Cost models are broadly used in query processing to drive the query optimization process, accurately predict the query execution time, schedule database query tasks, apply admission control and derive resource requirements to name a few applications. The main role of cost models is to estimate the time needed to run the query on a specific machine. In a multi-cloud environment, cost models should be easily calibrated for a wide range of different physical machines, and time estimates need to be complemented with monetary cost information, since both the economic cost and the performance are of primary importance. This work aims to serve as the first proposal for a bi-objective query cost model suitable for queries executed over resources provided by potentially multiple cloud providers. We leverage existing calibrating modeling techniques for time estimates and we couple such estimates with monetary cost information covering the main charging options for using cloud resources. Moreover, we explain how the cost model can become part of an optimizer. Our approach is applicable to more generic data flow graphs, the execution plans of which do not necessarily comprise relational operators. Finally, we give a concrete example about the usage of our proposal and we validate its accuracy through real case studies.

1. Introduction

More and more companies and organizations consider moving their infrastructures and applications on the cloud, motivated by the promise of clouds to achieve economies of scale. One of the most attractive features of cloud computing is that it provides an alternative to the procurement and management of expensive computing resources, which are associated with high upfront investments and considerable human effort, respectively.

Cloud technology has evolved significantly and nowadays, it is considered as robust and trustworthy. It leverages several traditional notions of distributed computing, such as the virtualization of resources and the provision of *virtual machines (VMs)*, typically at a certain monetary cost. Such VMs come with several names depending on the provider;

Peer review under responsibility of Qassim University.

* *Corresponding author*.

E-mail addresses: gounaria@csd.auth.gr (A. Gounaris), zkarampa@gmail.com (Z. Karampaglis), anaskos@csd.auth.gr (A. Naskos), manolopo@csd.auth.gr (Y. Manolopoulos).

http://dx.doi.org/10.1016/j.jides.2015.02.002

for example, Google calls them *"machine types"*, Amazon calls them *"instance types"* whereas other names include *"server sizes"* [1], but in all cases they refer to the provision of specific hardware combination of compute, memory and I/O resources. Cloud resources are not limited to emulations of raw physical machines; they can also cover provision of software middleware, databases and specialized tools. The proliferation of cloud options has raised the following problem faced by cloud users: *which cloud providers should be chosen to execute a specific task on the cloud?* This issue is not only important but also complex, especially when the requested resources can be offered by multiple providers. A key point to answer this question is to provide estimates of both the running time and the monetary cost; this is exactly the topic of our work.

We focus on database queries and more generic data-flow tasks that can be executed over remote resources provided by multiple providers [2,3]. For example, assume a database query that joins data from cloud-enabled data stores, such as anonymized population census data and commercial data offered by a set of providers. Or, analyzing patient data using a series of specialized cloud-enabled services, as described in [4]. In such scenarios, to be in a position to take final allocation decisions, we need to be able to accurately estimate the running time of the tasks on cloud resources and the price to use such resources.

Estimating query execution time plays an important role in several applications and processes, including query optimization, scheduling, admission control and allocation of resources [5]. Typically, the query cost models are applied to physical execution plans and assume that the physical resources to be employed in query execution are predefined. These cost models either encapsulate a component to estimate the cardinalities of the data processed or accept such cardinality statistics as input; in the output, they produce an estimate of the query running time. Examples of such cost models in a distributed environment are provided in [6,7]. In a multi-cloud environment, providing information only about the running time for specific processors is insufficient because: (i) the machines employed are not defined *a-priori*, and (ii) time estimates need to be complemented with monetary cost information. We directly address the second shortcoming, while, for the first one, we leverage existing techniques that provide practical solutions to calibrating generic cost models for different physical machines.

The main contribution of this work is the proposal of a database query cost model that provides estimates of both the expected running time and the economic cost associated with running a specific query over VMs provided by one or more cloud providers. The cost model is modular and can be applied to arbitrary DAG (directed acyclic graph) data flows apart from simple query execution plans consisting of relational operators. It supports the main modes of fee charging to date, which leverage the pay-as-you-go approach. Nevertheless, the modularity of our proposal allows for easily plugging-in further models for running time and cost estimates, while the model is not tailored to any specific charging policy. In this work, we show how our proposal can be used to derive running time and monetary cost estimates through a detailed example and a validation case study on a real cloud infrastructure. Last but not least, we explain how the cost model can be fitted into a state-of-the-art optimizer, which requires as its input pairs of time and money cost estimates for different execution options.

The reason we focus on these two objectives is that time and economic cost are basically anti-correlated in a way that is not easy to be modeled analytically. For example, we cannot claim that the two metrics are inversely proportional due to the wide range of factors involved, such as the types of the machines and the charging policies. In general, using multiple physical machines to execute queries is known that can yield higher performance if employed judiciously [8,6] at a higher energy cost [9], while using more machines incurs extra money cost on the cloud. Our modeling proposal makes such trade-offs explicit. Additional relevant criteria include security, reliability, availability and so on, but we leave their inclusion into a cost model as an interesting direction for future research.

The remainder of this article is structured as follows. In the next section, we discuss related work. We provide background material in Section 3 regarding the most prominent pricing policies adopted by modern cloud providers, the factors that affect the cost associated with a cloud infrastructure, and the multi-cloud optimizer that our proposal aims to directly support. In Section 4, we give the details of our cost model. Section 5 deals with the validation case studies. We conclude in Section 6.

This article is a significantly extended and modified version of the paper in [10]. The main parts of the new material comprise the following items: (i) extensions to the evaluation with additional real experiments to consider multi-cloud settings (discussed in Section 5.3); (ii) extensions to the discussion on how the cost model can be merged with optimizers (discussed in Section 4.3); (iii) clarifications on the assumptions and extensions to the cost model to better consider parallelism and overlaps in the time domain (discussed in Sections 4.1 and 4.2.2, respectively); and (iv) elaboration on the existing cost models for distributed queries (discussed in Section 2).

2. Related work

Traditional cost models for distributed queries are single-objective; they do not consider the economic cost associated and are also limited to scenarios where the estimates are in abstract time units rather than trying to predict the actual running time. Note that it is adequate for single-objective optimizers to be based on abstract time units because their aim is to relatively compare the alternative plans in terms of performance, and for this reason, the cost models employed mostly focus on data statistics, histograms and so on [11]. However, this convenient simplification becomes a strong limitation when predicting actual monetary cost, since the latter is a function of actual running time.

A typical approach to predicting time estimates is presented in [6], where the query execution time is split into the time needed to execute CPU tasks, retrieve and store data to the disk and send data across hosts. More specifically, the total cost is given by the following formula:

$$TotalCost = c_{cpu}\#instructions + c_{I/O}\#I/Os + c_{msg}\#messages + c_{tr}\#bytes$$

where c_{cpu}, $c_{I/O}$, c_{msg} and c_{tr} denote the cost in time units to execute an instruction on the cpu, perform a disk I/O access, initialize and receive a message and transmit a byte over the network, respectively. Apparently, the last two components correspond to the communication cost and differentiate this cost model from its counterparts for centralized databases. If we are interested in the wall-clock time, which will be referred to in this work as *TotalTime*, then time overlaps need to be considered.

More detailed cost functions are provided in [7]; however, this work follows the same approach as in [6]. The additional details correspond to splitting each relational operator into its more elementary operations, which is commonly done for non-distributed database queries (e.g., [5]).

More sophisticated approaches to distributed cost modeling, such as [12,13], perform more efficient and dynamic cost function calibration but still suffer from the main limitation mentioned above, that is they do not combine time costs with monetary costs. Nevertheless, for the time cost estimates, our work can regard time estimate techniques as a black box, and as such, can fully encapsulate them; later we show how we do this with regards to [5] and [14]. The work in [5] is a recent technique that provides actual time estimates for non-distributed queries and can be easily calibrated for any type of machines. This makes it attractive for cloud settings. Other techniques that are capable of providing actual time estimates rely on statistical machine learning (e.g., [15,16]) and cannot be easily calibrated for machines not specified *a-priori*. Complementarily, [14] describes how we can model network connections and data communication costs. Two other approaches to calibrating cost models appear in [17,18].

