# Supporting Application Development with Structured Queries in the Cloud

Michael Smit, Bradley Simmons, Mark Shtern and Marin Litoiu
*York University, Canada*
{msmit|bsimmons|mlitoiu}@yorku.ca *and* mark@cse.yorku.ca

***Abstract*—To facilitate software development for multiple, federated cloud systems, abstraction layers have been introduced to mask the differences in the offerings, APIs, and terminology of various cloud providers. Such layers rely on a common ontology, which a) is difficult to create, and b) requires developers to understand both the common ontology and how various providers deviate from it. In this paper we propose and describe a structured query language for the cloud, Cloud SQL, along with a system and methodology for acquiring and organizing information from cloud providers and other entities in the cloud ecosystem such that it can be queried. It allows developers to run queries on data organized based on their semantic understanding of the cloud. Like the original SQL, we believe the use of a declarative query language will reduce development costs and make the multi-cloud accessible to a broader set of developers.**

***Keywords*-software development, query languages, adaptive systems, cloud computing**

## I. Introduction

Developing an application for the cloud, particularly for infrastructure as-a-service (IaaS), necessitates several important design decisions. There are multiple competing providers, with different terminology, many different offerings, incompatible APIs, and various value-added features. The cloud is also evolving beyond limiting applications to a single administrative domain; a *multi-cloud* (also called the intercloud or cloud federation [1], [2]) is a collection of multiple clouds that are arranged to provide additional value to the end user.

For example, a private and public cloud can be combined to address data privacy concerns while still enjoying some public cloud benefits (i.e., *hybrid clouds*). A national research project in Canada [3] proposes an architecture where virtualized resources exist close to end users (i.e., the *smart edge*), allowing applications to access either low-latency resources near end users, or standard public cloud data centers (i.e., the *core*). Multiple public clouds can be federated to improve availability [4], reduce lock-in [5], and optimize costs [5] beyond what can be achieved with a single cloud provider.

To enable development for the multi-cloud, various abstraction layers have been created to present a unified API that can be used to manage resources from many providers. Apache Deltacloud[1] offers a RESTful API which translates requests to the appropriate provider-specific calls; other abstraction layers provide a programmatic API, like JClouds[2] for Java, SimpleCloud[3] for PHP, and Apache Libcloud[4] for Python.

These abstraction layers attempt to map disparate sets of providers, terminology, and resources to a common ontology. This task is complicated by the fact that cloud providers furnish different offerings, describe them differently, and change them frequently. The design goals of an abstraction layer are directly contradicted by the notion of frequent changes to the common ontology and this results in imperfect and inconsistent mappings. Resources and features that do not precisely map to the common ontology require the developer to write provider-specific code. In summary, an application developer must learn the abstraction layer's common ontology; additionally, they must remain well-versed and current with regards to the domain-specific intricacies and minutiae of each provider's offerings. This is frustratingly complex, scales poorly and may act as a barrier to multi-cloud adoption.

While abstraction layers are focused on actual IaaS providers, the ecosystem extends further. An application developer may need to augment information about her infrastructure with information from third party services, such as information about pricing and available resources [6], benchmarks (e.g. Cloud Harmony[5]), third party auditing services, monitoring services, and others.

We propose and describe a structured query language for the cloud, *Cloud SQL*, along with a system and methodology for acquiring and organizing information from cloud providers and other entities in the cloud ecosystem such that it can be queried (i.e., the *data model*). A developer defines *views* of information they require by choosing the data they need from service providers and mapping it to their own semantic model, allowing them to use their own knowledge domain rather than relying on an unfamiliar global ontology. The query processor assumes the complexity of building a view from the APIs of various service providers, enforcing limitations on the view, and passing modifications to the contents of the view to the various cloud providers. The details of the data model and Cloud SQL are presented in Section III.

A proof-of-concept implementation is presented in Section IV. In Section V we demonstrate the ability to respond to a variety of queries, and compare the complexity of using existing abstraction layers versus our proof-of-concept in a real application. We begin with a brief discussion of related work (Section II), and conclude with thoughts on future directions (Section VI).

