Supporting Extensible Performance SLAs for Cloud Databases

Olga Papaemmanouil
olga@cs.brandeis.edu
Brandeis University, Waltham, MA, USA

***Abstract*— Despite the fast growth and increased adoption of cloud databases the lack of application-specific Service-Level-Agreements (SLAs) hinders the adoption of cloud data services by large-scale enterprises. Defining application-specific QoS objectives and constraints, monitoring the performance factors to ensure acceptable QoS levels and isolating the source of QoS degradation, are some of the critical tasks that are still addressed through custom, ad-hoc tools at the application level, which drastically increases the application development and maintenance overhead. In this work-in-progress paper, we argue that performance management of data management applications should itself be offered as a service. Towards this goal, we present *XCloud*, a suite of SLA management services for cloud databases that enables the definition and monitoring of application-specific performance criteria and customizable performance SLAs.**

I. Introduction

By relying on virtualization technologies, cloud computing offers a rich suite of computing, storage and communication services that enable users to easily deploy, scale and migrate their applications. This utility-driven model of cloud computing has fuelled a significant body of academic and research efforts on Database-as-a-Service (DaaS) solutions (e.g., [4], [5], [6], [8]). Despite the relatively fast growth and increased adoption of cloud databases the lack of application-specific Service-Level-Agreements (SLAs) hinders the adoption of cloud data services by large-scale enterprises, as performance SLAs are considered the cornerstone of every IT service that wishes to provide QoS guarantees.

Despite the importance of performance SLAs, currently, SLAs for cloud services are offered only at the service level (e.g., service availability [1], network performance [3]). *Application-specific SLAs*, i.e., SLAs that can be customized by the hosted applications, are not supported. Therefore, critical tasks, such as monitoring performance factors to ensure acceptable QoS levels and isolating the source of QoS degradation, are still addressed through custom, ad-hoc tools at the application level, which drastically increases the application development and maintenance overhead.

Furthermore, state-of-the-art cloud databases do not allow the specification of QoS metrics at the application nor at the end-user level but instead they are optimized for specific pre-defined performance metrics (e.g., [4], [6], [8], [10]). However, the diversity of data processing applications (e.g., transactional, analytical, etc.) unavoidably implies the need to support equally diverse performance metrics (e.g., throughput, response latency, network traffic, load balancing, etc.) and offer extensible performance SLAs that can better align cloud computing infrastructures with their tenants' needs.

Supporting application-defined performance SLAs for cloud databases raises a slew of challenges related to (a) the specification of performance metrics and (b) scalable SLA management. Expressing application-specific QoS performance goals and SLA parameters requires the design of a declarative, well-structured specification language that can express performance criteria of data processing tasks. This language should be able to seamlessly incorporate application properties and performance characteristics of the underlying data service and cloud infrastructure in order to facilitate application developers in expressing their specific QoS goals.

Furthermore, SLA management becomes drastically complex for customizable SLAs, as performance monitoring, detection and problem isolation of QoS violations become particularly critical as the number of hosted applications with diverse performance criteria increases. While existing cloud databases offer SLA-driven solutions [4], [6], [8], [10], these approaches are tailored for specific performance metrics and therefore cannot be extended for application-defined QoS criteria. Going forward, there is a need for SLA management services that can be automatically customized by application designers to handle their performance SLA expectations.

In this work-in-progress paper, we report on the design of *XCloud (eXtensible Cloud)*, a suite of cloud services that offers customizable SLA management solutions to data processing applications. XCloud relies on extensible formal grammar for defining customer-specific performance criteria. The grammar facilitates the definition of application-specific and user-specific performance models and constraints which are used to *automatically* customize the functionality of core SLA management tasks such as performance monitoring, SLA violation detection and penalty evaluation.

II. The XCloud System Model

XCloud operates on an Infrastructure-as-a-Service (IaaS) cloud (e.g., Amazon AWS [1], GoGrid [3]) that allows database providers to lease computing resources (e.g., virtual machines, storage units) on demand and host data processing applications. The system's architecture, shown in Figure 1, includes a (modified) *Data Management Service*, an *SLA Specification Service* and an *SLA Management Service*. The database relies on a multi-tenancy model where each application is deployed on different set of virtual machines.

