Elastic Dataflow Processing on the Cloud

Herald Kllapi*

National and Kapodistrian University of Athens,
Department of Informatics and Telecommunications
herald@di.uoa.gr

Abstract. Clouds have become an attractive platform for large-scale data processing, especially due to the concept of elasticity, which characterizes them: resources can be leased on demand and used for as much time as needed, offering the ability to create virtual infrastructures that change dynamically over time. Modern applications build on clouds require processing of complex queries that are expressed in high-level languages and are typically transformed into data processing flows (dataflows). A logical question that arises is whether elasticity affects dataflow execution and in which way. It seems reasonable that the execution is faster when more resources are used, however the monetary cost is higher. This gives rise to the concept eco-elasticity, an additional kind of elasticity that captures the trade-offs between the execution time and the amount of money we pay for it as influenced by the use of different amounts of resources. In this thesis, we approach the elasticity of clouds in a unified way that combines both the traditional notion and eco-elasticity. This unified elasticity concept is essential for the development of auto-tuned systems in cloud environments. First, we demonstrate that eco-elasticity exists in several common tasks that appear in practice. Next, we present two cases of auto-tuned algorithms that use the unified model in order to adapt to the query workload: *1)* processing analytical queries in the form of tree execution plans to maximize profit and *2)* automated index management taking into account compute and storage resources. Finally, we describe Exareme, a system for elastic data processing on the cloud that we used and extended. Exareme exploits both elasticities of clouds by dynamically allocating and deallocating compute resources in order to adapt to the query workload.

1 Introduction

Imagine an internet web site that offers a service, in which users can search for restaurants. Most systems are designed to be able to handle peak-load in order to provide a certain quality of service to the users at any given time. Distributed systems are usually deployed in fixed infrastructures (clusters), by acquiring the appropriate amount of resources to handle peak-load with investments made up-front. Clusters have a relatively small operational cost and a large administration cost. However, this is far from optimal. It is reasonable to assume that during the day, the service will be used more than during the night, since much more restaurants are open during the day. As also observed in practice [47], most systems are not used uniformly over time. Instead, a periodic usage behavior is observed, typically with daily or weekly patterns. Thus, designing for peak-load implies low utilization of resources, high energy cost, and waste of money.

Clouds promise to overcome these inefficiencies [10, 20]. Using cloud infrastructures, one can acquire resources automatically and use them for as much time needed, in exchange for a service fee [6]. No up-front or administration cost is needed. This makes it possible to create virtual infrastructures that change dynamically over time and are automatically adapted to the workload. This ability to change the resources used dynamically over time, is called *elasticity* [20]. Pioneered by Amazon [7], now cloud services are offered by many providers [3, 37]. Typically, clouds offer three levels of services: infrastructure (IaaS), platform (PaaS), and software (SaaS) [10]. At the IaaS level, clouds offer compute resources in the form of virtual machines (VM) [6] whose service fee is based on a per quantum pricing scheme (e.g., constant fee per hour of usage). IaaS clouds also offer network and storage resources [1]. At PaaS level, clouds offer platforms to develop applications. Finally, at SaaS level, they offer software services.

For almost a decade, clouds have attracted much attention in the research community and software industry and fundamental database problems are being revisited [19]. The ability to use computational resources that are available on demand, challenges the way we implement systems and applications [10]. The automated adaptation of the allocated virtual infrastructure based on the query workload is crucial for energy efficiency and cost savings. A logical question that arises is whether elasticity affects the response time of the system and in which way. It seems reasonable that the response should be faster when more resources are used, however the monetary cost of using them is higher.

An Additional kind of Elasticity: In addition to the elasticity presented earlier, there is another kind of elasticity present in clouds, which captures the trade-offs between the response time of the system on a

* Dissertation Advisor: Yannis Ioannidis, Professor.

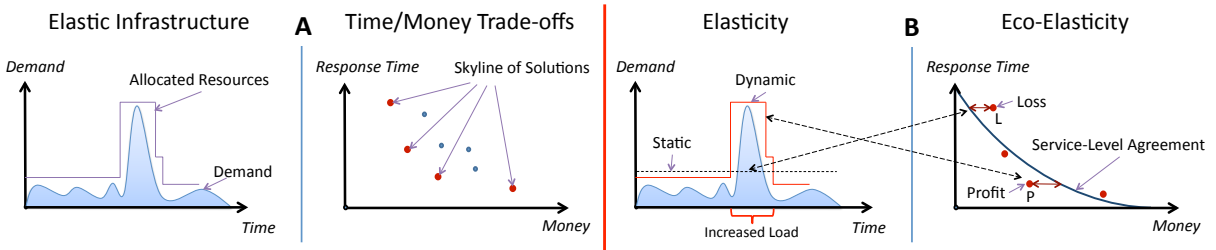

Fig. 1. A) elasticity: dynamic virtual infrastructure (left) and eco-elasticity: time/money trade-offs (right), **B)** Relationship of elasticities in IaaS clouds.

query and the amount of money we pay for it [17, 24, 29, 44]. To distinguish it from the traditional notion, in this dissertation, we refer to it as *eco-elasticity*, since this term is borrowed from economics [48]. Figure 1(A) shows both elasticity and eco-elasticity properties of clouds. Elasticity is illustrated in the left part; The system reserves additional resources to be able to meet the demand, and releases them when the demand is reduced. Eco-elasticity is illustrated in the right part; The figure shows different strategies of execution, with each solution being a point in the 2-Dimensional space that corresponds to a different trade-off between time and monetary cost. The optimal trade-offs are the ones that belong to the skyline [48]. This observation motivates our work. An elastic IaaS cloud-enabled system may allocate and de-allocate compute resources dynamically, trying to identify the optimal trade-offs between execution times of a given workload and the monetary cost of using the resources. To make this possible, we consider a unified model of both elasticities.

A Unified Elasticity Model of Clouds: Both elasticities are strongly related to each other. The size of the allocated virtual infrastructure essentially corresponds to the *Investment* on resources. A particular virtual infrastructure size provides a certain *Return on Investment (RoI)*, that is ultimately related to the fee charged to the users of the service, possibly in the form of service-level agreements (SLAs). A typical SLA is a function of money charged for the service provided based on the response time of the system on the issued queries [51, 52]. This fee is related to the trade-offs between time and monetary cost of using the resources. This relationship is illustrated in Figure 1(B). The right part of the figure shows a typical SLA that specifies the fee that the service charges given the response time (or quality of service provided); faster response times correspond to higher fees. The profit generated is computed as the difference between the fee charged to the users and the cost of the allocated resources. Consider two resource allocation alternatives as shown in the left part of Figure 1(B): static and dynamic. Each alternative corresponds to a strategy that maximizes profit as shown in the right part of the same figure. Under normal load, both static and dynamic should select the P point of operation in the skyline of time/money alternatives, since that maximizes profit. However, if the same amount of resources is used when the load is increased, they are shared among many more users, and the response time of the system is decreased. The static allocation strategy will force the system to operate at point L, which implies a loss since the cost of executing queries is higher than the fee charged to the users. Using dynamic allocation, the system is able to allocate additional resources when the load is increased, allowing it to operate again at point P. This example illustrates that both elasticities are very important and should be taken into account in a cloud environment in a unified approach [19, 29].