## II. Background and related work

In declarative programming languages, the developer specifies a goal, and the language implementation is responsible

[1] http://deltacloud.apache.org/
[2] http://www.jclouds.org/
[3] http://www.simplecloud.org/
[4] http://libcloud.apache.org
[5] http://cloudharmony.com

for executing the steps required to achieve that goal. SQL is (mostly) a declarative language, originally proposed (as SEQUEL) to lower costs for software development and to "bring the non-professional user into effective communication" with a source of data [7]. Cloud SQL builds on this base, using a familiar language and applying the same principles to the development of applications for the cloud.

The use of queries to interact with semi-structured or unstructured data is precedented. For example, the Yahoo Query Language[6] allows users to define a data table in XML (or use an existing definition from the community), and run queries on that table. A data table is backed by a single provider, and typically uses the provider's semantics. In contrast to Cloud SQL, the developer must still work using semantics defined externally. There are also no YQL data tables currently defined for cloud providers; moreover, if tables were defined, they would use the provider's semantics. Other approaches have focused on search and retrieval (e.g. Liquid query [8], W3QL [9]) or have demonstrated the feasibility of executing structured queries on semi-structured data (e.g. Squeal [10], WebSQL [11]). Previous attempts to create a query language for the cloud have not succeeded (as evidenced by several since abandoned projects[7]) but suggest interest; this is the first functioning proof-of-concept and methodology.

Our declarative querying approach is the opposite of most attempts to resolve issues of multi-cloud interoperation. Existing abstraction layers (JClouds, Deltacloud, SimpleCloud, Libcloud, et al.) require acceptance of a global ontology and the definition of a procedure to invoke the service and acquire results. We believe efforts to define a global ontology for the cloud [12] or a cloud service measurement index [13] should be supplanted by the simple elegance of allowing a user to use her own semantics.

## III. Structured Queries in the Cloud

This section describes the overall design of Cloud SQL, including the data model (i.e., how structure is imposed on the data set) and the actual query language. The design and features described here are for the first version of Cloud SQL; we anticipate further development as the project matures. Parsing queries and executing operations on structured data are well understood, and will not be covered here.

### A. Data Model

The role of the data model is to facilitate a mapping from the the users' domain knowledge to the information and resources available at each provider. The term *provider* is figurative, not literal; a provider is anything that provides data. Thus a provider data table can be defined by anyone; for example, a set of provider data tables could be defined based on abstraction provided by the JClouds library. A table is a data structure, with columns and rows; each cell in the table can be an array. Queries are executed on their own representation of that information, called a *view*. The Cloud SQL engine is responsible for translating to and from that view from the *provider data tables*.

[6]http://developer.yahoo.com/yql/

[7]Google code projects https://code.google.com/p/cql/ and https://code.google.com/p/cloud-query-language/

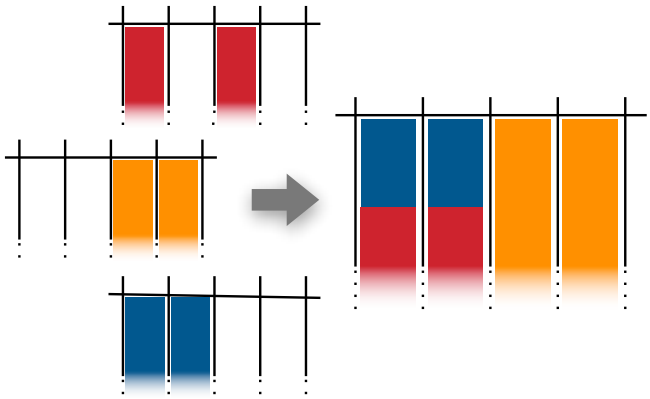

Fig. 1. Visual depiction of populating a view with data from multiple sources.

**Provider data table** (PDT): These tables are defined for each provider, or can be determined automatically based on the API the provider offers. For example, each operation of a Web Service, as described in a WSDL file, may represent a data table (i.e., as in the Yahoo Query Language). Each PDT includes a list of columns provided by that table. A table does not imply storage as in a relational database; rather, it indicates that the provider will provide a set of records on request. How these records are generated is up to the provider.

A PDT specifies prerequisites that it requires in order to return records. It also specifies optional parameters that, if included in a request for records, may alter the set of records returned. Finally, a PDT has a set of supported operations (for example, inserting a row may be meaningless or impossible for some tables).