XCloud's SLA specification service allows application designers to express performance models and expectations for

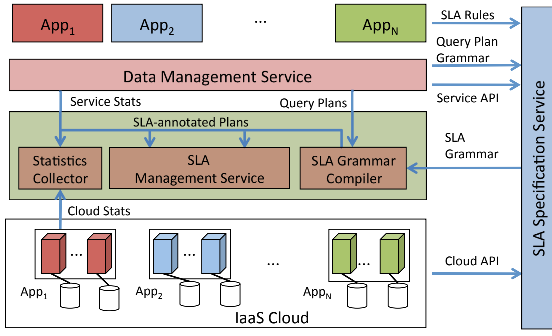

Fig. 1. The XCloud architecture.

query processing tasks through a formal grammar (*SLA grammar*). This grammar is extensible in multiple dimensions. First, the data management service defines the basic grammar for its query plan expressions, i.e., the query language symbols and the query plan structure (*Query Plan Grammar*). Currently XCloud offers a grammar for SQL and hence the query plan grammar can express query plans that include joins, selections, etc. The SLA grammar can also be customized by the cloud and data service provider to include performance-related properties of the underlying cloud and database service, respectively. Specifically, the database publishes a *service API* that exports system parameters or optimizer statistics (e.g., table and buffer pool size, selectivity, cardinalities, location of an operator/query, number of VMs per application, etc.), while the cloud provider offers a *cloud API* that exports properties of the underlying virtual machines (e.g., I/O rate, memory availability, latency between servers, etc.). Both these APIs and the Query Plan Grammar are included in the final SLA grammar published by the SLA specification service.

Application designers specify their SLA performance metrics and parameters through a set of *SLA rules*. These rules are declarative extensions to the SLA grammar and they define the performance metric (e.g., average/max query latency, throughput, etc), its evaluation model as well as the penalty if certain performance expectations are not met. The SLA Management Service evaluates the SLA rules on the query workloads and verifies the SLA compliance. To achieve this, it relies on an *SLA Compiler* which given the SLA Grammar service evaluates the SLA rules for the currently running query plans. A *Statistics Collector* periodically collects the performance metrics and properties defined in the SLA from the data service, the cloud and application. The set of currently running annotated query graphs are used to monitor their SLAs and prevent and address potential violations.

III. SLA Specification Service

The SLA Specification Service offers an *extensible* SLA language that can express traditional application-defined performance metrics and SLA parameters. We address the extensibility challenge by using an *attribute grammar* [7] as the language for defining operators, their properties and their performance evaluation models. An attribute grammar is a formal way to define attributes for the symbols of a formal grammar and synthesized them through *attribute rules*. The

```
PDAG-> Plan
Plan-> R=Flow (P1 = Plan, P2 = Plan) |

    $R.bandwth = LK-Cloud ($P1.VM, "BANDWIDTH");
    $R.cost=$P1.cost+ $P1.size/$ R.bandwth if $P1.VM!=$P2.VM;
    $R.cost=$P1.cost,  if $P1.VM==$P2.VM;

    R=Join (A1= ID "=" A2= ID, P1=Plan, P2=Plan) |

    $R.VM = LK-Cloud($R, "VM IP");
    $R.buffer= LK-DB("BUFFER POOL");
    $R.CPU = LK-Cloud($R.VM, "CPU");
    $R.cost = max($P1.cost, $P2.cost) +
             JLatency($P1.size, $P2.size, $R.buffer);

    R=Select (A = ID "= " K= LITERAL, P= Plan) |

    $R.VM = LK-Cloud($R, "VM IP");
    $R.IO = LK-Cloud($R.VM, "IO RATE");
    $R.cost = readLatency($P.size, $R.IO) + $P.cost;
    $R.size = LK-DB($P, "CARD") * LK-DB($P, "TUPLE SIZE");

    R=Table(T=ID) |

    $R.VM = LK-Cloud($T, "VM IP");   $R.cost = 0;
    $R.size = LK-Service($T, "SIZE");

SLA->  R=MAX_LATENCY

    $S.evaluationP= 2min;
    $S.pastQSet= LK-MONITOR("past queries", $S.evaluationP);
    $S.metric = MAX(Q.cost, forall Q in S.pastQSet);
    $S.penalty = if $S.metric < 20ms then 0,
                 elseif $S.metric<40ms then 0.1 else 0.2;
```

Fig. 2. An SLA specification attribute grammar.

evaluation of these attributes occurs when a grammar expression is processed by some parser or compiler like ANTLR [2].