2 Dissertation Summary

Our vision is to build self-organizing elastic data processing systems that automate capacity planning, exploiting both elasticities of IaaS clouds in a unified approach: dynamically changing the size of the allocated virtual infrastructure and taking into account the monetary cost of using the resources. In this section, we discuss our work that is inspired by both elasticities of clouds, and approaches some of these new challenges from the elasticity point of view of IaaS clouds.

Dataflow Scheduling on the Cloud: Initially, we consider the elastic scheduling of dataflows on an IaaS cloud environment [24, 29]. Queries expressed in high-level languages are optimized and are typically transformed into dataflow graphs in the form of directed acyclic graphs (DAG) with operators as nodes and data dependencies as edges. In a distributed environment, the optimizer must decide, among others, where each node of the graph will be executed. Scheduling the processing nodes of a dataflow graph onto a set of available machines is a well-known NP-complete problem, even in its simplest form [21, 39]. Traditionally, the only criterion to optimize is the completion time or makespan of the dataflow, and many heuristic scheduling algorithms have been proposed for that problem [31]. In this work, we show that eco-elasticity is present in many types of computations that typically appear in practice. We propose a simple, yet effective search algorithm for the dataflow scheduling problem on the cloud, to efficiently explore the 2D search space of time and money to find skyline schedules. The algorithm does not assume a fixed size of infrastructure, making it ideal for automated capacity planning for IaaS clouds. Our approach is able to successfully find trade-offs

between execution time and monetary cost, and thus, exploit the eco-elasticity property of clouds. Since our approach is generic and deals with a fundamental problem at the heart of all distributed data processing systems, our algorithms could potentially be used to incorporate elasticity into many different systems.

Analytical Query Workloads: Next, we consider the elastic execution of analytical query workloads [26]. Many of these queries perform joins and heavy aggregations and often include UDFs. The most efficient way to process them is using tree execution plans of a specific form [35]. In this work, we develop an engine with a suite of specialized techniques that take advantage of the form of such plans and process them very efficiently in an IaaS cloud environment. The engine offers its services for a fee according to service-level agreements (SLAs) associated with the incoming queries. We lay out the allocated VMs in a "tree" shape so that query execution plans are mapped naturally to the processing resources. Furthermore, we introduce an online algorithm that exploits the elasticity of clouds to *dynamically adapt* to the query workload by allocating and deallocating VMs so that the processing engine maximizes its profit after removing the costs it incurs in using the cloud resources. We present an extensive experimental evaluation that demonstrates that our approach is very efficient (exhibiting fast response times), elastic (successfully modifying the cloud resources it uses as it adapts to query workload changes), and profitable (approximating very well the maximum differential between SLA-based income and cloud-based expenses).

Automated Management of Indexes: Furthermore, we investigate the automated management of indexes, that is a typical way to accelerate dataflow execution [28]. The automated management of indexes, views, and in general data structures, has always been an interesting and challenging research topic for the database community. The traditional problem is constrained by the total storage needed or the time required to build them. We investigate this problem taking into account the monetary cost to maintain indexes, which is equally important in a cloud environment. In this work, we identify the opportunity to use idle compute resources that are charged by cloud providers to eliminate the monetary cost of building indexes. This phenomenon emerges because of the nature of dataflows and the prepaid leasing policy of compute resources. We propose an online auto-tuning algorithm to assess the importance of indexes taking into account the trade-offs between the dataflow speed-up they offer and the monetary cost needed to store them, maintaining only beneficial indexes. Furthermore, our algorithm eliminates the cost to build indexes by efficiently using idle compute resources without delaying dataflow execution using appropriate scheduling techniques. Our experimental analysis reveals that we are able to increase the utilization of resources and significantly reduce both execution time and monetary cost needed to execute dataflows.

The Exareme **Elastic Processing System:** Finally, we discuss Exareme [30, 46], a system for the elastic large-scale data processing on the cloud that defines the context in which the three problems discussed in the previous sections are investigated. The system offers a declarative language based on SQL with user-defined functions (UDFs) extended with parallelism primitives to declare potential data parallelism. Building the appropriate runtime environment is essential to fully exploit the elasticity of clouds. We design and implement the elastic functionality of the Exareme system, incorporating the techniques proposed in this dissertation. We present the relevant component design and the dataflow language we developed. The language offers several parallelism primitives to declare potential data parallelism and let the system make the actual decisions at runtime. We also present the results of several experiments that demonstrate the effectiveness and promise of our approach. To the best of our knowledge, Exareme is the first effort to build a system that exploits both elasticities of clouds.

2.1 Related Work

There are several areas of data management where related work has been conducted. We briefly outline here key results from the fields of data warehouses, *NoSQL*-systems, and elasticity.

Data Warehouses: Data Warehouses store very large volumes of data and are typically used for report generation and historical analyses to discover trends. Several systems have been implemented that are open–source (e.g., *Hive* [42]), proprietary (e.g., *Tenzing* [13]), or commercial (e.g., *Vertica* [32]). The most popular open–source warehouses are based on *MapReduce* [16, 42] and typically offer high level languages (e.g., *SQL*) to express queries. The latter are ultimately transformed to one or more *MapReduce* jobs [33]. The *MapReduce* abstraction however is not efficient for heavy aggregate queries that we target in this work. In *MapReduce*, multi-level aggregations can only be expressed using multiple jobs, rendering the approach less efficient than that of a tree abstraction. Moreover, the optimization goal of these systems is to both minimize the number of jobs they produce as well as to maximize parallelization in order to minimize their total execution time. The monetary cost of the resources is by and large ignored. The same holds for *Dremel* [35] and *Scuba* [5] which has been recently proposed as specialized systems targeting query–tree executions, and, furthermore, to the best of our knowledge, are not elastic.

NoSQL–Systems: Several systems have been proposed to manage data in formats different than relational tables. Examples include *MongoDB* [14], *Sawzall* [38], *PigLatin* [36], and *FlumeJava* [12]. All of the

above are built either on top of *MapReduce* and so, they inherit all pertinent weaknesses mentioned earlier, or built from scratch by following approaches that are not suitable for the queries we target here [14]. Furthermore, no such system offers a clean and simple way to define new UDFs and their properties so that they may be used during optimization.

Elasticity: Several works focus on cloud elasticity [43, 45], and dynamically allocating resources to increase performance. A recent work [40] focuses on how to minimize the number of VMs used to save on cost, but this is not a plausible strategy in our setting where queries are associated with SLAs and the goal is to maximize profit. Some works examine cloud elasticity in the context of in-memory distributed transactions [15]. In our setting, the data are updated using bulk loading every day or week. Elasticity for array databases is examined recently [18]. This work, similarly to our methodology, makes predictions about the future based on past queries. However, the proposed algorithm is only applicable to array-based scientific data (that only grow in size and rarely deleted) and considers only increasing the size of the virtual infrastructure. We focus on a more generic problem.