Another related research area is to develop query progress indicators (e.g., [19,20]). However, such proposals aim to dynamically report on the expected completion time of a query at runtime based on continuous feedback, rather than on providing accurate time estimates before the query execution begins. Finally, the work in [21] defines the trade-off between performance and cost, when running an application over a different number of VMs of the same type in the same data center under volatile load. Our problem is different, since we consider cases where the applications consist of several subtasks that can run on different types of VMs, which are possibly provided by multiple providers.

3. Background

Before delving into cost model's details, we provide a brief overview of the main pricing policies currently adopted by cloud providers and are meant to be supported by our cost model. We also discuss the main factors involved in the cost of developing and maintaining a cloud infrastructure. In the last part of this section, we present the currently established cost models for distributed queries, which do not consider economic costs. Finally, we show how our proposal complements bi-objective optimizers that are suitable for multi-cloud database queries.

3.1. *Cloud pricing policies and costs*

Cloud providers offer VMs at a specific price. The price depends on several factors including the computational characteristics of the VM, the reservation time and mechanism, and whether the VM comes with specific software installed (e.g., as typically occurs in PaaS/SaaS settings) or not. The price of VMs typically differs among providers, even when the offered VMs share the same characteristics.

3.1.1. *VM characteristics related to charging*

A main characteristic that affects the charging fee is the exact type and volume of computational resources that each client requests. There is a significant deviation in the price depending on CPU speed,[1] memory and storage space. Usually providers have some fixed combinations of the above components, so that they offer complete pre-specified VM options to users aiming to cover a broad range of needs. In addition, some providers allow their customers to build their own combination of resources, i.e., to customize their VMs. Some examples are Amazon Web Services[2] and CloudSigma,[3] respectively. Finally, some providers, like Amazon[4] and Rackspace[5] follow a hybrid approach and offer additional storage space with extra cost on top of pre-specified VM instances.

Installed software is equally important. The software that is used (e.g., databases, operating systems, and so on) may be open-source or commercial. Generally, VMs with pre-installed open source software are less expensive than those with commercial software due to licensing fees. The built-in support for providing programming frameworks (e.g., Apache Hadoop) increases the cost. The same holds for non-functional features, such as monitoring services, security features, ease of migration, and so on. Other price factors are the data transfer and geographical position of VMs. Some providers have additional charges for data transfer either from or to their servers and also apply different rates depending on the geographical location of the servers running the VMs (e.g., Amazon's EC2 policy[6]).

A cost model needs to be VM-independent to be usable in a multi-cloud environment. Our cost model is not tailored to any specific hardware characteristics. Rather, it provides generic formulas that can be calibrated according to the specific VM types at the disposal of a consumer of cloud resources.

3.1.2. *Cloud costs*

The development and maintenance of cloud infrastructure requires the investment of a big amount of money that is amortized over a long time period. Except from the initial purchase costs, a cloud data center is expensive to maintain and run. The major development cost is the acquisition of

[1] CPU power is often abstracted through the use of the so-called ECU units.

[2] http://aws.amazon.com/ec2/instance-types/.

[3] http://www.cloudsigma.com/#features.

[4] http://aws.amazon.com/ebs/.

[5] http://www.rackspace.com/cloud/block-storage.

[6] http://aws.amazon.com/ec2/pricing/.

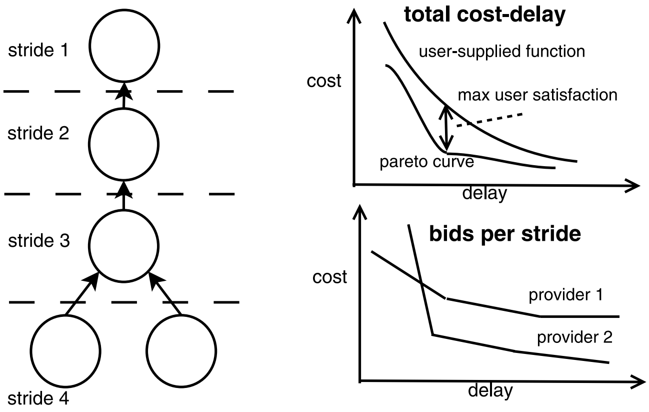

Fig. 1 – An analysis DAG (left) and example user and provider cost-delay functions (right) [25].

the raw servers and network infrastructure. In addition, cloud data centers have high energy needs and require special power and cooling infrastructure, which incurs extra cost [22]. Licenses for software, such as OS and virtualization software, constitute an additional expense. Finally, there is the cost for the real estate, where the data center physically resides [23]. The most important economic cost related to maintenance cost is due to the significant power consumption. It is usually the 15%–20% of the total budget [22,24]. Other costs include the network expenses, which are the costs for communicating with the end users, and the salaries of the data center technicians and the rest of employees.

The cost of running a cloud infrastructure directly impacts on the charges requested by the end users for its usage. However, in our cost model, we do not deal with the issue of configuring the price of the VM options offered by the cloud providers taking into account their actual cost. Rather, we assume that the charges are fixed and provided as input parameters to our model.

3.1.3. *Charging models*

The charging models are orthogonal to the VM characteristics and the costs for developing and running cloud infrastructures. Here, we review the most important charging models, which are all supported by our cost model.

The most common charging model is the *"Pay-as-you-go"* one, where the customer is charged for the actual period she uses the infrastructure. The usage periods are monitored in different granularities though; i.e., providers may have different minimum time unit for charging. For example, one provider's minimum time unit may be 1 h and another's 5 min. So, if someone uses the former infrastructure for 1 h and 23 min, she will be charged as if she used the infrastructure for 2 h, whereas, in the latter case the price will be for 1 h and 25 min. The trend is the time granularity to further decrease and charge per minute or even per seconds of resource usage [1].

The *"Pay-as-you-go"* charging model is encountered in three main forms in Amazon EC2, but those forms are essentially generic to any cloud provider:

- *On-demand Instance*, where the payment is done after the use of the infrastructure charging for as long as the customer used it, without any other commitment, as explained above.
- *Reserved Instance*, where the customer pays a small fee upfront for a specific time (e.g., a month or a year) and after that, she is charged like the on-demand policy for the time using the infrastructure, but with a great discount on the fee.
- *Spot Instance*, which is like an auction. The customer bids whatever price is willing to pay for the infrastructure and, if the bid is above the current spot price, she gets the VM and is charged for the actual usage period but with a lower price than that of the on-demand policy. The drawback is that, if the spot price goes above the customers bid price, her VM will be shut down.

An additional charging model is the *"Committed VM"*. In this model, the client rents the infrastructure for a predefined time. This predefined time can be from one month to a year, or even more in some cases. During this time, the customer can use the infrastructure whenever she wants without any extra cost (except maybe network traffic). Usually, it is less expensive than *"Pay-as-you-go"* when the usage is high. An example of this model is GoGrid.[7]

3.2. *How can the cost model be used?*

Our setting is depicted in Fig. 1. We assume that there exists a centralized optimizer that builds an execution plan in the form of a query tree, that is, vertices correspond to operators and data flows from bottom to top; the root node produces the final query results. At this stage, there is no locality information about which VM each operator is executed on. After that, the execution plan is decomposed into smaller sub-queries, where each subquery corresponds to an execution stage. Those stages will be referred to as *strides*. Before a stride begins its execution, all the lower strides must have been completed. In the figure, we provide an example of a query plan decomposed in four strides.

Assume now that for each stride, each cloud provider is capable of providing a bid as shown in the bottom right of the figure. Each cloud provider can offer multiple combinations

[7] http://www.gogrid.com/products/cloud-servers##pricing.

of type and number of VMs at a different cost, and each such combination may result in different expected execution time. In the generic case, the complete offer per provider per stride is described by a continuous function. In addition, we assume that each user specifies her own function that represents the worst acceptable trade-off. We further assume that the total monetary cost of the query plan is the sum of the cost of each stride; similarly, the total time delay is the sum of the delays in each stride. The aim of a multi-cloud optimizer is to derive an optimal assignment of strides to VMs. This is equivalent of determining exactly one bid point from the set of all bids for each stride, so that the total monetary cost maximizes the difference from the user-supplied function. We are interested in this difference, which is termed as *user satisfaction*, because it captures the savings from the worst acceptable payment.