In XCloud, the SLA Compiler translates a query plan expression to a directed acyclic graph (DAG). To be able to capture properties of distributed query plans it augments the graph with *FLOW operators* that represent data flows between two operators. A `FLOW` operator can be "executed" over the network (if data is transmitted) or locally at the location of its source endpoint. These operators are customized to express a local or networked implementation of the communication of the operators, depending on the distribution of the specific plan. In our grammar, DAG nodes (operators and flows) are expressed as grammar symbols, while their properties are expressed as attributes of these symbols and the performance evaluation models as sets of attribute rules.

The main advantage of an attribute grammar is that it offers a structured, formal language that can be easily extended in multiple dimensions. First, each data service designs the grammar symbols and syntax rules to express DAGs that capture the syntax of its specific processing language and query plan structure. For instance, a grammar for SQL will use symbols for joins, selections, etc. Second, the data service, the cloud provider as well as the statistics collector can define attributes for the grammar's symbols that capture properties of the underlying database engine (e.g., table size, buffer pool size, etc.) and cloud infrastructure (e.g., average I/O rate, CPU and memory availability, etc.) or statistics collected by XCloud (e.g., cost of queries executed so far), respectively. Finally, the grammar is extended by the application designers which define their performance models and penalty functions through the grammar's attribute rules. Different applications deployed on the same database service use the same query plan grammar and the same cloud and service attributes/properties, but they can synthesize them independently by expressing different attribute rules to express their SLA.

Performance models can be expressed in varying degrees

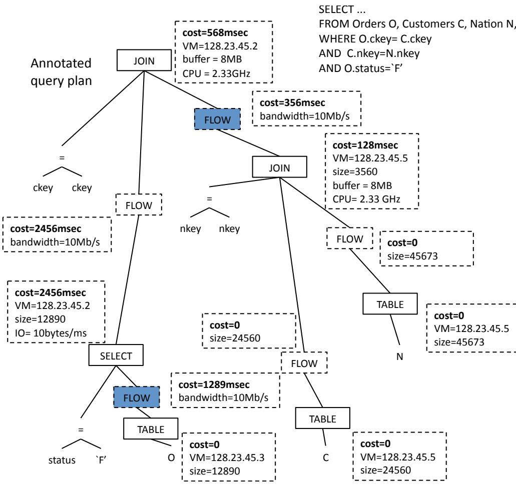

Fig. 3. An annotated query plan graph.

of granularity. Attribute rules could define the performance of operator and then express the performance of the query as an aggregation of the operators' performance. We refer to this as *white-box specification*. Note that performance models for processing operators may be defined by the database provider as it requires in depth understanding of the database internals. Applications may also provide a black-box function that predicts the query performance based on service and cloud and query properties. White-box specifications allow us to experiment with more fine-grained SLA-driven optimizations of the query execution plans. *Black-box specifications* can capture performance models that have been recently studied within the context of cloud databases (e.g., [9], [10]).