3 Results and Discussion

In this section, we present in details our approach for the elastic processing of analytical query workloads. These workloads are present in many modern applications that face the need to process voluminous data using ad-hoc queries [27, 38, 41]. They also call for the use of complex user-defined functions (*UDF*s) that do not come from a pre-defined set of operators with well known semantics for which `SQL` proper is often not sufficient or efficient to use. Further, these queries must demonstrate very fast and near-interactive response times [2, 5, 35]. It has been shown that, in appropriate computational environments, specific tree execution plans, can answer queries of the above kind on trillions of objects in seconds [5, 35]. Figure 2(A) shows a generic image of such a tree execution plan: the leaves of the tree represent the data that are partitioned appropriately based on the application. The remaining nodes represent operators (e.g., `group bys`) and the connections between them correspond to operator dependencies. The operators at the first level (L_0) typically perform joins and filtering. The internal operators (levels L_1 to L_{n-2}) perform partial aggregations. Finally, the root operator (level L_{n-1}) performs global aggregations and produces the final result.

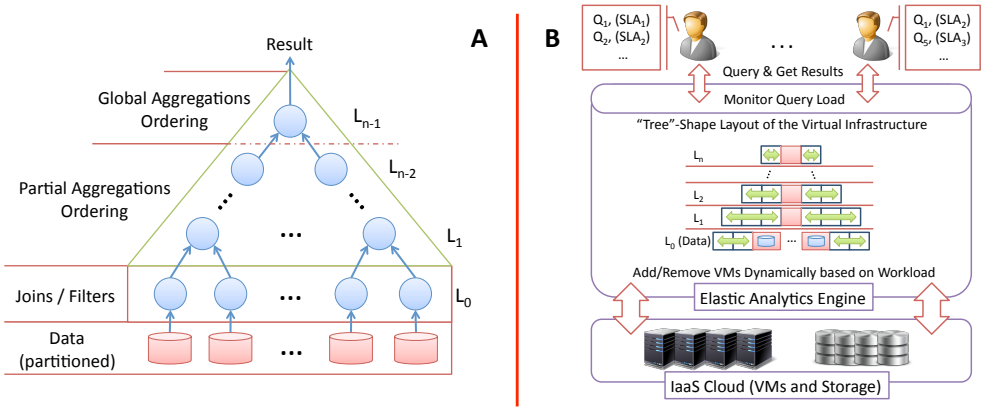

Fig. 2. A) Generic form of tree execution plans and B) Engine for Elastic Analytical Query Processing

Several systems have been proposed for large-scale data processing [2, 35, 42]; they are typically built on top of *IaaS* clouds [10, 20] which have emerged as an attractive platform for analytical query processing. The defining characteristic that favors *IaaS* clouds over other competing environments is *elasticity*, i.e., the ability to lease compute and storage resources on–demand and use them only for as long as needed. This makes possible to create an *elastic virtual infrastructure* that may change over time. *IaaS* clouds offer compute resources in the form of virtual machines (*VM*s). The cost of leasing a *VM* is determined based on a per time-quantum pricing scheme, where one pays for the entire quantum independently of the extent of the use of the *VM* resources. An elastic cloud-enabled engine may allocate or de-allocate *VM*s dynamically, trying to identify the optimal trade-off between the need to minimize execution times for a given workload and the requirement to minimize the monetary cost of using the cloud resources [19, 29].

In this work, we develop an elastic processing engine operating atop an *IaaS* infrastructure that is capable of executing efficiently and cost-effectively a large class of analytical queries demonstrating a tree execution plan of a specific form. We have implemented the functionality within Exareme [25, 46], our system for dataflow execution on the cloud. Figure 2(B) depicts the salient characteristics of our engine: arbitrarily complex queries, possibly having *UDF*s with arbitrary user–code, are continually submitted to the engine. Each query is associated with an *SLA* that designates the price that a query instigator must pay for answering

the query depending on its response time (faster response times are associated with higher prices). The data is originally stored on the cloud and is partitioned to increase flexibility and performance.

In this context, our proposed engine and its requisite mechanisms make the following contributions:

• We introduce an online algorithm that exploits the elasticity of *IaaS* clouds to *adapt* the size of the virtual infrastructure to the query workload at hand by dynamically allocating or de-allocating *VM*s. This is done so that our engine maximizes its profit while taking into account the monetary cost of expended cloud resources as well as the *SLA*s of the submitted queries.

• We propose to lay out the *VM*s allocated in a "tree" shape, so that query execution plans are mapped naturally to *IaaS* processing elements. The *VM*s at the leaf-level *fetch data* from the cloud storage and cache it to their local disk for processing, thereby decoupling compute and storage resources. For partition assignments we use an extension of *consistent hashing* and devise a simple, yet quite accurate, analytical formula to approximate the cost of partition reassignment; we use this formula when our online algorithm searches for an optimal choice when considering changes in the deployment of resources.

• We have implemented our approach within Exareme and have performed an extensive experimental evaluation which indicate very promising results. Our method compares favorably to *Cloudera Impala* [2] on sheer performance offering near-interactive response times, it adapts quickly to workload changes, and it increases the processing engine profit significantly compared to static infrastructures.

3.1 Problem Formulation

We present in details key aspects of the problem we address and the relevant notation and definitions.

IaaS Cloud: A *container* or *VM* is the unit of cloud **compute resources** and includes CPU(s), memory, disk(s), and network resources. All containers furnished for general use have the same size, i.e., the same capacity in every type of resource they provide, e.g., equal memory size. By and large, this is typical of most clouds where only a limited number of *VM*s has substantially enhanced resources to help them run core services (e.g., *namenodes* for *Hadoop* [9]). The price M_Q^c for using a container is a fixed amount in $ per time quantum T_Q. The set of containers allocated to a cloud application, such as our query processing engine, constitutes the **virtual infrastructure** of the application. The cloud also offers **data storage resources**, which are decoupled from its compute resources for flexibility. *VM*s transfer data from these storage resources and cache it to their local virtual disks for processing.

Data Partitioning: Tables are **partitioned** and **replicated** so that joins (if any) are local to containers and only aggregations require data transfer. Hence, partitioning is based on foreign keys used in joins. If the database has only one table (the usual case in *NoSQL*–systems), it is partitioned randomly into shards of equal size. If the database has multiple tables as it happens in data warehouses, the largest tables (one or more, depending on the available storage) are partitioned and all others are replicated wherever the partitions are stored. In this regard, in the *TPC-H* benchmark, it may be most beneficial to partition the two largest tables `lineitem` and `orders` with hash partitioning on `l_orderkey`, which is a foreign key in table `orders`, and replicate the other tables. This is precisely the partitioning scheme we use for *TPC-H* in our experiments.

Properties of Analytical Queries: Issued `SQL` queries may include filters, joins, and two types of group aggregate functions: **distributive** and **algebraic** [22]. Distributive functions are directly parallelizable, as they are commutative, associative, and for a table T with two partitions T_1, T_2, satisfy the property $f(T) = f(f(T_1) \cup f(T_2))$. Examples of such functions from `SQL` include `min`, `max`, and `sum`. Algebraic functions are indirectly parallelizable, as they can be expressed as algebraic combinations of distributive or other algebraic functions. Examples from `SQL` include `count`, `avg`, `stdev`, all expressed as increasingly more complex combinations of `count` and `sum`. More importantly, the queries we support may also include *UDF*s with arbitrary code that may correspond to distributive or algebraic functions. A *UDF*-example is the function of `reservoir sampling` [49] which randomly selects a subset of a table's records with equal probability.

