SLA-driven Workload Management for Cloud Databases

Dimokritos Stamatakis* Olga Papaemmanouil*
* *Brandeis University, Waltham, MA, USA*
*{dimos,olga}@cs.brandeis.edu

***Abstract*—Despite the fast growth and increased adoption of cloud databases, challenges related to Service-Level-Agreements (SLAs) specification and management still exist. Supporting application-specific performance goals and SLAs, assigning incoming query processing workloads to the reserved resources to avoid SLA violations and monitoring performance factors to ensure acceptable QoS levels, are some of the critical tasks that have not yet been addressed by the database community. In this position paper, we argue that SLA management for cloud databases should itself be offered to developers as a cloud-based automated service. Towards this goal, we discuss the design of a framework that a) enables the specification of custom applicaton-level performance SLAs and b) offers workload management mechanisms that can automatically customize their functionality towards meeting these application-specific SLAs.**

I. INTRODUCTION

While existing cloud infrastructures and services significantly reduce application development time, they still require significant effort by cloud tenants, for application deployment often involves a number of challenges including but not limited to performance monitoring, admission control and workload allocation. These tasks strongly depend on application-specific workload characteristics and performance objectives, therefore their implementation is left to the application developers.

Furthermore, despite the importance of performance guarantees for cloud databases, SLAs for cloud services are currently offered only at the service level (e.g., server uptime [1], [2], internal and external network latency/jitter [2], storage availability [2], [1]). Application-specific SLAs, i.e., SLAs that can be customized by the hosted applications, are not supported. Therefore, critical performance management tasks, such as performance monitoring, are also addressed through custom, ad-hoc solutions at the application level.

A growing number of efforts on cloud databases attempt to address some of these challenges (e.g., workload allocation [3], [4], admission control [5], resource provisioning [6], [7]) for specific performance metrics (typically query latency). However, the diversity of the data processing applications and workloads (e.g., scientific, financial, e-commerce, business-intelligence, etc) unavoidably implies the need for customizable services that support equally diverse performance metrics (e.g., throughput, response latency, load balancing, network traffic, etc). This paper describes our vision of an extensible SLA-driven framework that allows application developers to express their SLA objectives and customizes SLA-driven monitoring and workload management mechanisms.

Our proposed framework includes an SLA-specification grammar, named *XCLang*, designed to capture performance goals and constraints for data processing applications. The grammar allows application developers to declaratively express their own effectiveness and efficiency criteria for their deployed cloud-based data management application. XCLang specifications are leveraged to automatically customize a cloud-based middleware that periodically monitors these performance metrics and manages incoming query workloads.

Workload management aims to increase client satisfaction by meeting the application's SLAs. In this work we propose extensible query admission and query allocation techniques that can automatically accommodate diverse performance metrics. Admission control decides which queries should be allowed in for execution aiming to reduce the chance of overload. Once queries are admitted, query scheduling is responsible for deciding the database server they will be routed for execution such that the expected frequency of SLA violations is minimized. Both procedures are implemented by extensible mechanisms that are customizable to work for diverse SLA specifications defined using XCLang.

The paper first discusses the system model of our proposed framework (Section II). We introduce the SLA specification language XCLang in Section III. Section IV demonstrates how application-defined SLAs can be used to design extensible admission control and query assignment tools. Our final remarks and our plans for future work are presented in Section V.

II. SYSTEM MODEL

In this section we provide an overview of our proposed SLA-driven admission control model. Our system model is depicted in Figure 1. We envision our system running as a service on an Infrastructure-as-a-Service (IaaS) cloud (e.g., Amazon [1], GoGrid [2]). This service will run as a middleware lying between the application users and the application's cloud deployment and will be responsible for the admission control and allocation of incoming queries.

Our model addresses the workload management needs of data management applications. Each application is responsible for renting its own set of virtual machines (VMs) from the IaaS cloud provider. VMs run a preloaded database engine (e.g., SQL Server, MySQL, etc) and each VM hosts a replica of the application's database that serves read-only queries. We assume acceptable performance isolation across VMs. Incoming queries will be assigned for execution to one of these

VMs or rejected by the *Admission Control* component. These decisions rely on a performance prediction models trained on the VM configurations used by the application.

A unique feature of our framework is that it allows application developers to specify SLA specifications for their data management application. These specifications are expressions of our proposed *SLA definition grammar* named *XCLang* (eXtensible Cloud Language) and are submitted to our middleware. Developers use XCLang to declaratively specify their SLA metric, its composition (i.e., how it can be calculated), the SLA evaluation period, as well as the conditions under which the SLA is violated. SLA expressions are used to customize the *Statistics Manager* component. This component resides on each VM and is responsible for collecting the necessary statistics for evaluating the application's SLA. Statistics are pushed periodically to the admission control module.