White-box SLA specification example. Figure 2 shows a simple grammar for a relational database (we show only the join and selection operators). The grammar, shown in boldface and expressed in ANTLR-like syntax, specifies that a DAG expresses a query plan (denoted by `Plan`) which can be of 4 types: (1) a plan that returns a named base table (`Table(T=ID)`, (2) a selection over a plan using an equality predicate comparing an attribute of a table with a literal (`Select (...)`), (3) an equijoin of two plans (`JOIN (...)`), or (4) a flow operator that connects two plans (`Flow (...)`). (The grammar rules that identify `TABLE, SELECT, JOIN, ID, LITERAL` and all quoted punctuations as grammar tokens are not shown here.) ANTLR notation permits any token or nonterminal used in a grammar rule to be assigned a variable name (e.g., `T in T=ID`) for use in the attribute rules that follow each grammar rule. In this example, `cost, VM, IO, memory, size, bandwth` are attributes of the nonterminal `Plan`. `VM, IO, memory`, and `bandwth` refer to the identification (e.g., IP address), I/O rate, available memory and network bandwidth, respectively, of the virtual machine hosting the specific operator, table, or start point of a network flow. These values are accessed through a lookup function exported by the cloud provider (denoted by the `LK-Cloud(VM, [property name])` and `LK-Cloud($Plan, [VM id])` functions). Similarly, the `size` of a plan refers to an operator's output cardinality or a table's size. Both can be retrieved through the data service's API, i.e., `LK-DB($Plan, [property name])`. Service-level properties (e.g., buffer pool size) are accessed through the function `LK-DB([property name])`. Figure 3 shows an annotated DAG when an expression of a logical plan of the query Q10 of TPC-H is parsed by the SLA Compiler.

In this example, the performance metric `cost` refers to the end-to-end latency of plan, i.e., the time to receive the data from the base table to the time it emits its output. The grammar shows how the latency can be specified in terms of operator properties. The example is kept simple to make it possible to show within our page constraints and it calculates a plan's latency in a hierarchical manner, where the latency of an operator is the maximum latency between its input plans plus its processing latency. For example, the application may define the latency of a selection operator as the current accumulated latency of its input plan plus the latency to read the input (which is expressed as function of the VM's I/O rate, `readLatency($P.size, $R.IO) + $P.cost`). Similarly, the latency of a join is the maximum latency of its input plans plus the processing latency of the join which is expressed as a function of the database engine's buffer pool size. The latency of a flow operator is zero if its two connecting plans/operators reside on the same machine, otherwise it equals to the time to transfer the source operator's output to its destination. This is a function of the bandwidth of the link connecting the two operators. Based on this performance metric the application defines the SLA metric as the maximum cost of the queries seen within the last 2 minutes (`MAX (...)`).

Client-level SLA Specification. Application-specific SLAs require a certain degree of expertise from the application and SLA designer. This is due to the fact that an offline performance analysis should identify the performance factors of their workload and how these are related to the properties of the underlying database service and the IaaS cloud. Therefore, the framework described above, despite its highly extensibility and expressiveness, does not provide the straightforwardness required by end-users. Furthermore, application-specific SLAs offer the same performance guarantee to all clients. However, users will most likely be interested in the performance of their own queries (e.g., "execute my query in $< xsecs$") than in application-wide guarantees (e.g., average query latency). To address these challenges, XCloud extends its support for performance SLAs to the end-user level.

The specification of client-specific SLAs requires a mechanism for controlling a) the type of queries that support performance SLAs, b) the type of performance metrics of these SLAs and c) their profit/loss margins. XCloud controls the type of SLAs clients can express by designing a set of *template SLAs* that clients can customize with their performance and budget constraints. Each template SLA will be implemented by a different attribute grammar allowing applications to restrict client-based SLAs to specific query types. For example, if the template grammar does not cover join expressions, join queries could not be parsed by the SLA Compiler. Furthermore, template SLAs would have to be compatible and not conflict

with any application-defined SLAs. Furthermore, an offline profit analysis is required to identify templates that minimize the risk of SLA violation while maximizing the profit. The analysis could identify the upper and lower bounds for their SLA metric and budget function, allowing the system to immediately reject SLA with parameters outside these bounds.

IV. SLA Management Service

The goal of SLA monitoring is to oversee the compliance of the existing SLAs and estimate the SLA penalties if violations occur. SLA monitoring has been addressed in cloud databases by independently monitoring *each* application's end-to-end performance (e.g., [10]). Applying this approach for application and user-specific SLAs has three main drawbacks. First, it does not scale as the number of running SLAs increases. Second it cannot automatically address the diversity of customized SLAs since a different monitoring mechanism needs to be developed for each SLA. Finally, it cannot distinguish the reliable provider for the performance factors and hence cannot identify the source of an SLA violation.