Using the above properties, we may readily transform flat queries into tree plans by recursively unwrapping all algebraic functions until only distributive functions are left. For example, consider two tables `R(A,B,...)` and `S(B,...)`, both partitioned on column `B`, and the following flat query:

select avg(A) as AA from R, S where R.B = S.B

We transform the above `SQL`-statement into a tree-based one using the following four "conceptual queries": *leaf*, *internal-initial*, *internal-recursive*, and *root*. The particulars of each query are as follows:

Leaf (carrying out filtering and joins): select A from R, S where R.B = S.B;

Internal-initial (executing the distributive aggregate initialization):
select sum(A) as SA, count(*) as CA from leaf;

Internal-recursive (producing partial distributive aggregation(s)):
select sum(SA) as SA, sum(CA) as CA from internal-initial;

Root (compiling sought algebraic aggregation(s)):
select sum(SA) / sum(CA) as AA from internal-recursive;

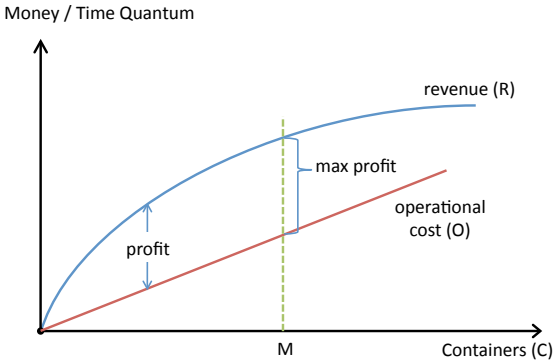

Fig. 3. Profit maximization based on revenue and operational cost.

The above conceptual queries have to be placed on the query execution tree. The *leaf* queries are placed at level 0 of the execution tree in order to be executed in parallel on each partition. Since *internal-initial* also functions on each partition independently, this type of query can be part of level 0. Between level 1 of the tree and its root, we place *internal-recursive* queries. Given the commutativity and associativity of distributive functions, there may be an arbitrary number of levels of *internal-recursive* queries, without affecting correctness. The actual number of the internal level of the resulting query–tree depends on the size of the original tables and the affordable degree of parallelization. Finally, note that, for a query without algebraic functions, the *root* query is identical to the *internal-recursive* query.

Service Level Agreement: An *SLA* is a function having query execution time as input and money as output, namely, $SLA : \mathbb{R}^+ \rightarrow \mathbb{R}$, both in appropriate units, often in seconds and dollars respectively. *SLA*s can be step-wise or more sophisticated [51]. Inspired by other works, we use a generic form of *SLA*s defined as follows: $SLA(u, q, t) = \alpha \cdot e^{-t/\gamma}$, where α and γ are respectively regulators of the maximum amount of money a user pays and the monetary cost reduction rate with time. A small γ indicates a critical query that should be rapidly executed as its value drops drastically. Alternatively, a large γ indicates a best-effort query.

Profit Maximization Problem: The queries are issued to the engine in a streaming fashion. Each query is associated with its own *SLA*. The price of the query charged is computed using both its *SLA* and its execution time. The **revenue** generated by the engine during a particular time period p is computed as the summation of the prices all queries launched during the period in question. The **operational cost** in p using c containers is computed as: $O = c \cdot p/M_Q^c$. The **profit** P during the same period is computed as $P = R - O$. Our optimization objective is to maximize the provider's profit during the operation of the engine, i.e., maximize the difference between operational cost and revenue.

Figure 3 illustrates our optimization goal; it shows a typical revenue curve per time quantum as affected by the number of containers [45]. The y-axis indicates the rate with which the revenue is generated. The figure also shows the operational cost of the engine per time quantum, which is linear to the number of containers allocated as the incurred expense for every *VM* by the provider is the same. Our goal is to identify a point M, that is the optimal number of containers that help maximize profit, i.e., the difference between revenue and operational cost is maximized. Notice that that the revenue function is a "moving target" as it highly depends on the query workload and so, M does *change* over time. The engine should be able to dynamically adapt to workload changes and find the optimal point of operation at any moment.

3.2 Overall Approach

In this section, we present the overall approach we use to maximize profit. Time is separated into windows of fixed length (e.g., epochs of 300 seconds) and inside each window we do not adjust the virtual infrastructure. All queries issued within a window, are scheduled assuming a fixed container layout. In the beginning of each window, we compute the new layout based on the measurements collected from the queries in a number of previous time windows while taking into account data re-configuration cost. In this section, we discuss the data partitioning scheme we employ, the elastic container layout, our online elastic layout allocation approach, and the query scheduler we use.

Container Layout: A **container layout** is a hierarchical overlay on top of the allocated containers that defines the allowed communication channels between them. Figure 2(B) shows this generic layout. Each level has a fixed number of initial containers (shown in red in the figure) and is elastic, i.e., can change in size by allocating or deleting containers while enforcing optional minimum/maximum thresholds. The table partitions are located at the lowest level of the layout. Each *VM* found at internal level L_i can communicate *only with* the levels above (L_{i+1}) and below (L_{i-1}). Trees with height of 4 or more are rarely needed in practice and only appear in very large data centers [35]. For this reason, we use 3 levels in our setting, however this is configurable.

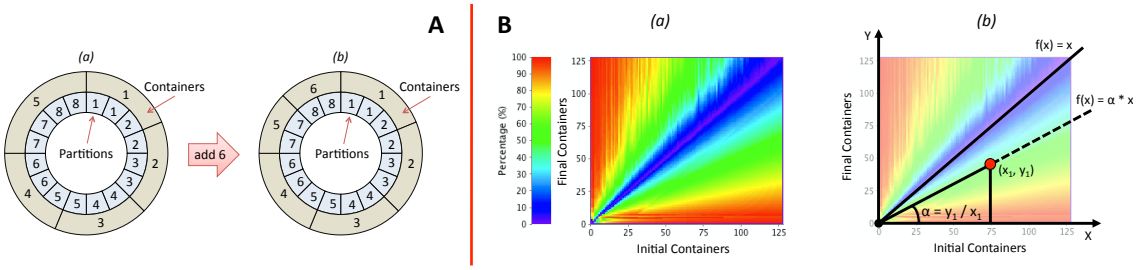

Fig. 4. A) Partitioning and Placement of Data using consistent hashing and B) Percentage of partitions assigned to a different container when changing their number (a) and the modeling of data movement (b).

Data Partitioning and Placement: Our method is based on *consistent hashing (CH)* [23] as its present good theoretical bounds on the size of data required to move when containers are added or deleted. Table partitions are placed in a logical circle as shown in the inner-circle of Figure 4(A.a). The outer circle consists of the deployed containers at L_0 with each one assigned one or more partitions. For example, partition 3 is assigned to container 2. Notice that we place each partition multiple times in the inner circle. The first time a partition is accessed from the cloud storage is cached for subsequent usage. When a new container is added, it is placed in the outer circle at a position next to the container having the largest number of partitions; the latter sheds half of its data partitions to the new arrival. For example, when new container #6 is added, is placed next to #5; Containers #5 and #6 then split the existing partitions as shown Figure 4(A.b).