SLA expressions are parsed and evaluated by our *SLA Grammar Parser* component. The parser applies the statistics collected from the current application deployment to identify (or predict) possible violations. For instance, this evaluation is applied upon the arrival of a new query and is used to identify the impact of the new query on the SLA. Based on this impact, our admission control module decides whether to admit the query and if so, to which replica it should be executed.

Our proposed model is easily extensible. For instance cloud providers can offer a set of predefined SLA templates using our grammar, each offering a different performance level for a different price. Application developers would simply customize the SLA template that fits their needs better. Finally, our grammar allows for SLA definitions on both the application and the query level. For instance, one application might want to ensure that the maximum query latency over all queries completed within an evaluation period will not exceed x secs, while another application might want to ensure that *each* admitted query is guaranteed a maximum latency of x secs.

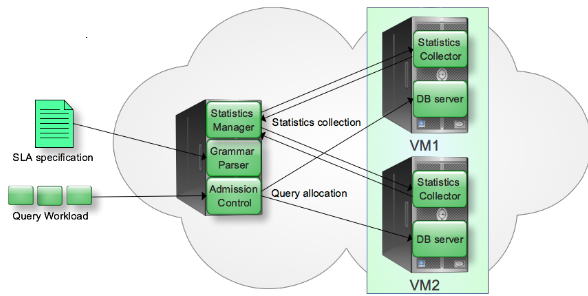

Fig. 1. Our system model.

III. SLA Specification with XCLang

Supporting extensible SLAs for diverse data management applications requires a declarative language for specifying the application's SLA metrics and parameters. In this section we describe a declarative language for specifying SLA metrics as a function of system properties and statistical performance-related metrics. XCLang aims to offer a unified grammar that facilitates the expression SLAs based on properties of different levels of granularity, ranging from cloud deployment, database, resource properties to query and operator features. Depending on their familiarity with the database internals, developers can compose their SLA metrics based on either high-level properties (e.g., number of VMs, query response time, etc) or lower level query properties (e.g., I/O requests, query plan characteristics, operators etc).

TABLE I
The built-in context-free grammar

Symbol	Grammar Rules
<Deployment>	::= <VM>*
<VM>	::= (<ID>, <DB-Engine>)
<DB-Engine>	::= (<ID>, <Query>*)
<Query>	::= (<ID>)

XCLang Our declarative SLA specification language leverages formal context-free attribute grammars (AGs). AGs provide a modular framework to define attributes for the symbols of a context-free grammar and synthesize them through *attribute rules*. XCLang represents system entities as grammar symbols and expresses their relationships through grammar rules. Examples of entities we have used so far are the following: (a) *Deployment* that represents the cloud deployment of an application, b) *Virtual Machine (VM)* that captures the VMs used by the specific application deployment, c) *DB-Engine* that represents the database engine running on each VM and d) *Query* that captures the queries executing on the database engine running on a given VM. These entities can be expanded to include query plans, query operators, etc, providing a more fine-grained definition model, if needed. However, we believe that very few developers will be comfortable expressing SLAs as a function of low level query properties and semantics.

An example grammar is shown in Table I. The grammar expresses that an application's deployment is defined as its set of reserved virtual machines (`<VM>*`). Each VM is defined by the database engine it hosts (`<DB-Engine>`) and the ID of the VM (e.g., its IP address). The database engine is defined by the currently executing queries (`<Query>*`) and the ID of the database engine. Each query is defined by its ID. Grammar rules cannot be modified by the application.

Application developers can *extend* this grammar by providing *attribute rules* that define various performance metric(s) for the system entities. They define also one of these metrics to be the SLA metric and specify the condition that satisfies the SLA. An example of an SLA specification that defines the maximum query latency is shown in Table II. Each rule refers to specific grammar symbol (e.g., `<VM>`, `<Query>`, etc) that defines the name of a metric and its evaluation rule. Metrics are categorized as *built-in* or *composite* or *predicted*:

1) *Built-in*: Attribute rules can assign a value to a metric through built-in methods that expose entity properties to the user. These methods are assumed to be available by the cloud and database providers through APIs. Examples include retrieving the execution latency of a query, or the number of running queries of a DB-engine.
2) *Composite*: One can define metrics through (optional) *multi-level composition steps*. At the lowest level, one

can specify a metric for a query and combine this (e.g., through aggregation) to express metrics for the database engine the query is running on. Similarly, database metrics can be used to define application properties (e.g., its performance metric). For instance, one can define the maximum query latency of an application by aggregating the query latency metric.

3) *Predicted*: These are metrics that are evaluated through prediction models. XCLang includes built-in performance prediction models for certain traditional metrics (e.g., query latency) and also allows applications to incorporate their own prediction techniques by applying user-defined functions on system properties.