The problem above involves the computation of the pareto frontier and is NP-hard [26]. A simpler version of this problem has been investigated in the context of the Mariposa distributed query processing system [2,26], where the bid of each provider for each stride is a single point rather than a continuous function. The proposal in [25] generalizes the initial solutions allowing arbitrary non-increasing cloud provider functions and guarantees optimal solutions with bounded relative error in pseudo-polynomial time.

The main problem with the above multi-cloud optimizers is that to date, there is no mechanism to provide the cloud provider bids, which serve as their input. This work aims to complement the proposals in [2,26,25] and provide such a mechanism. So, apart from the fact that a bi-objective cost model is significant in its own right, our proposal serves a secondary purpose, namely to assist in rendering the existing approaches to multi-cloud optimization applicable in practice. A more complete example is provided in Section 4.3.

4. Our cost model

The model we are presenting is used for estimating the time and the economic cost of a query plan executed on cloud-based VMs. To achieve this, we have built on top of the single-objective cost models described in [5,14], although we can encapsulate additional single-objective models. Also, our model can be applied to more generic data flows that are still expressed as DAGs. For simplicity, we start assuming that our process is a traditional query plan, and at the end of this section, we generalize.

4.1. *Assumptions*

Before we describe the cost model's rationale and functions, we need to state the assumptions we make:

- The shape of the query plan and the operator ordering have already been chosen by a centralized optimizer.
- There exists a mechanism that decomposes the query plan into strides in place. The strides can be produced either manually, or in an automated manner, e.g., using the same partitioning approach as in [27]. Each stride comprises vertices that can refer to sub-queries in the original query plan that is groups of simple database operations, or stand-alone operations. Our model does not distinguish between these two cases and treats each stride vertex as a non splittable operator.

Initially, we examine a specific case where all operators within the same stride can be executed in parallel. Also, we assume that every operator can be executed only on one processing node. The number of VMs that will be used in every stride can be up to the number of the query plan vertices that the stride comprises. As such, there is no intra-operator parallelism, where an operator runs on several processors.[8] Later, we show how those assumptions can be relaxed.

4.2. *The cost model*

Our cost model is modular and consists of components that model the charging policies, the computational and the communication execution time, respectively. Based on those components, the economic price is derived as explained below.

4.2.1. *Modeling the charging policies and fees*

In the first part, we model the charging policies described in Section 3 and we map them to specific VM offers. The notation is as follows:

- P_t^a: Denotes the charging policies of the providers, where:
  - a: identifies the type of charging policy; the following list is extensible:
    - $a = 1$: corresponds to *On demand Instance*;
    - $a = 2$: corresponds to *Reserved Instance*;
    - $a = 3$: corresponds to *Committed VM*;
    - $a = 4$: corresponds to *Spot Instance*;
  - t: denotes the charging time unit of charging in minutes; for example, if providers may charge either per minute of usage or 5 min or per hour, then $t \in \{1, 5, 60\}$. In policies where the actual usage is not monitored, as may happen in the *Committed VM* one, t is set to its minimum value, i.e, typically 1 min, to allow for more detailed model estimates.
- VM_k: denotes the specific VM instance, where $1 \leq k \leq |Available\ VMs|$.
- $F_{pr}(P_t^a, VM_k) = (f_a, f_u)$: denotes the price offer from cloud provider pr for the VM instance VM_k according to the pricing policy P_t^a. It consists of two parts: f_a corresponds to the amortized price per time unit for the policies that involve an upfront fee payment (e.g., *Reserved Instance*), whereas it is normally 0 for the *On demand Instance* policy; f_u corresponds to the fee related to the actual VM usage, which is set to 0 for the *Committed VM* policy.

4.2.2. *Estimation of execution time*

To estimate the time that a query takes to be executed on a specific VM, we use the following formula:

$$TotalTime = \sum_{s=1}^{n} \max(S_{s,1\to i}^{VM_{k\to k'}}, \dots, S_{s,m^s\to j}^{VM_{k\to k'}}) \tag{1}$$

[8] We use the terms node, processor and VM interchangeably.

where:

- n = the number of strides.
- m^s = the number of operators in the sth stride.
- $S^{VM_{k \to k'}}_{s,i \to j} = O^{VM_k}_{s,i} + T^{VM_{k \to k'}}_{s,i \to j}$, where:
  - $O^{VM_k}_{s,i}$ = time to execute ith operator of s stride on VM_k.
  - $T^{VM_{k \to k'}}_{s,i \to j}$ = time to transfer the data produced by operator i of the sth stride, which runs on VM_k, to node j, which runs on $VM_{k'}$. Typically, j belongs to the $(s+1)$th stride, but this is not necessary to allow for arbitrary complex execution tree plans. The following conditions hold:
    - $1 \leq s \leq n$,
    - $1 \leq i \leq m^s$,
    - $1 \leq j \leq m^{s+1}$,
    - $1 \leq k \leq |Available\ VMs|$.

The rationale behind the *TotalTime* formula is that: (i) all strides are executed sequentially in a bottom-up fashion, and (ii) all operators belonging to the same stride are executed in parallel. In general, we expect these conditions to hold. Moreover, by taking the maximum in Eq. (1), we silently assume that the execution of one operator does not interfere with the execution of the other operators in the same stride. Unfortunately, this is not always the case. One notable exception is when the network forms a kind of bottleneck in a way that, when several upstream operators send data to a specific operator in another stride, the total amount of elapsed time is the sum of the individual sending times. In general, we can alternatively use the following formula:

$$TotalTime = \sum_{s=1}^{n} \sum_{i=1}^{m^s} S^{VM_{k \to k'}}_{s,i \to j}. \qquad (2)$$

Eqs. (1) and (2) can be further elaborated in a straightforward manner, if only a subset of operators interfere with each other. In that case, the execution time of those operators is summed and then we treat them as a single meta-operator, the execution of which fully overlaps in the time domain with the remainder of operators on the same stride. Then we can apply Eq. (1).

The rationale of $S^{VM_{k \to k'}}_{s,i \to j}$, defined as the sum of the local computation cost and the data transmission cost of each operator, is that each operator first completes its local execution and then starts transmitting data to its consumer. If pipelining is supported, then data transmission starts as soon as the first results are ready. When the data to be processed is large, as expected in cloud settings, the local computation and data transmission operations overlap almost fully, and the time cost of an operator becomes $S^{VM_{k \to k'}}_{s,i \to j} = max(O^{VM_k}_{s,i}, T^{VM_{k \to k'}}_{s,i \to j})$. Overall, any combination of *TotalTime* and $S^{VM_{k \to k'}}_{s,i \to j}$ types of estimation is valid, and it rests with the query optimizer to choose the most appropriate one in each setting.

To calculate $O^{VM_k}_{s,i}$, we can employ the technique described in [5] although our approach is orthogonal to the way $O^{VM_k}_{s,i}$ is estimated. According to [5], the equation for calculating the cost of an operator given a specific VM_k is:

$$\begin{aligned} O^{VM_k}_{s,i} &= \mathbf{n}^{\mathbf{T}} \mathbf{c}_{\mathbf{VM_k}} \\ &= n_s c_s^{VM_k} + n_r c_r^{VM_k} + n_t c_t^{VM_k} + n_i c_i^{VM_k} + n_o c_o^{VM_k} \qquad (3) \end{aligned}$$

$\mathbf{c}_{\mathbf{VM_k}}$ is a vector of statistical metadata for the instance VM_k. The values of $\mathbf{c}$ depend only on the underlying hardware and can be found through a simple calibration procedure. This also means that the calibration of $\mathbf{c}$ has to be done only once for every VM that we want to test, since it is independent of specific queries.

$\mathbf{n}$ is a vector of statistics of the data processed by the operator in the form of cardinalities. For estimating the cardinalities $\mathbf{n}$, Wu et al. in [5] propose a sampling-based approach. These values depend only on the query plan and not on the hardware that will be used to execute the query. Since the query plan is only one and is known, it is possible to calculate the cardinalities with this method without incurring big overhead.