To increase parallelism and flexibility we use **over-partitioning** and **replication**. We partition the tables into many more parts than the number of maximum data containers predicted to use. Thus, changing the number of containers will cause only data transfers between the cloud storage and *VM*s, yet, it does not call for extensive re-partitioning on the cloud storage; this last operation is in general very expensive and incurs high network traffic [34]. Furthermore, we employ replication by adding each partition multiple times to the inner circle of Figure 4(A) in adjacent positions. Thus, when high parallelism is needed, the same partition will be assigned to multiple containers. Here, we balance the load between the containers that have assigned replicas. If more than one replicas happen to be assigned to the same container, we keep only one copy.

Figure 4(B.a) presents the outcome of an experiment using *CH* with 128 partitions and replication degree 3 whose goal is to demonstrate the robustness of the method. The x and y axes show the initial and final number of containers (i.e., going from x to y containers). If $x<y$, then new containers are allocated, otherwise are deleted. We observe that when the changes are near the diagonal of the $2D$–space, *CH* is robust to changes as the percentage of partitions requiring for re-assignment remains low ($\leq$10%). This characteristic makes *CH* ideal as a partition placement policy for our elastic processing engine. We need to model the above behavior of *CH* to use it in our optimization process and thus, take into account data re-organization when adjusting the size of the deployed virtual infrastructure. Figure 4(B.a) reveals a strong linear correlation between the number of containers and the percentage of partitions moved. Figure 4(B.b) provides the sought model that predicts the data needed to be transferred when the number of *VM*s changes. Let x and y be the previous and new number of containers. The size of data that have to move is modeled as: $size_d(x, y) = (1-min(x/y, y/x))\cdot data\_size$, with $data\_size$ is the total volume of the tables taking into account partitioning and replication. Factor $min(x/y, y/x)$ is used to remove the symmetry of the $2D$–space on the diagonal.

Elastic Layout Allocation: Our suggested algorithm for *Elastic Layout Allocation* helps dynamically change the container layout based on the query workload received to maximize profit. The proposed online algorithm works as follows: it uses the queries issued on a historical window W_H, their CPU load, and the data the queries transferred through the network. Using these statistics, the algorithm makes predictions for a window of size W_P in the future [8, 18]. We model the profit as a multivariable function, representing each level of the container layout with a variable that indicates the number of containers allocated (l_i). The goal is to find the optimal number of containers in each level that maximize profit in the prediction window. In our experiments, we use a historical window of 2 epochs (i.e., 600 seconds) to make predictions for the upcoming window of 300 seconds. Notice that a large W_H will cause the engine to adapt slowly to the workload and low W_H may cause it to change rapidly: both extremes are not ideal. We experimentally ascertained that these window sizes behave well and leave for future work the automated learning of these numbers.

The queries are separated into a finite number of classes each having its own *SLA* which is the usual case in practice [45]. We denote as $\overrightarrow{\alpha}$ and $\overrightarrow{\gamma}$ the vectors carrying the respective values for all *SLA*s. Let $\overrightarrow{Q_H}$ be the vector with the number of queries per *SLA* that have been executed during the historical window W_H. The total number of queries is $numQ_H = \sum_i(\overrightarrow{Q_H}[i])$. We denote as $\overrightarrow{L_H}$ the current number of containers allocated at each level of the layout. Similarly, $\overrightarrow{CPU_H}$ is the vector with the sum of CPU loads at every level of the layout within the historical window and $\overrightarrow{NET_H}$ is the total amount of data transferred outwards every level. Furthermore, we designate *conc* to be the average number of queries running concurrently at any point in time. We compute *conc* by summing the execution times of all queries within the historical window

and divide this number by the length of this window. All the concurrently running queries share the same resources, and thus, they implicitly affect each other.

Dealing with the prediction window W_P, we denote as $\overrightarrow{L_P}$ the *VM* topology computed. Using the historical measurements and $\overrightarrow{L_P}$, we can predict the average running time of the queries in the prediction window as:

$$t_P = \frac{conc}{numQ_H}\Big[\frac{\overrightarrow{CPU_H}[1]}{\overrightarrow{L_P}[1]} + \sum_{i=2}^{|\overrightarrow{L_P}|}\Big(\frac{\overrightarrow{CPU_H}[i]}{\overrightarrow{L_P}[i]} + \frac{\overrightarrow{NET_H}[i]}{net\_speed \cdot min(\overrightarrow{L_P}[i-1], \overrightarrow{L_P}[i])}\Big)\Big]$$

where $\overrightarrow{CPU_H}[i]/\overrightarrow{L_P}[i]$ is the CPU load per container at level i of the layout. The factor $1/numQ_H$ above calculates the average time expended per query and we have to multiply by *conc* in order model the delay that each query poses on others running concurrently. At this point, our model assumes that it can achieve perfect load-balance at every level of the layout. The rationale behind this is that we have many operators at each level, and each of them is not expensive to execute. Given that, we can solve the relaxed problem and round the solution to integer values. The total network time of each container at level i is computed as: $\overrightarrow{NET_H}[i]/(net\_speed \cdot min(\overrightarrow{L_P}[i-1], \overrightarrow{L_P}[i])$ since the maximum network throughput between two consecutive level i-1 and i is determined by the minimum number of containers in these two levels.

We separate the prediction window into two parts: the first involving re-organization along with query execution (denoted as t_P^d) and the second involving query execution only. The length of the first period is estimated by the time needed to perform data re-organization using $size_d(x, y)$ defined above as follows:

$$t_P^d = \frac{size_d(\overrightarrow{L_H}[1], \overrightarrow{L_P}[1])}{|\overrightarrow{L_H}[1] - \overrightarrow{L_P}[1]| \cdot Arc \cdot net\_speed}$$

where $(|\overrightarrow{L_H}[1] - \overrightarrow{L_P}[1]| \cdot Arc)$ is the number of containers Arc in the circle affected by the change. These containers will transfer table partitions from the cloud storage with *net_speed* being the network speed. Thus, the length of the second period exclusively dedicated to query processing is $W_P - t_P^d$. Notice that the faster the time to re-organize the data is, the longer the period of time spent to execute queries.

Our modeling could potentially include in the re-organization part the time to create a *VM* and initialize it. A simple approach would be to consider this time a constant (e.g., 1 minute). However, in most clouds, this is relatively small compared to the actual time that the *VM*s are used and some cloud providers allow for pre-configured instances[1] which can be created in seconds, making the initialization time negligible. Most importantly however, changing the shape of the virtual infrastructure does not directly imply the allocation of new *VM*s. In our implementation, containers scheduled to be deleted, are kept until their entire quantum has finished. If the virtual infrastructure needs to grow in size, we opportunistically re-use any available containers from those scheduled to be deleted, and essentially eliminate their initialization cost.