TABLE II
ATTRIBUTE RULES EXAMPLE

Symbol	Attribute Rules
<Deployment>	SLACondition::=AppPerformance $< 60 secs$ SLAMetric::=AppPerformance AppPerformance::=`MAX`(VMPerformance,<VM>*)
<VM>	VMPerformance::=<DB-Engine>.DBPerformance numQueries::=<DB-Engine>.numQueries
<DB-Engine>	DBPerformance::=`MAX`(latency, <Query>*) numQueries::=`getConcurrentQueries(ID)`
<Query>	latency::=`getQueryLatency(ID)`

Proof-of-concept attribute rules for the above grammar are shown in Table II. At the lowest level, query performance is calculated through the method `getQueryLatency` that returns the response time of a query (built-in metric). In the next level, the performance of a database (`DBPerformance`) is defined as the maximum latency of the queries executed in that engine (composite metric). In our model each VM hosts one DB engine and hence the performance of the VM (`VMPerformance`) equals to the performance of the hosted database. The performance of the application (`AppPerformance`) is an aggregation of the VM performance (i.e., maximum query latency) over all the reserved VMs (composite metric). The SLA metric is defined as the application performance metric (i.e., maximum query latency) and it must be less than 60secs for the SLA to be satisfied.

The example demonstrates the extensibility feature of XCLang. Declarative specifications abstract away the cloud deployment (i.e., number of VMs, query assignment to VMs). This allows our framework to transparently modify the underlying deployment (e.g., adding/removing VMs) and the workload allocation without affecting the evaluation models of its performance criteria. Furthermore, XCLang can be customized for any API exported by the cloud and DBaaS providers and therefore it allows developers to experiment with different parameters that may affect the database performance. One challenge is to identify APIs that are both useful to the application developers (e.g., expose metrics that impact commonly used performance metrics) but at the same time protect the providers from exposing metrics that cannot be easily measured or controlled. We plan to study existing performance prediction models [8], [3], [6] on cloud environments and diverse workloads and identify the common APIs based on which we can accurately capture diverse performance criteria.

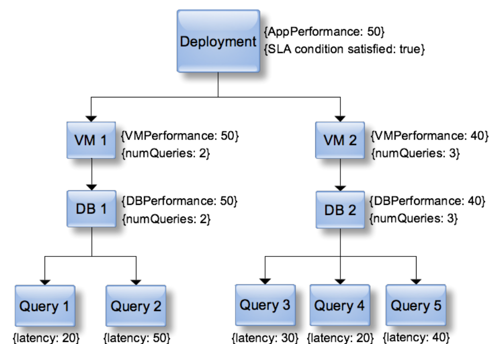

Fig. 2. Deployment representation and SLA evaluation.

SLA Expression Evaluation SLA expressions need to be periodically evaluated in order to identify (or predict) any possible SLA violations. This process is executed by the SLA Grammar Parser component. The parser implementation relies on ANTLR [9], a software for generating compilers for attribute grammars. Our SLA parser receives as input the SLA expression as well as tree-based representation of the current application deployment. An example of such representation is shown in Figure 2. The parser then evaluates the attribute values in a bottom up fashion on the deployment tree. This allows the aggregation of performance metrics on the deployment tree. Once the SLA metric is evaluated the parser is designed to check if the SLA constraints are satisfied and decide whether the SLA is met based on the current deployment and collected statistics or not. Figure 2 shows also the annotation of the tree with the attributes and their values.

The above process is agnostic to the semantics of the attributes and their evaluation method (e.g., built-in, composite, predicted). This allows the SLA evaluation process to be extensible: it can be customized to work with any SLA specification that follows the rules of our XCLang grammar. Furthermore, depending on the evaluation method specified in the SLA, the evaluation can be used to identify or predict SLA violations. For instance, if the query latency is defined to be its final execution time collected by a built-in method, the evaluation will simply detect an existing violation. However, if the query latency gets its value from a plugged-in prediction model then the process will output a predicted value for the SLA metric, allowing us to identify *expected* violations and address them proactively.

IV. SLA-DRIVEN WORKLOAD MANAGEMENT

Our workload management mechanisms leverage the SLA specification and evaluation process to make SLA-driven admission control and query allocation decisions. Workload management aims to increase client satisfaction by meeting the application's SLAs. In our project we aim to design query admission and allocation tools that can automatically accommodate diverse SLA specifications. Towards this end we propose a timeline workload analysis mechanism driven by metric specifications in XCLang. Our approach handles SLA evalu-

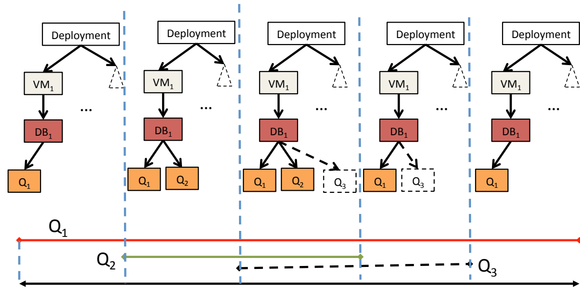

Fig. 3. Timeline analysis example.

ations on user-defined metrics for which a concurrency-aware performance prediction model has also been provided [8], [10].