To calculate $T^{VM_{k \to k'}}_{s,i \to j}$ we use, as our basis, the model described in [14], along with the results of cardinality estimation used for estimating the execution time of the operator on the ith node. With the cardinality estimation results, we can calculate the size of produced data X, that will be transferred over the network. We can employ different cost estimation formulas depending on the physical position of the nodes. The nodes i and j of the stride s and $s+1$ respectively, can belong either to the same cluster or to a different one. In the first case we have *intra-cluster* communication, while in the second we have *inter-cluster* communication. These cases are examined as follows.

- Intra-cluster communication, where we have two subcases:
  - *Same VM instance* ($k = k'$): the jth operator is executed on the same node as the ith operator executed. This means that the produced data are already in the same VM. So we have:
    $T^{VM_{k \to k}}_{s,i \to j} = 0$.
  - *Different VM instance* ($k \neq k'$): the jth operator is executed on a different VM instance than the ith operator, but in the same cluster u. This means that data (X) has to be transferred from node i to node j. In this case:
    $T^{VM_{k \to k'}}_{s,i \to j} = C^u_{k \to k'}(X) = \alpha_{k,k'} X + \beta_k$
    where $\alpha_{k,k'}$ is the communication cost to transfer one unit of data from node k to node k' and β_k is the communication startup cost.
- Inter-cluster communication. In this case, operator i is executed on a node in cluster u, while operator j is executed on a node in cluster v. So the transfer time is estimated by the following formula:

$$T^{VM_{k \to k'}}_{s,i \to j} = C^u_{k \to g_u}(X) + C_{g_u,g_v}(X) + C^v_{g_v \to k'}(X) \qquad (4)$$

where:

- $C_{g_u,g_v}(X)$ is the communication cost for X amount of data between cluster u and v through their gateways g_u and g_v respectively.
- $C^u_{i \to g_u}(X)$ is the transmission cost between node k and the gateway node (g_u) in cluster u.
- $C^v_{g_v \to j}(X)$ is the transmission cost between gateway node (g_v) and node k' in cluster v.

Typically, the second of the components above dominates, so Eq. (4) is approximated as:

$$T^{VM_{k \to k'}}_{s,i \to j} \simeq C_{g_u,g_v}(X).$$

4.2.3. *Monetary cost estimation*

The third part of our model is the estimation of the cost of a query in monetary units. To estimate this, we need to combine the pricing offers of the different providers with the time estimation of our model. The total price depends on the execution time of each operator and the associated fee:

$$\sum_{s=1}^{n}\sum_{i=1}^{m^s} Price(S_{s,i\to j}^{VM_{k\to k'}}, F_{pr}(P_t^a, VM_k), F_{pr}(P_{t'}^a, VM_{k'})), \qquad (5)$$

where *Price*, computes the fee for using a VM_k for S time based on the P_t^a charging policy and transferring data to $VM_{k'}$.

Assume that $F_{pr}(P_t^a, VM_k) = (f_a, f_u)$ and $F_{pr}(P_{t'}^a, VM_{k'}) = (f'_a, f'_u)$. Then *Price* is estimated as follows:

$$Price = (f_a + f_u)\left\lceil \frac{S_{s,i\to j}^{VM_{k\to k'}}}{t} \right\rceil + (f'_a + f'_u)\left\lceil \frac{T_{s,i\to j}^{VM_{k\to k'}}}{t'} \right\rceil$$

capturing the fact that data transfer results in concurrent usage of both the sender and the receiver VMs.

It is worth noting that there are several alternatives when defining f_a, depending on the expected usage of the model. For example, if the user is not interested in the pre-paid amount, then f_a can be set to zero for any charging policy. Also, if the cost estimation is used to decide whether to request further VMs in addition to those already reserved, then f_a for the *on-demand* instances may include the amortized cost of the reserved instances instead of being 0. The formulas presented above are generic enough to support such scenarios. Finally, it is straightforward to extend them to support charges based on the volume of the data transferred across the network.

4.3. *An example*

To give a better view of our model, we will present a simple example. Suppose we have query q that is executed in two strides executed sequentially so that we can employ Eq. (1) for time cost estimates. The query consists of two operators, one in each stride. For simplicity we will assume that:

- The time to transfer the intermediate data from *Stride 2* to *Stride 1* is 1 h (60 min) if the VMs of each stride are different, or 0 if it is the same VM. The data transfer takes place after the completion of local execution.
- There is no charge for data transfer between the nodes that execute the query. Usually this only happens when the nodes are of the same provider.
- All the data that the query needs, including the initial data and the intermediate data, fit completely in the storage space provided by the specific instance. This implies that we do not have any extra cost for storage space.

We have two IaaS providers, A and B, with their provided VMs presented in Table 1. The charging policies are in Table 2. The pricing offers based on those charging policies can be seen in Table 3 along with their respective f_a and f_u values. Finally, the estimated time to execute each operator on the provided VMs ($O_{s,i}^{VM_k}$) can be seen in Table 4.

Table 1 – Example of VM instances taken from AWS.

VM	Providers	ECU	Memory	Storage
VM_1	A, B	3	3.75 GiB	1 × 4 SSD
VM_2	A	6.5	7.5 GiB	1 × 32 SSD
VM_3	B	6.5	15 GiB	1 × 32 SSD
VM_4	B	13	15 GiB	2 × 40 SSD
VM_5	A	14	7.5 GiB	2 × 40 SSD

Table 2 – Pricing policies.

Pricing policy	**Charging time unit**
P_{60}^1	Hour
P_5^1	5 min
P_{60}^2	Hour
P_1^3	Month
P_1^3	3 Month

Table 3 – Pricing offers in the example.

Pricing policy	f_a	f_u	Description
On-demand			
$F_A(P_{60}^1, VM_1)$	0	5	5$/h
$F_A(P_{60}^1, VM_2)$	0	17	17$/h
$F_B(P_5^1, VM_3)$	0	1.70	1.70$/5 min
$F_B(P_5^1, VM_4)$	0	4.80	4.80$/5 min
$F_A(P_{60}^1, VM_5)$	0	60	60$/h
Reserved Instance			
$F_A(P_{60}^2, VM_2)$	0.685	15	15$/h +500$/Month
$F_A(P_5^2, VM_3)$	0.057	1.5	1.50$/5 min+500$/Month
$F_A(P_5^2, VM_4)$	0.057	4.3	4.30$/5 min+500$/Month
$F_A(P_{60}^2, VM_5)$	0.685	53	53$/h +500$/Month
Committed VM			
$F_A(P_1^3, VM_1)$	0.1667	0	7300$/Month
$F_B(P_1^3, VM_1)$	0.1826	0	8000$/Month

Table 4 – Example of estimated execution times.

	VM_1 (min)	VM_2 (min)	VM_3 (min)	VM_4 (min)	VM_5 (min)
$O_{2,1}^{VM_k}$	2000	1200	1150	650	610
$O_{1,1}^{VM_k}$	500	310	300	150	130

In Fig. 2, we present a diagram of the estimated execution times along with the estimated monetary costs for the query q for every combination of VMs with *On demand* and *Reserved* charging policies in Table 3. Black bullets represent the *On-demand* model, while the white ones correspond to the *Reserved Instance* model. In general, the monetary cost is inversely proportional to the execution time and *Reserved Instance* pricing is less expensive than *On demand*. In Fig. 2, the rightmost combination (A) is to allocate both operators to VM_1, which is the less expensive albeit the slowest VM, while combination C corresponds to the mapping of the two query strides to $VM_5 \to VM_4$, which are the faster and most expensive VMs.

It can be seen that some VM combinations dominate some others, i.e., they are both more efficient in terms of execution time and less expensive. This is attributed to the different charging policies (e.g., charges for each hour or for each 5 min

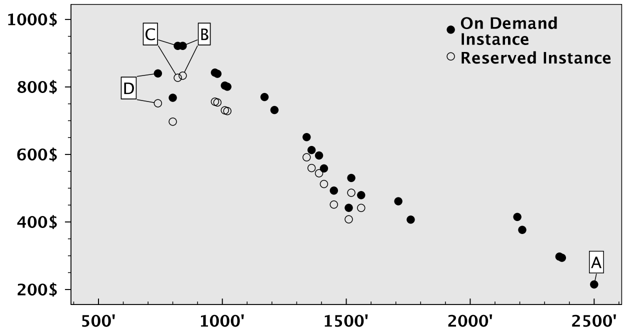

Fig. 2 – Time and monetary cost combinations for the example query.

period of usage). One other factor is the fact that, in some combinations, both operators are executed on the same VM, and as such, there is no data transfer across the network, which leads to reduced execution time and monetary cost. For instance, for combination D on the diagram, VM_5 is used for both operators. Due to the absence of intermediate data transfer, this combination has both lower cost (by 60$) and running time (by 60 min) under the *On demand* policy than combination B for example, where data has to be transferred from VM_4 to VM_5.