We compute the estimated number of queries per *SLA* in each of the parts of the prediction window as: $\overrightarrow{Q_P^d} = \overrightarrow{Q_H} \cdot t_P^d/W_H, \overrightarrow{Q_P} = \overrightarrow{Q_H} \cdot (W_P - t_P^d)/W_H$ Using the above, the predicted revenue per *SLA* class for the two part of the prediction period is as follows: $\overrightarrow{R_P^d} = \overrightarrow{Q_P^d} \cdot \overrightarrow{\alpha} \cdot e^{(-(t_P^d + t_P)/\overrightarrow{\gamma})}, \overrightarrow{R_P} = \overrightarrow{Q_P} \cdot \overrightarrow{\alpha} \cdot e^{(-t_P/\overrightarrow{\gamma})}$ Notice that we include the time to perform data re-organization t_P^d in the calculation of the revenue in the first period ($\overrightarrow{R_P^d}$) of the prediction window. The total revenue and cost in the prediction window is as follows:

$$R = \sum_i (\overrightarrow{R_P^d}[i]) + \sum_i (\overrightarrow{R_P}[i]), O = M_Q^c \cdot \frac{W_P}{T_Q} \sum_i (\overrightarrow{L_P}[i])$$

The profit generated is computed as $R - O$. We seek to find $\overrightarrow{L_P}$ that maximizes profit.Since the number of container layouts is limited assuming a maximum number of containers per level (e.g., 100), we could potentially compute the revenue enumerating all different layouts. The total number of layouts with height 4 and a maximum of 100 containers/level is 10^8. In practice, this number is infeasible to compute exhaustively. Instead, we maximize the profit function using the *L-BFGS-B* Algorithm [11] which is a general purpose iterative optimization method that finds local maxima/minima of multivariable functions. Since the *L-BFGS-B* finds solutions with real numbers, we round the solutions to the ceiling.

We seed *L-BFGS-B* with the previous layout ($\overrightarrow{L_H}$) as the starting point. Extensive experimentation through enumeration of all solutions and comparison of outcomes to those derived with the help of *L-BFGS-B* showed that solutions are very close (yet, they are not identical due mostly to rounding). This was expected as changes is the topology are mostly gradual because of the data re-organization cost. The seeding the *L-BFGS-B* with the previous container layout ($\overrightarrow{L_H}$) is sufficient to adequately guide the algorithm.

[1] Okeanos: okeanos.grnet.gr, eCloudManager: www.fluidops.com

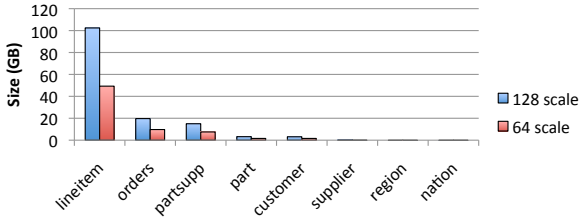

Fig. 5. *TPC-H* table size distribution at 64 GB and 128 GB scales.

Query Tree Scheduler: The execution tree plan is scheduled by performing load balance on every level of the layout while considering current load at each container. The load is quantified as the number of running and queued operators. First, we find the **rank** of each operator that is the height of the node in the execution tree. The rank of an operator determines the level of the layout at which is scheduled. As there is at least one container allocated in each level, we can always find at least one valid schedule. Once we determine the levels in which all operators are placed, we order containers at each level according to their load. The scheduler maps the operators of the each level of the query tree to the corresponding containers using the increasing ordering in a round robin fashion. For generic dataflow graphs, the scheduling problem is a much harder [29]. However, in this work we consider only tree–query plans. The specialized scheduling algorithm discussed here works because of the following two reasons: *A)* individual operators are not expensive to execute and they do not generate voluminous data as they use aggregate functions. This has as a consequence that even sub-optimal assignments of operators will not cause much imbalance, *B)* operators that are at the same level of the execution tree, will have approximately the same execution time since the data is balanced.

Our scheduling method is robust to use in practice since it neither assumes a particular operator behavior nor uses a model to predict execution times. The elastic layout allocation algorithm exclusively uses historical measurements taken after queries have been executed and so actual running times of their operators are known. Further, ongoing queries are not affected by changes in the container layout as partitions located at the respective *VM*s are not deleted even if they are re-assigned elsewhere. This is possible because of the de-coupled nature of the used compute and storage resources. Finally, our proposed algorithm is ideal when used for queries featuring *UDF*s unknown properties. *UDF*s are encountered frequently and their modeling and behavior prediction remains an open problem.

3.3 Experimental Evaluation

Here we present our experimental effort. The objectives are: *A)* evaluate our engine and show that we can achieve near-interactive response times for analytical queries, *B)* show that we can efficiently execute complex analytical queries with *UDF*s that have arbitrary user code, and *C)* examine the effectiveness of the proposed elastic container layout algorithm and ascertain its ability to adapt to the workload.

Environment: We have implemented the functionality presented within Exareme [46], our system for dataflow execution on the cloud. We compare our approach with the latest version of *Cloudera Impala*, the state-of-the-art in-memory analytics platform [2]. We deployed the systems in the *Okeanos* cloud and used up to 64 *VM*s for processing, each with 1 CPU, 4 GB of memory, and 20 GB of disk. We measured the network bandwidth to be around 150 Mbps. We set the quantum T_Q to 300 seconds and the cost of the quantum M_Q^c to \$0.41 (or equivalently ∼\$5/hour). The memory of the operators in the execution tree is set to 10% of the container's memory, i.e., at most 10 *leaf*, *internal*, or *root* queries can run concurently in each *VM*. We also used a latest version of the HDFS 2.6 as a storage service deployed in 8 *VM*s to store table partitions.

Datasets: We used two datasets namely, *TPC-H* [4] that typically models data warehouse settings, and *Freebase*, an *RDF* dataset[2]. The *TPC-H* benchmark has eight tables: lineitem(128, l_orderkey), orders(128, o_orderkey), part, partsupp, supplier, customer, region, nation. In parentheses, we indicate the number of partitions we have created for each table and the key(s) based on which we performed table partitioning. We partition tables *lineitem* and *orders* on their foreign key using hash partitioning and replicate all other tables. We used the 32 (∼32GB), 64 (∼64GB), and 128 (∼128 GB) as the *TPC-H* scale–factors. Figure 5 shows the sizes of the benchmark tables illustrating the large size difference between the fact table *lineitem* and the rest.

Freebase contains approximately 2.5 billion tuples in the form of *RDF* triples: <*subject*> <*predicate*> <*object*> *"."* and its volume stands at 250 GB. If the object is text, it is tagged at its end with the appropriate language symbol (e.g., `@en` means text in English).