XCloud's SLA monitoring relies on a *"divide and monitor"* approach that leverages the SLA rules provided by the application and divides the monitoring overhead among the cloud, service and application providers. The SLA specification language clearly separates cloud, service and application properties and offers a natural way to achieve this with the assumption that applications define their performance criteria as functions of these properties. In this case, service and cloud providers can monitor only their internal properties (e.g., I/O rates, buffer pool access latency, etc.) while the application is only responsible for its workload parameters (e.g., input query rate). In this scenario, SLA compliance is verified by applying the SLA rules on the statistics collected from each provider.

All application-defined SLAs for the same cloud database are expressed over the same SLA grammar and therefore they rely on the same API available by the database and cloud provider. Therefore, our unified SLA specification approach eliminates the need for custom monitoring solutions for each hosted application. XCoud handles all custom SLAs uniformly while the data service and cloud providers need to monitor only a single set of performance properties for all defined SLAs. This increases the scalability of the system as the underlying database and cloud providers do not need to maintain any state regarding their hosted application.

SLA Violation Management. XCloud relies on the SLA grammar to derive minimum and maximum bounds for cloud and service properties and use them for identifying the source of an SLA violation. The presumption is that if these bounds are all met, the cloud and service providers are meeting the SLA agreement, while their violation indicates the accountable party (assuming these metrics are independent). To derive these bounds XCloud relies on slack splitting techniques. Given an acceptable range for a query's performance it identifies a set of acceptable value ranges for the cloud, service and application properties. The process relies on the hierarchical structure of the annotated query plans. Given the SLA constraint, the performance slack, i.e., the difference between the constraint and the current value of the SLA metric, is divided among the current query plans based on a weighted scheme that depends on their progress and expected completion time. The performance slack per query is propagated downstream its annotated query plan to derive the slack for cloud, services and application properties. This propagation is implemented by "reverse" attribute rules that reverse the performance models. Such reverse rules can be provided by the application or service provider. Given these rules, the SLA violation manager can identify when compiling the SLA rules whether particular properties fall outside their slack and therefore pinpoint the source of the problem.

V. Open Challenges & Ongoing Work

Here, we summarize several open issues we plan to address.

SLA-driven query optimization. The annotation of query plans with performance properties gives a unique opportunity to customize the cost-based query optimization process for application-specific performance goals through our SLA grammar. One approach is to extract the set of promising logical/physical plans the optimizer considers, translate them to annotated DAGs and return them back to the optimizer with a new "cost" metric that represents the application's performance measure. This would drive the optimizer to prune plans with cost beyond the acceptable limit and eventually pick a plan that satisfies the application's SLA objectives.

Automated cloud brokers. Given the plethora of cloud providers, cloud brokers aim to automatically select a provider that can meet an application's requirements and budget constraints. Supporting an automated cloud broker requires a machine-readable SLA specification framework through which application designers specify their performance criteria and the broker can automatically check these against different providers' characteristics. Our extensible SLA specification grammar is a step towards this direction. Applications could specify their SLA rules, the broker will apply the performance properties of the different providers, evaluate the expected SLA of a given workload and identify the best cloud provider.

References

[1] Amazon Web Services, http://aws.amazon.com/.

[2] ANTLR Parser Generator, http://www.antlr.org/.

[3] GoGrid, http://gogrid.com/.

[4] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, N. Zeldovich, and H. Balakrishnan. Relational Cloud: A Database-as-a-Service for the Cloud. In *CIDR*, 2011.

[5] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An elastic transactional data store in the cloud. In *HotCloud*, 2010.

[6] A. Floratou, J. M. Patel, W. Lang, and A. Halverson. When free is not really free: What does it cost to run a database workload in the cloud. In *TPCTC*, 2011.

[7] D. Knuth. The genesis of attribute grammars. In *Proceedings of the International Conference on Attribute Grammars and their Applications*, pages 1–12. Springer-Verlag New York, Inc., 1990.

[8] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards Multi-Tenant Performance SLOs. In *ICDE*, 2012.

[9] J. Rogers, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Modeling and Prediction of Concurrent Query Performance. In *SIGMOD*, 2011.

[10] 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.