Our timeline analysis aims to identify the time segments that query executions overlap and quantify the SLA impact independently for each segment. This can lead to more accurate violation predictions. For these predictions we rely on a model that evaluates the benefits and detriments of concurrent query executions [8] and can be plugged-in using attribute rules.

Let us assume we want to evaluate the expected SLA metric if a new query, q, is assigned to a rented VM. First, we quantify the progress of all queries that are presently executing in that VM since each current query began. This can be achieved with a progress indicator similar to the one proposed in [8]. Next, we estimate the expected value for the SLA metric for the new mix (i.e., existing queries plus q), operating under the temporary assumption that the mix will not change. We then pick the query with the least remaining time and under the assumption that it will terminate first, we remove it from the mix and we re-evaluate the SLA metric under the new smaller mix. We keep iteratively predicting and eliminating the query with the least remaining time until we have completed our estimates for all queries running on that VM. We aggregate the different SLA metric estimations we have collected and we return either their weighted average or the worst predicted performance, depending on the application's constraints.

An example of the timeline analysis is shown in Figure 3, where Q_3 arrives while Q_1 and Q_2 are running. The example shows the annotated deployment trees generated by the parser as the query mix changes. Using the XCLang parser this timeline analysis becomes a metric-independent process and can work with any given SLA.

Our admission control mechanism directly applies the timeline analysis model striving to reduce potential SLA violations. Specifically, we issue a request for a timeline analysis on each rented VM under the hypothesis that the new query will be executed on this VM. The query is admitted if the results are positive, i.e., there is at least one VM where the analysis indicates that the application's SLA will be met.

Once queries are admitted, the query allocation process is responsible for deciding the VM they will be routed for execution and the order of execution within that VM's queue. Again we rely on the timeline analysis to make predictions under different scenarios. For instance, we can consider alternative execution orderings within each VM or different start time offsets for the queries in the VMs queue. We can hence formulate different assignments of the submitted queries to the available VMs and run the timeline analysis to estimate the expected SLA metric of each assignment. The query is then scheduled for execution based on the assignment that minimizes the probability of SLA violations.

Similar strategies can be developed for queries that are not admitted into the system. Instead of dropping them, our timeline analysis can be used to identify cases where re-sending them in a certain time interval would allow them to be executed without violating any SLA.

V. Conclusions & Future Work

In this paper we presented our vision and initial design steps of an extensible SLA specification and workload management framework. Our framework offers a new grammar for the specification of performance criteria and performance models through which developers will be able to explore the factors that affect the efficiency of their data processing applications. By building upon this grammar, we also provide new extensible workload management tools. Supporting application-specific SLAs and seamlessly integrating them into admission control and workload allocation mechanisms will have a clear positive impact on both the development overhead and the quality of web-based data management applications.

Currently, we have an implementation of our SLA specification and evaluation module. This allows us to declaratively define diverse SLA metrics and parameters. We are also in the process of implementing the workload management tools discussed in this paper and evaluating the trade-off of different admission control and query assignment strategies. Our SLA violation predictions rely on our previous work for latency prediction under concurrent query executions [8]. This allows us to evaluate the effectiveness of alternative admission control and query assignment techniques when SLAs are expressed as a function of the query execution times. Our goal is to incorporate alternative prediction models (e.g., for throughput, load balancing) and test our mechanisms on these as well.

References

[1] Amazon web services. [Online]. Available: http://aws.amazon.com
[2] Gogrid. [Online]. Available: http://www.gogrid.com
[3] J. Rogers, O. Papaemmanouil, and U. Cetintemel, "A generic auto-provisioning framework for cloud databases," in *SIGMOD*, 2010.
[4] Y. Chi, H. J. Moon, and H. Hacigümüş, "iCBS: Incremental Cost-based Scheduling Under Piecewise Linear SLAs," *Proc. VLDB Endow.*, 2011.
[5] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ, "ActiveSLA: A Profit-oriented Admission Control Framework for Database-as-a-service Providers," in *SOCC*, 2011.
[6] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus, "Intelligent management of virtualized resources for database systems in cloud environment," in *ICDE*, 2011.
[7] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan, "Towards multi-tenant performance slos," in *ICDE*, 2012.
[8] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Performance prediction for concurrent database workloads," in *SIGMOD*, 2011.
[9] Antlr. [Online]. Available: http://www.antlr.org
[10] B. Mozafari, C. Curino, and S. Madden, "DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud," in *CIDR*, 2013.