Queries: we use a subset of the *TPC-H* queries that cover a wide range of the types of queries we target. In particular, we choose queries 1, 3, 4, 5, 7, and 9. 1 uses only table *lineitem* and has 8 aggregate functions. Queries 3 and 4 have a small number of joins (less than 3) and a small number of aggregate functions while queries 5, 7, and 9 feature a large number of joins and several aggregate functions. With *Freebase*, we utilize

[2] `developers.google.com/freebase/data`

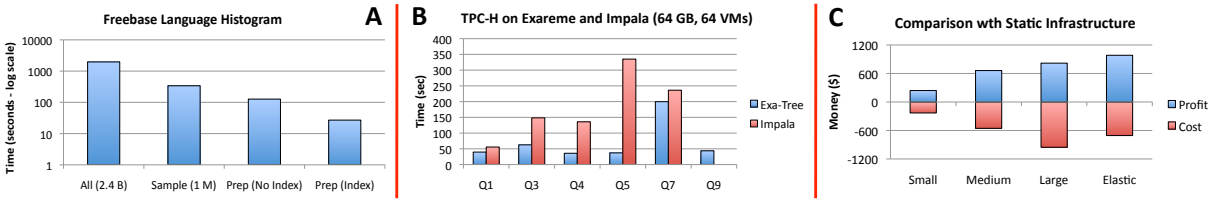

Fig. 6. A) Execution times for *Freebase* queries, B) *TPC-H* with 64 GB on *Impala* and *Exareme* using 64 *VM*sand, C) Elastic configuration vs. static layouts.

two queries with complex *UDF*s to create a histogram of the languages that appear in the dataset. The *first* query uses regular expressions to separate the language of each object and then counts the number of languages encountered. The query is as follows:

```
SELECT lang, count(lang) as c
FROM (SELECT REGEXPR('.*@(.*)', o) as lang FROM freebase WHERE o like "%@%")
GROUP BY lang ORDER BY c desc;
```

The *second* query uses *reservoir sampling* to sample 1 million rows from the table and computes the histogram though a *UDF* that is applied on the sample and detects the language of a given text using a statistical model. The query is the following:

```
SELECT lang, count(lang) as c
FROM (SELECT DETECTLANG(sobj) as lang FROM (SELECT SAMPLE(1000000, obj) as sobj FROM freebase))
GROUP BY lang ORDER BY c desc;
```

SLAs and Query Generator Client: We use two types of SLAs: "normal" with $\alpha = 10$ & $\gamma = 80$ and "high priority" with $\alpha = 20$ & $\gamma = 40$. We also created a generator that launches queries with a Poisson distribution. More specifically, the generator computes the arrival time k (in seconds) of the next query as $f(k;\lambda) = \Pr(X = k) = \lambda^k e^{-\lambda}/k!$, where λ is the expected value of X (in seconds). We can achieve desired query rates by setting λ appropriately (e.g, for $\lambda = 10$ one query is issued every 10 seconds on average).

Algorithms and Measurements: We use our elastic *VM* layout allocation algorithm to adjust the size of the virtual infrastructure. As a baseline, we select a *static layout* that remains fixed over time. We use three such static allocations: *small* with (10, 4, 1), *medium* with (26, 8, 2), and *large* (42, 12, 3); here, we designate within parentheses the number of containers per layout level starting from the lower level L_0 that contains the data. We bootstrap our dynamic layout allocation algorithm with the *medium* static configuration. Finally while experimenting, we measure the following: average execution time for queries, revenue, cost, and average number of *VM*s used at each layout level.

Near-Interactive Analytics: In our first set of experiments, we validate the efficiency of the system by executing a single type of query at a time and measuring corresponding turnaround time. We run each query 4 times and report the average of the last 3 measurements, a technique also followed by others [50]. In this way, the observed execution times reflects the behavior of the system in live operation. We use the *TPC-H* benchmark with 64 *VM*s on *Okeanos* and a 3-level execution tree. Figure 6(B) compares performance of our implementation, termed *Exa-Tree*, with that of *Impala* while using 64 GB of data on 64 *VM*s. We observe that *Exa-Tree* is comparable, and in some cases more efficient, for the types of queries we focus on in this work. This is due to our data partitioning and placement scheme that reduces network traffic during query execution (due to replication) and the tree execution plans. As *Impala* runs entirely in memory, we were not able to run query 9 because we reached memory limits.

Complex Analytics: In the second set of experiments, we assess the efficiency of our engine on complex analytics expressed in *UDF*s, again by executing a single query at a time and measuring respective execution times. As previously, we run each query 4 times and report the average of the last 3 times. We use the *Freebase* dataset and the two queries mentioned earlier in the section using 64 *VM*s. Figure 6(A) depicts the attained execution times for the two queries (*All* and *Sample*). In the first query (*All*), operators at the leaves of the execution tree take most of the time as computing 2.4 billion regular expressions is expensive. The second query (*Sample*) being highly selective completes in 339 seconds. It is worth mentioning that both queries produce similar distributions. We also pre-processed the <*object*>-column by extracting the language tag and created an additional column on the table hosting the *Freebase*. Here, the histogram on the entire dataset is computed in merely 107 seconds without indexes and in 27 seconds using indexes. This performance highlights the near–real-time capabilities of our engine in large datasets.

Elasticity under Dynamic Workloads: In this set of experiments, we examine both the effect that the elasticity has on query execution time and the profit generated. For these experiments we used TPC-H with scale factor 32. The clients connected to the system issue the queries 1 and 3 of the benchmark.

Compare with Static Infrastructures: Figure 6(C) depicts the profit gained when the static *VM* configurations are used to handle the workload as well as the profit generated by our approach. We run the

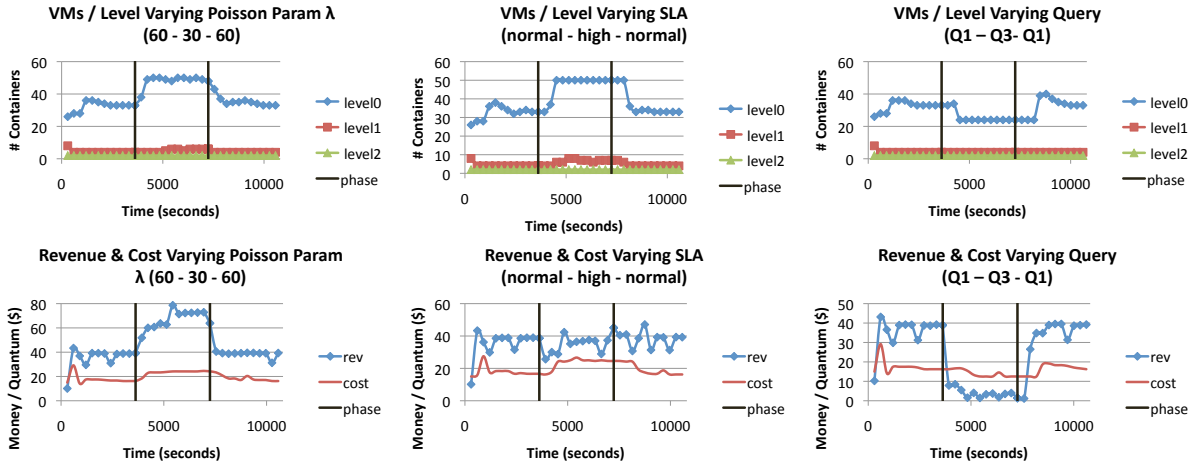

Fig. 7. Elastic containers allocated per tree level and revenue and cost for workload with different phases.

system for 3 hours using a client that issues $Q1$ in three phases, each of 1 hour duration. In the first and third phase, the Poisson parameter λ is set to 60 and in the second phase to 30 (the rate is doubled). We readily ascertain that smaller-sized infrastructures produce less revenue as expected. Similarly, the expended costs increase as more *VM*s and time quanta are used. The elastic layout allocator however produces a better-fitted layout that adapts to the workload changes and yields the highest profit compared to all static choices. Lastly, the elastic approach does generate less revenue (profit = revenue - cost) than the *large* infrastructure. However, this is in sequence with our design as we optimize for profit and not for revenue.