In this example, we do not have any extra cost for the transfer of intermediate data between strides. If we had such a cost, we could use the cardinality estimation to determine the size of data transferred. Since we know the data that to be transferred and the charging policies of the IaaS providers, we are able to determine the cost for transferring data between strides. Also, in the example, we have assumed that all the data, including initial and intermediate data, fit completely in the storage space. If this is not the case, we can estimate the data volume that needs to be stored based on the cardinality estimates, and then estimate the monetary cost for using extra storage space.

4.3.1. *Fitting the cost model into an optimizer*

The role of the optimizer is to choose the most appropriate combination of VMs, considering both time and monetary cost. The example discussed above is simple but adequate to provide insights into the complexity of the problem when every VM combination is examined. More specifically, the average time complexity of estimates for each stride is proportional to the number of pricing offers, the number of available VMs and the number of stride vertices. The estimates of the complete query plan are exponential in the number of strides. As such, an optimizer that relies on examining explicitly each combination is not practically feasible. Nevertheless, not every combination needs to be examined, since a big portion of such combinations are dominated by others, i.e., for both criteria there exists at least one combination that is equal or better.

Producing non dominated combinations is the same as producing the pareto curve. An optimizer can build an exact pareto curve with the help of a simple pseudopolynomial dynamic programming algorithm. The sketch of such an algorithm is given in [26], and here we present a more complete description. The algorithm computes for each operator in each stride, the minimum monetary cost C to compute the query plan up to that operator provided that the time does not exceed a delay threshold d. This is repeated for all possible d values, which range from the minimum time granularity t_{min} (typically 1 min) to the maximum possible total time d_{max} in steps of size equal to t_{min}. Detecting an upper bound for d_{max} is trivial, since we can sum the maximum $S_{s,i\to j}^{VM_{k\to k'}}$ values over all operators in the plan.

The algorithm builds a 2-dimensional array $C_s^i(k, d)$ for each operator $i = 1 \ldots m^s$ for each stride $s = 1 \ldots n$, where $1 \le k \le |Available\ VMs|$ and $t_{min} \le d \le d_{max}$. It starts from the bottom strides and proceeds to the top ones. The C arrays are completed column-by-column. A specific cell keeps the minimum economic cost of computing the query plan up to the corresponding operator in time no more than d and having the results of that operator residing at VM_k.

Let $x = O_{s,i}^{VM_k}$. For simplicity of notation, we can assume that each VM comes with a single pricing policy, so that each k can be mapped to a single (f_a, f_u) pair. The recursive function used to drive the dynamic programming approach is as follows:

$$C_s^i(k, d) = \begin{cases} \infty & \text{when } x > d \\ mincost_s^i(k, d) & \text{otherwise} \end{cases}$$

$$mincost_s^i(k, d) = min \begin{cases} (f_a + f_u)\left\lceil \frac{x}{t} \right\rceil + \sum\limits_{v \in Children(i)} C_{s-1}^v(k, d - x) \\ min\left(C_s^i(j, d - T_{s,i\to *}^{VM_{j\to k}}) + (f_a + f_u + f_a^\star + f_u^\star) \right. \\ \quad \left. \times \left\lceil \frac{T_{s,i\to *}^{VM_{j\to k}}}{t} \right\rceil \right) \quad \forall j \neq k. \end{cases}$$

The rationale of *mincost* is to check the following two cases for each combination of VM and acceptable total time: either to perform the computation of the *i*th operator of the *s*th stride directly on VM_k or to compute that operator at another place and transfer the results to VM_k. For the latter, we examine the (already computed) optimal solutions for all VM_j, $j \neq k$ (corresponding to the $(f_a^\star + f_u^\star)$ pair) and we take their minimum.

The global solution is in the array of the root operator (typically C_1^n). Taking the minimum of each column provides the less expensive cost for a given time threshold thus yielding only the pareto curve instead of all combinations. The optimizer can choose the exact allocation either with the help of user-defined functions (as mentioned earlier) or through conversion of the bi-objective optimization problem to a single-objective one with a constraint on the other optimization criterion. For example, it can choose the solution with the lowest total time that has economic cost below a supplied threshold.

4.4. *Generalizations*

The cost model described in this section can be generalized in two main ways not covered previously: to support arbitrary data analysis flows and intra-operator parallelism.

Data analysis flows are typically represented as directed acyclic graphs (DAGs), which can be naturally split in multiple stages, exactly as the query plans we consider do. The main difference between arbitrary data flows and query plans is that query execution plans consist of operators from the extended relational algebra, whereas data flows also encompass data and text analytics, machine learning operations, and so on [3]. The implication in our cost model is in the way $O_{s,i}^{VM_k}$ is estimated. The approach in [5] cannot apply because it is specific to atomic query operators; so we need to resort to micro-benchmarking solutions, e.g., as described in [18,28]. The rest of the cost model details remain the same.

Regarding intra-operator parallelism, the extensions are straightforward as well. We can assume that, if we fix the degree of intra-operator parallelism, then we can modify the query execution plan, so that each instance of a partitioned operator appears as a separate query plan vertex. Then, we can apply the cost model without any modification.

Finally, a shortcoming of the monetary cost estimation in Section 4.2.3 is that a VM that receives data from a remote host is activated and, if it does not start processing its results immediately, it may not be de-activated. The formula presented does not capture this, but it is straightforward to devise more sophisticated formulas that keep track of the first time a VM is activated until it finishes the execution of all the tasks allocated to it.

5. Validation case study

In this section, we demonstrate how exactly the cost model is used in a real multi-cloud environment. The first part shows how the cost model is calibrated in a single cloud infrastructure and how we derive time estimates. Monetary cost estimates are covered by the example in Section 4.3. We then present more complex settings that employ multiple cloud infrastructures and depart from relational database queries.

5.1. *Simple experimental setting and model calibration*

For our experiments we used *okeanos*. *okeanos* is a IaaS platform for the Greek Academic and Research Community [29]. More specifically, we used 3 VMs with the following hardware configurations:

- 2 VMs (VM_1 and VM_3) : 60 GB Disk, 4 GB Ram, 1-core × 2.1 GHz.
- 1 VM (VM_2): 40 GB Disk, 2 GB Ram, 1-core × 2.1 GHz.

Our software setup includes the installation of PostgreSQL 9.1.11 on Linux Kernel 3.2.0-58-generic. The data we used come from the TPC-H decision support benchmark and the database size is 26 GB.

To use the cost model, we need to parameterize the time estimates for data transfer, writing (resp. reading) raw data to (resp. from) disk and for the relational operators. For the calculation of the network speed, our case is that the VMs belong to the same cluster. So the network formula is given by:

$$T_{s,i\to j}^{VM_{k\to k'}} = C_{k\to k'}^{u}(X) = \alpha_{k,k'}X + \beta_k$$

where $\alpha_{k,k'}$ is the communication cost to transfer one unit of data from node k to node k' and β_k is the startup cost.

Table 5 – Network α and β parameters.

	$\alpha_{k,k'}$ (ms/bytes)			β_k (ms)
	$k' = 1$	$k' = 2$	$k' = 3$	
$k = 1$	–	6.29e−05	5.54e−05	8.61e02
$k = 2$	6.60e-05	–	6.39e−05	9.05e02
$k = 3$	5.67e−05	6.09e−05	–	1.08e03

Table 6 – Disk performance in MB/s.

	VM_1	VM_2	VM_3
Read speed (MB/s)	291.60	255.30	43.88
Write speed (MB/s)	100.83	106.20	96.39

To find α and β, we conducted two experiments with different X for every combination of VMs. We used the command *dd* of unix to produce two files and the command *scp* to transfer these files between servers and measure the network performance. We repeated this experiment 10 times for each X value and then calculated the mean time. Since we have a linear function, we can calculate α and β with simple maths. The results are presented in Table 5.

Next, to estimate the query time correctly, we need to know the read and write speed of the disk of every VM. We measure the sequential speed using the *dd* unix tool while we set the block size to 4096 bytes. Again, we repeated the measurements 10 times and we calculated the mean values. The results are presented in Table 6.

Finally, to estimate the execution time $O_{s,i}^{VM_k}$ of relational operators executed by PostgreSQL, we need both cost units $\mathbf{c}$ and cardinalities $\mathbf{n}$. For cardinalities $\mathbf{n}$, we assume that they can be calculated with the method described in [5] and thus are accurately known. To calculate $\mathbf{c}$, we have used the calibration queries from [5]. We have calculated only the cost units that are involved in the experiments in the next part; the results are in Table 7. The queries are:

- `SELECT * FROM R`
  R is memory resident. This query is used to find *cpu_tuple_cost*.
- `SELECT COUNT(*) FROM R`
  R is memory resident. This query along with the previous one is used to find *cpu_operator_cost*.
- `SELECT * FROM R`
  R does not fit in memory. With the help of the first query, this query is used to find *seq_page_cost*.

where *seq_page_cost* gives the time cost to sequentially perform an I/O operation accessing a disk page. *cpu_tuple_cost* is the time to retrieve and process a tuple. *cpu_operator_cost* captures the extra cost of applying a hash or an aggregate function to a tuple (that cost is not covered by *cpu_tuple_cost*).

5.2. *Running time estimates*

In our experiments, we tried to validate whether our cost model can predict with adequate precision the execution time

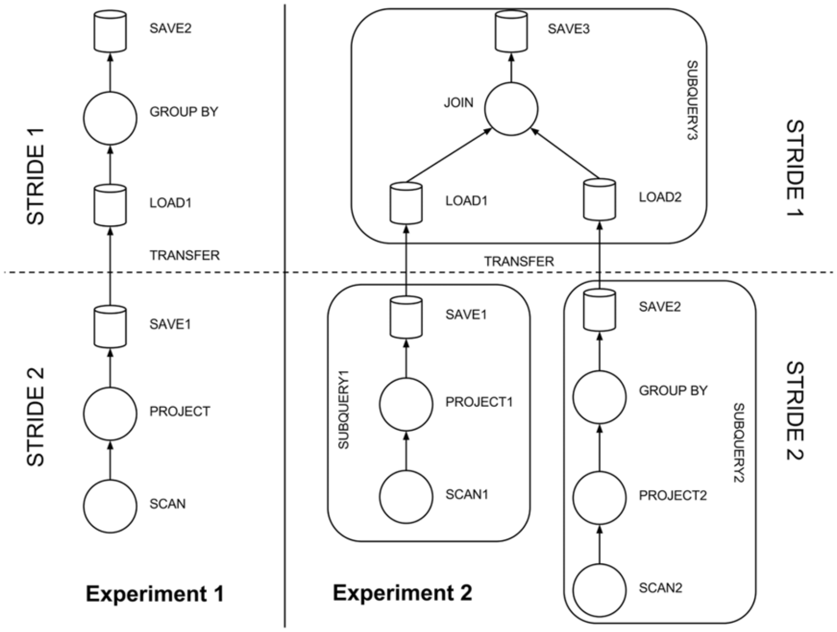

Fig. 3 – Plans of the experiments.

Table 7 – Cost units of our VMs.

Optimizer parameter (ms)	VM_1	VM_2
seq_page_cost	2.16e−02	3.21e−02
cpu_tuple_cost	3.76e−04	3.48e−04
cpu_operator_cost	2.17e−04	2.48e−04

Table 8 – Results of actual execution time and estimated time.

	Estimated values (ms)	Actual values (ms)
Exp. 1	15531	16172
Exp. 2	3131	3285

of a query. We used the PostgreSQL database only for the operators of the bottom strides. All the other operators are implemented with unix scripts. The cost units **c** were used only to predict the running time of the PostgreSQL operators. To calculate the time of operators implemented with scripts, we treated them as black boxes, we executed them with different inputs and measured their performance.

Experiment 1
The query of the first experiment is:

```
SELECT c_nationkey, count(*)
FROM customer
GROUP BY c_nationkey
```

The corresponding plan is presented in Fig. 3 on the left.

For the execution of the above query, we use two VMs (VM_1 and VM_2). VM_1 is used for stride 1, while VM_2 runs stride 2. The execution is split in three steps as follows:

1. SCAN, PROJECT and SAVE1 are executed on VM_1 as a single SQL sub-query submitted to the PostgreSQL database.
2. TRANSFER is performed using the unix *scp* command.
3. LOAD1, GROUP BY and SAVE2 is implemented as a single unix script running on VM_2.

We sum the execution time of each of the above steps to calculate the total actual execution time of the query. Our experiments were repeated 10 times, and their mean value is in the *Actual Values* column of Table 8.

To find the estimated execution time, we used the actual cardinality. For the calibration of the execution time of the *GroupBy* operator, we first applied this operator on a set of 1 million random numbers for 10 different times. Again, the total time is given by the sum of the times of the three steps and the results are in Table 8 (see the *Estimated Times* column). From the table, we can observe that the deviation between the actual and the estimated times in that experiment is less than 4%.

Experiment 2
The query of the second experiment is:

```
SELECT nation.n_name, count(*)
FROM customer, nation
WHERE customer.c_nationkey = nation.n_nationkey
GROUP BY nation.n_nationkey
```

The detailed execution plan can be seen in Fig. 3 on the right. We can observe that the database optimizer has performed an optimization, which pushes the group-by under the join yielding significantly lower execution times.

For the execution of this query, we use three VMs. VM_1 and VM_2, which have both PostgreSQL database installed, are used for the bottom stride, while VM_3 executes the top stride. The execution consists of three sequential steps:

1. SubQuery1 and SubQuery2 are executed in parallel on VM_1 and VM_2, respectively.
2. The intermediate data is transferred with the help of *scp* as previously.
3. SubQuery3 is implemented as unix script (using *join* command) that runs on VM_3.

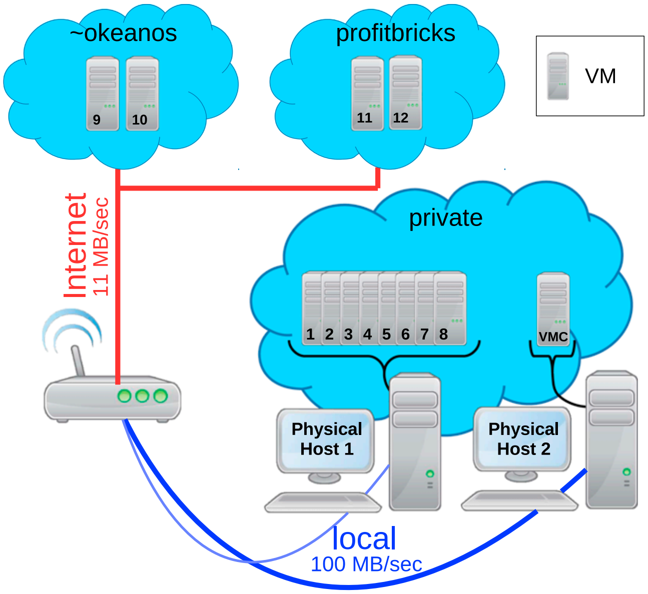

Fig. 4 – Our experimental topology.

The total time is estimated by adding the maximum estimated time of subqueries 1 and 2 in Stride 2 along with the data transfer from the first stride to the second one. Then we add the estimated time of subquery3. The average results of the 10 runs are in Table 8. Again, the deviation of the estimates from the actual running times is low (4.69%). It is important to clarify that the VMs used were not installed on dedicated machines. On the contrary, the cloud infrastructure in our experiments is heavily used by a big community that shares the physical resources provided.

5.3. *Validation using multi-cloud NoSQL databases*

The previous experiments showed the accuracy of the model in a single cloud setting. We now move to a multi-cloud environment and we focus on queries that access HBase, a widespread NoSQL system. We employ three HBase clusters. The version of HBase in each cluster is 0.94.20 on top of Hadoop 1.2.1. To produce the data, we have modified the YCSB's v0.1.3 table loading module to create records/rows of 100 kB each (10 fields of 10 Kb each in every row). The rows are distributed according to pre-specified (i.e., pre-split) regions. All the region data fit into the main memory so that there are no disk accesses due to caching.