Adaptivity with Dynamic Workload: In our final set of experiments, we evaluate the adaptability of our elastic algorithm in presence of workloads whose features change over time. In particular, we employ a workload consisting of three stages, of 1 hour each, where query workload characteristics are perturbed between the stages. As a default workload we issue Q1 with a Poisson parameter $\lambda = 60$ and using the "normal" *SLA* . We change this default query workload in the second stage using the following three options:
Varying Query Rates: we vary the rate with which queries are issued by setting the Poisson parameter to $\lambda = 30$ in the second stage and essentially, doubling the rate. The left part of Figure 7 shows the *VM*s allocated per layout level as well as revenue. Our approach does rapidly adapt to varying workload and starts adjusting the number of *VM*s exactly at the phase boundaries. We also observe the number of containers allocated is increased along with the query rate as more revenue is generated.
Varying SLAs: we vary the *SLA* type to "high priority" during stage 2, while phases 1 and 3 have queries with the "normal" *SLA*. The middle part of Figure 7 shows our execution results: for queries with a higher price, our algorithm designates more *VM*s to be able to generate the same revenue since the *SLA* requires faster execution times for the same price.
Varying Query: we vary the type of the queries issued. In stages 1 and 3, we use $Q1$ and in stage 2 we use $Q3$. The right part of Figure 7 shows the results. Because $Q3$ is much more expensive the revenue is low. However, our elastic algorithm detects that and drops the number of *VM*s used to minimize the loss.

4 Conclusions

Our vision is to build auto-tuned data processing systems on IaaS clouds that automatically adjust the amount of resources they use by exploiting the elasticity property. In this dissertation, we have investigated dataflow processing techniques that exploit both elasticities of IaaS clouds: the traditional elasticity that is related to the ability to create virtual infrastructures that change dynamically over time, and eco-elasticity that is related to the trade-offs between execution time and monetary cost of using compute and storage resources. We proposed a set of specialized techniques for the elastic execution of analytical queries in the form of tree execution plans. These types of queries constitute a large subset of analytical SQL queries that involve heavy aggregations. We propose to layout the VMs in a "tree" shape in order to naturally map the execution plans of the queries to the virtual infrastructure. Our elastic allocation algorithm dynamically change the layout of the VMs based on the query workload in order to maximize the profit generated. A major challenge that affect elasticity is the data partitioning and placement. We used a technique based on consistent hashing since is robust to changes in the infrastructure and can be accurately modeled. We found in practice that is essential for the elasticity of the system. Our work shows that both cloud elasticities are essential and should be taken into account in IaaS clouds in a unified approach. We strongly believe that both elasticities will play an important role in the design of modern data processing systems in the cloud.

References

1. Amazon Simple Storage Service (S3), http://aws.amazon.com/s3/.
2. "Cloudera Impala: Open source, interactive SQL for Hadoop, http://www.cloudera.com".
3. Google Cloud Platform, http://www.google.com/appserve/.
4. TPC-H Benchmark, `http://www.tpc.org/tpch/`.
5. L. Abraham et al. Scuba: Diving into data at facebook. *PVLDB*, 6(11):1057–1067, 2013.
6. Amazon. Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/.
7. Amazon. Amazon Web Services, http://aws.amazon.com.
8. K. H. Ang et al. PID control system analysis, design, and technology. *IEEE Trans. Contr. Sys. Techn.*, 13, 2005.
9. Apache. Apache hadoop, http://hadoop.apache.org/.
10. M. Armbrust et al. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.
11. R. H. Byrd et al. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 1995.
12. C. Chambers et al. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.
13. B. Chattopadhyay et al. Tenzing A SQL implementation on the mapreduce framework. *PVLDB*, 4, 2011.
14. K. Chodorow et al. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010.
15. S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5, 2013.
16. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.
17. E. Deelman et al. The cost of doing science on the cloud: the montage example. In *IEEE/ACM SC*, 2008.
18. J. Duggan and M. Stonebraker. Incremental elasticity for array databases. In *SIGMOD 2014, Snowbird, UT, USA, June, 2014*, pages 409–420, 2014.
19. D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 2009.
20. L. M. V. Gonzalez et al. A break in the clouds: towards a cloud definition. *Computer Communication Review*, pages 50–55, 2009.
21. R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17, 1969.
22. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
23. D. R. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.
24. H. Kllapi, D. Achlioptas, and Y. Ioannidis. Time is money but how much? elastic dataflow processing on the cloud. Manuscript submitted for review, 2014.
25. H. Kllapi et al. Distributed query processing on the cloud: the optique point of view (short paper). In *OWLED, Montpellier, France, 2013.*, 2013.
26. H. Kllapi et al. Elastic processing of analytical query workloads on iaas clouds. *CoRR*, abs/1501.01070, 2015.
27. H. Kllapi, B. Harb, and C. Yu. Near neighbor join. In *ICDE*, pages 1120–1131, 2014.
28. H. Kllapi, V. Kantere, and Y. Ioannidis. Automated management of data structures in the cloud. Manuscript submitted for review, 2014.
29. H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. E. Ioannidis. Schedule optimization for data processing flows on the cloud. In *SIGMOD Conference*, pages 289–300, 2011.
30. H. Kllapi, L. Stamatogiannakis, M. Tsangaris, and Y. Ioannidis. Exareme: Sailing through flows of big data. Manuscript submitted for review, 2014.
31. Y.-K. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.
32. A. Lamb et al. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.
33. R. Lee et al. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25–36, 2011.
34. Y. Low et al. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5, 2012.
35. S. Melnik et al. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3, 2010.
36. C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
37. Oracle. "Oracle Cloud, https://cloud.oracle.com/home".
38. R. Pike et al. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.
39. G. M. R. et al. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of operations research*, 1976.
40. J. Schaffner et al. RTP: robust tenant placement for elastic in-memory database clusters. In *SIGMOD*, 2013.
41. A. Simitsis. Modeling and managing etl processes. In *VLDB PhD Workshop*, 2003.
42. A. Thusoo et al. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.
43. C. Tinnefeld et al. Elastic online analytical processing on ramcloud. In *EDBT*, pages 454–464, 2013.
44. I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *SIGMOD, UT, USA, June 22-27, 2014*, pages 1299–1310, 2014.
45. K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible use of cloud resources through profit maximization and price discrimination. In *ICDE*, pages 75–86, 2011.
46. M. M. Tsangaris, G. Kakaletris, H. Kllapi, et al. Dataflow processing and optimization on grid and cloud infrastructures. *IEEE Data Eng. Bull.*, 32(1):67–74, 2009.
47. J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *WSDM*, 2013.
48. H. R. Varian. *Intermediate Microeconomics : A Modern Approach*, chapter 15, Market Demand. W. W. Norton and Company, 7th edition, Dec. 2005.
49. J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
50. R. S. Xin et al. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.
51. P. Xiong et al. Admission control in cloud databases under service level agreements, 2014. US Patent 8,768,875.
52. L. Zhang and D. Ardagna. SLA based profit optimization in web systems. In *WWW*. ACM, 2004.