The three clusters consist of 12 VM in total and run on three distinct cloud infrastructures. In each cluster, 1 VM plays the role of the master HBase server, and the rest are region server VMs. Fig. 4 depicts the actual set up, which comprises:

- Our own *private* cloud infrastructure, which is supported by the ganeti v2.11 cluster virtual server management software. It physically resides at the premises of the Aristotle University of Thessaloniki, Greece. Our infrastructure consists of 2 physical host machines (host 1, host 2) where we create the first 8 VMs. The host machines, and thus every created VM, are inter-connected through a local network with 1 Gbps (100 MB/s) network speed. The connection speed to the internet is 100 Mbps (11 MB/s). Every VM has 2 VCPUs (at 2.0 GHz), 7 GB RAM (5.8 GB dedicated to HBase) and 100 GB of storage. The database created consists of 70K rows and the total database size is 7 GB, that is 1 GB per region server.
- The *okeanos* cloud infrastructure, which was introduced earlier and on which we have spawned 2 VMs to create a 2-node HBase cluster (1 region server VM_9, 1 master server VM_{10}). The VM characteristics are 2 VCPUs (at 2.1 GHz), 6 GB RAM (4.8 GB dedicated to HBase) with 100 GB and 40 GB of storage for the two VMs, respectively. We have created a database with 7K rows and total size of 700 MB. The database is pre-split into two equi-sized regions. The VMs physically reside in Athens, Greece.
- The *profitbricks*[9] cloud infrastructure, where we have created a 2-node HBase cluster hosting a database of 700 MB of size with 7K rows placed in one region. The corresponding VMs are VM_{11} and VM_{12}. The VM characteristics are 2 VCPUs (at 2.8 GHz), 5 GB RAM (3 GB dedicated to HBase) and 50 GB of storage.

In the experiments we use a client machine that either collects or joins the data from multiple VMs. The client VM is running in our private cloud infrastructure but on a different host machine, in order not to interfere with the HBase cluster's resources. The characteristics of the client VM are: 8 VCPUs (4.7 GHz), 20 GB RAM, and 500 GB of storage.

5.3.1. *Model calibration*

To use our model, we need to find the parameters for data transfer and local reads from the HBase servers. To avoid interference to the data transfer speed due to disk accesses, we employed the *iperf* tool,[10] which can measure

[9] https://www.profitbricks.com/.

[10] https://iperf.fr/.

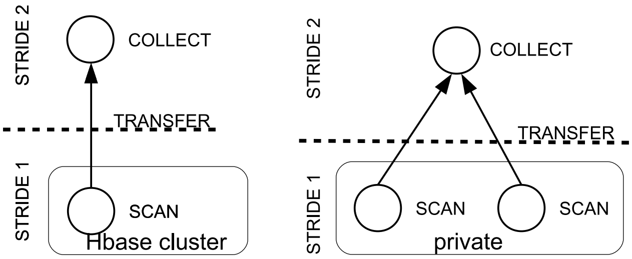

Fig. 5 – SCAN to a single HBase cluster at a time.

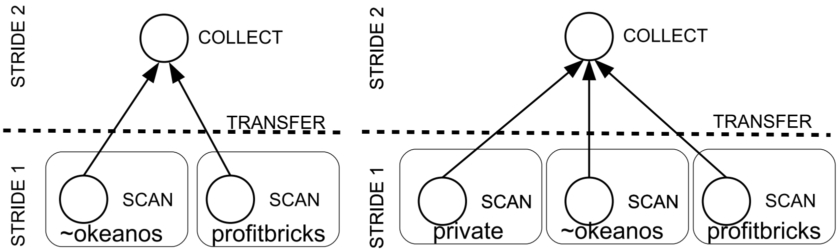

Fig. 6 – SCAN to region servers from separate HBase clusters.

Table 9 – Network α and β parameters.

HBase cluster	α (s/bytes)	β (s)
private	8.507e−09	0.02989
okeanos	8.603e−08	−0.03273
profitbrick	8.62e−08	0.3137

Table 10 – SCAN local call execution times (in seconds) as a function of the data region size requested.

	1 GB (s)	700 MB (s)	350 MB (s)
VM_1–VM_7	2.9995	–	–
VM_9	–	5.2305	2.9995
VM_{11}	–	4.8164	1.6004

the maximum TCP bandwidth. We have used the same formula as in Section 5.1 (corresponding to the approximate version of Eq. (4)). To compute the α and β values, we have collected measurements for different time intervals (i.e. 1, 2, 5, 10, 20 s) with 10 iterations for each setting and then applied linear regression. The calls were made by the client to the master of every HBase cluster (i.e., VM_8, VM_{10}, and VM_{12}, respectively). The results are presented in Table 9.

To compute the data retrieval operator's cost we have applied local calls (termed as SCANs) to the region servers, which hold the regions with the requested data. The results are presented in Table 10.

5.3.2. *Running time estimates*

In the first set of NoSQL experiments we apply scan queries, where the client retrieves data from the HBase clusters. The details of the experiments are as follows:

1. *SCAN-Experiment-1 (SE1):* We have requested data from one region server from each of the cloud databases, as shown in Fig. 5(left). The region sizes are 1 GB for the private cloud infrastructure, 700 MB for okeanos and 350 MB for profitbreaks.
2. *SCAN-Experiment-2 (SE2):* We have requested 2 GB of data from the HBase cluster running on our private cloud infrastructure, which corresponds to reading data from two region servers, as shown in Fig. 5(right).
3. *SCAN-Experiment-3 (SE3):* We have requested 700 MB of data (i.e., data from one region server) from the HBase cluster running on the okeanos cloud infrastructure and 350 MB of data from the HBase cluster running on the profitbricks cloud infrastructure; see Fig. 6(left).
4. *SCAN-Experiment-4 (SE4):* We have requested data from one region server from each of the HBase databases concurrently; see Fig. 6(right).

Table 11 – Results of actual execution time and estimated time for distributed scans.

Exp. ID	Estimated values (s)	Actual values (s)	Deviation of estimates
SE1-private	12.16	10.79	+12.97%
SE1-okeanos	68.34	65.28	+4.69%
SE1-profitbricks	33.55	45.12	−25.88%
SE2	21.29	20.95	+1.62%
SE3	101.89	99.19	+2.72%
SE4	101.89	101.78	+0.1%

In all experiments, the collection process on the client (Stride 2) takes negligible time. The estimated and the actual times are shown in Table 11, where it is shown that the accuracy is reasonably high apart from simply reading the data from *profitbricks*. Interestingly, the accuracy is much higher for the more complex settings of *SE2*, *SE3* and *SE4* rather than for *SE1*. As previously, we report the averages of 10 runs. For *SE1* and *SE2*, the estimated values were produced through simple application of Eq. (1) and summing the data transfer and local processing costs for each operator. The relatively high

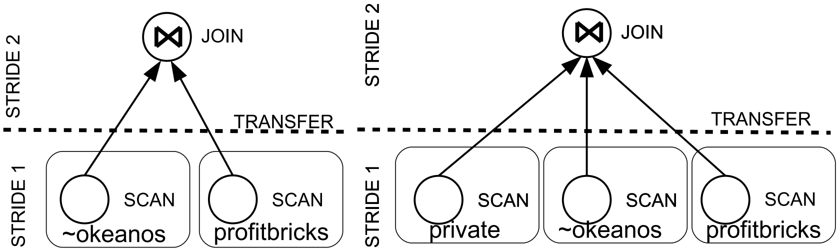

Fig. 7 – JOIN from region servers in separate HBase clusters.

inaccuracy of *SE1-profitbricks* is mainly attributed to the high geographical distance between the participating physical machines. For *SE2*, local experiments show that the scans in the bottom stride in Fig. 5(right) do not fully overlap so that is absolutely correct to take their maximum, but despite this fact, the inaccuracy of the estimate is only 1.62%. The estimates for the next experiments make use of Eq. (2), since the internet connection forms the main bottleneck. We can observe that we give the same estimate for *SE3* and *SE4*. This is because the client machine resides on the same local network as our private cloud infrastructure, therefore the bottleneck is only at the connection with the other two clouds. Therefore the estimate of the time cost of the bottom stride in Fig. 6(right) is the maximum between (i) the scan at the private cloud and (ii) the sum of the two other scans. According to the profiling metadata, the sum of the two other scans dominates, and the estimate is the same as if only these two cloud databases had been contacted.

In the second set of experiments we focus on more complex tasks that join data from multiple cloud providers. We conduct two further experiments, as shown in Fig. 7. These experiments replace the simple data collection task in *SE3* and *SE4* with a JOIN one.

To compute the join operator's cost we have previously applied local join calls on the client machine for 2-way and 3-way main memory nested loop joins [11]. The details of the experiments are as follows:

1. *JOIN-Experiment-1 (JE1):* We have requested 700 MB of data (7000 rows) (i.e., data from one region server) from the HBase cluster running on the okeanos cloud infrastructure and 350 MB of data (3500 rows) from the HBase cluster running on the *profitbricks* cloud infrastructure; see Fig. 7(left).
2. *JOIN-Experiment-2 (JE2):* We have requested 100 MB of data (1000 rows) from the private cloud infrastructure, 700 MB of data (7000 rows) (i.e., data from one region server) from the HBase cluster running on the okeanos cloud infrastructure and 350 MB of data (3500 rows) from the HBase cluster running on the profitbricks cloud infrastructure; see Fig. 7(right). Before the join, we filter 25% of the rows.

The results are presented in Table 12. As for *SE3* and *SE4*, the accuracy is remarkably high, and the average deviation does not exceed 1.57% of the actual value.

Table 12 – Results of actual execution time and estimated time for distributed joins.

Exp. ID	Estimated values (s)	Actual values (s)	Deviation of estimates
JE1	105.69	104.06	+1.57%
JE2	153.4	153.7	−0.2%

6. Conclusions

In this work, we present a bi-objective cost model that provides time and monetary cost estimates of query plans running on VMs from multiple cloud providers. Our cost model extends existing approaches that solely focus on time estimates when tasks run on predefined machines and is tailored to a multi-cloud environment, where resources are used at a price. More specifically, we leverage existing proposals for providing time estimates, and we provide a model for coupling time and monetary cost estimates while supporting the main charging policies of cloud providers to date. The cost model is also applicable to generic data flow tasks, and through a detailed example and validation case studies, we show how it can be employed in practice. In addition, we explain how the model can become part of and render applicable a state-of-the-art multi-objective query optimizer, which requires as inputs the pairs of time and money cost estimated produced by our model.

Our proposal is the first bi-objective cost model for cloud queries. However, it suffers from several limitations that can be addressed in the future, since providing time and cost estimates is a complex issue. Two of the most important directions are to devise cost models that map tasks to the amortized cost of using the infrastructure (rather than the price charged) and to perform more thorough validation after having developed and established suitable benchmarks. Finally, given that several other objectives are relevant to cloud databases including security, reliability and QoS, it would be interesting to extend our model in order to cover them.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund—ESF) and Greek National funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

REFERENCES

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, D. Tsafrir, The rise of raas: the resource-as-a-service cloud, Commun. ACM 57 (7) (2014) 76–84.

[2] M. Stonebraker, P.M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu, Mariposa: A wide-area distributed database system, VLDB J. 5 (1) (1996) 48–63.

[3] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, Optimizing analytic data flows for multiple execution engines, in: Proc. SIGMOD Conference, 2012, pp. 829–840.

[4] W. Zeng, C. Mu, M. Koutny, P. Watson, A Flow Sensitive Security Model for Cloud Computing Systems, Tech. Rep., 2014.

[5] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigumus, J.F. Naughton, Predicting query execution time: Are optimizer cost models really unusable? in: Proc. ICDE Conference, 2013, pp. 1081–1092.

[6] M.T. Özsu, P. Valduriez, Principles of Distributed Database Systems, third ed., Springer, 2011.

[7] D. Taniar, C.H.C. Leung, J.W. Rahayu, S. Goel, High Performance Parallel Database Processing and Grid Databases, John Wiley & Sons, 2008.

[8] D.J. DeWitt, J. Gray, Parallel database systems: The future of high performance database systems, Commun. ACM 35 (6) (1992) 85–98.

[9] K.A. Kumar, A. Quamar, A. Deshpande, S. Khuller, SWORD: workload-aware data placement and replica selection for cloud data management systems, VLDB J. 23 (6) (2014) 845–870.

[10] Z. Karampaglis, A. Gounaris, Y. Manolopoulos, A biobjective cost model for database queries in a multi-cloud environment (best paper award), in: Proc. MEDES Conference, 2014, pp. 109–116.

[11] H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems — the Complete Book, second ed., Pearson Education, 2009.

[12] V. Zadorozhny, L. Raschid, T. Zhan, L. Bright, Validating an access cost model for wide area applications, in: Proc. CoopIS Conference, 2001, pp. 371–385.

[13] A. Rahal, Q. Zhu, P.-Å. Larson, Evolutionary techniques for updating query cost models in a dynamic multidatabase environment, VLDB J. 13 (2) (2004) 162–176.

[14] Y. Slimani, F. Najjar, N. Mami, An adaptive cost model for distributed query optimization on the grid, in: Proc. OTM Conference, 2004, pp. 79–87.

[15] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, S.B. Zdonik, Learning-based query performance modeling and prediction, in: Proc. ICDE Conference, 2012, pp. 390–401.

[16] J. Duggan, U. Çetintemel, O. Papaemmanouil, E. Upfal, Performance prediction for concurrent database workloads, in: Proc. SIGMOD Conference, 2011, pp. 337–348.

[17] J. Li, J.F. Naughton, R.V. Nehme, Resource bricolage for parallel database systems, PVLDB 8 (1) (2014) 25–36.

[18] B. Huang, S. Babu, J. Yang, Cumulon: optimizing statistical data analysis in the cloud, in: Proc. SIGMOD Conference, 2013, pp. 1–12.

[19] S. Chaudhuri, V.R. Narasayya, R. Ramamurthy, Estimating progress of long running SQL queries, in: Proc. SIGMOD Conference, 2004, pp. 803–814.

[20] J. Li, R.V. Nehme, J.F. Naughton, GSLPI: A cost-based query progress indicator, in: Proc. ICDE Conference, 2012, pp. 678–689.

[21] D. Perez-Palacin, R. Calinescu, J. Merseguer, log2cloud: log-based prediction of cost-performance trade-offs for cloud deployments, in: Proc. SAC Conference, 2013, pp. 397–404.

[22] A. Greenberg, J. Hamilton, D.A. Maltz, P. Patel, The cost of a cloud: Research problems in data center networks, ACM SIGCOMM Comput. Commun. Rev. 39 (1) (2008) 68–73.

[23] X. Li, Y. Li, T. Liu, J. Qiu, F. Wang, The method and tool of cost analysis for cloud computing, in: Proc. CLOUD Conference, 2009, pp. 93–100.

[24] J. Hamilton, Cooperative Expendable Micro-Slice Servers (CEMS): Low Cost, Low Power Servers for Internet-Scale Services, Tech. Rep., 2009.

[25] E. Tsamoura, A. Gounaris, K. Tsichlas, Multi-objective optimization of data flows in a multi-cloud environment, in: Proc. DanaC Workshop, 2013, pp. 6–10.

[26] C.H. Papadimitriou, M. Yannakakis, Multiobjective query optimization, in: Proc. PODS Conference, 2001, pp. 52–59.

[27] A. Gounaris, R. Sakellariou, N.W. Paton, A.A.A. Fernandes, A novel approach to resource scheduling for parallel query processing on computational grids, Distrib. Parallel Databases 19 (2–3) (2006) 87–106.

[28] P. Shivam, S. Babu, J.S. Chase, Active and accelerated learning of cost models for optimizing scientific applications, in: Proc. VLDB Conference, 2006, pp. 535–546.

[29] V. Koukis, C. Venetsanopoulos, N. Koziris, ~okeanos: Building a cloud, cluster by cluster, IEEE Internet Comput. 17 (3) (2013) 67–71.