**Views**: A view is a mapping to one or more columns from one or more provider data tables. It consists of columns, where each column has a label assigned by the user and a mapping to one (or several) columns from a provider data table. Views are defined by the user, who establishes the mappings at creation time. We anticipate a graphical user interface that will assist users in establishing a View. Once a view is defined, the user references columns using the labels they defined; the underlying source of the data is obscured. The PDTs required to populate a view are referred to as the *backing PDTs*.

Information from multiple provider data tables can be combined using aggregation, joins, or both (ordering is important). Aggregation assigns multiple PDT columns to a single view column; the records from each PDT are appended. A join functions similarly to a database operation (the fields from two records are merged if they have certain values in common), including various types of join (left join, inner join, etc.). In the illustration in Figure 1, two columns from each of three PDTs are combined into a single view. The records from the first and third PDT are aggregated into the same two columns of the view, and then joined with the records from the second PDT.

The view must also include any information required by its backing PDTs. The Query Processor is responsible for resolving pre-requisites and identifying which information may be acquired from another PDT and which must be provided by the user (a provider namespace is reserved for user-entered information). This information may be parameterized and filled in at runtime.

Additional information can be included in views to simplify operation. Views can specify data transformations, including string operations, data type conversions, unit conversions, currency conversion, etc. Views may also list credentials to be used to connect to each PDT. Finally, a view specifies which operations it supports. In order to support an operation, the operation must be supported by the backing PDTs, and the columns of the PDT required for a given operation must be included in the view (for example, rows may not be inserted into a view if it excludes columns from the PDT without default values).

### B. Cloud Structured Query Language

A subset of the language including key operations is as follows:

- `SELECT ... FROM ... WHERE ...`: Acquire the named columns (or $\star$) from the given views, where given conditions are met. Views can be joined using this mechanism, with syntax like using SQL. The view will be populated from the providers each time.
- `SELECT CACHED ... FROM ... WHERE ...`: Like SELECT, but uses results from the last time the view was populated.
- `INSERT INTO ...` $(col)^+$ `VALUES` $(val)^+$: Add a record to a view. This operation is only defined for certain views; if a view excludes some columns from a PDT, a runtime error may occur if a partial row cannot be inserted.
- `DELETE FROM ... WHERE ...`: Delete matching records from a view. A constraint similar to the INSERT operation applies.

## IV. Implementation

Our proof-of-concept implementation includes three main components: a mechanism for defining views, mechanisms for defining provider data tables, and the query processor. We also defined several PDTs and several views for testing purposes; in practice, these would be defined by users. The implementation was written primarily in Java. The use case we identified as most challenging for our case study was SELECT queries.

**Query Processor**: The implemented query processor accepts queries from a simple command line client, with a pluggable design so the client can be replaced with an API or other interface. Select queries, for example `SELECT * FROM AllInstancesWithPricing`, are parsed and converted into a set of actions that are required to return the desired results. First the view is located in the view repository, and the backing PDTs and any dependencies identified. Then the provider is contacted (through a defined Java interface) to acquire record sets from the backing PDTs, with parameters passed if available. These record sets are converted to the user's abstract representation, and then joined or aggregated together as specified. The result returned by the API is a ViewResultSet, where rows can be retrieved much like working with a database. The command line client simply prints the result.

```
<view>
  <columns>
    <entry>
      <string>InstanceSize</string>
      <column>
        <id>1</id>
        <name>InstanceSize</name>
        <referencedElement>JClouds::ListNodes::size</referencedElement>
      </column>
    </entry>
    <!-- additional entries omitted -->
  </columns>
  <name>AllInstancesWithPricing</name>
  <joinKey>InstanceSize</joinKey>
  <operations>
    <type>JOIN</type>
    <!-- details omitted -->
  </operations>
</view>
```

Fig. 2. XML representation of a view.

**Views**: In order to share views among the view creation user interface (not implemented), the users, and the query processor, we defined an XML format. This XML format can be edited directly, but can also be marshalled and unmarshalled from/into a POJO by a helper class. A sample view is shown in Figure 2. Defining two columns with the same label (`name`) but provided by different PDT columns (`referencedelement`) indicates how data should be combined; in this case, by a Join on InstanceSize. The effect of this view is to join a list of instances from multiple providers with their prices as recorded by a cloud metadata service [6], along with the status of the instance (running, stopped, etc.). In this case, no parameters are stored, and the credentials are omitted. PDT columns are referenced in a cross-implementation standard `<provider>::<table>::<column>` format.

**Provider data tables**: Our implementation defines a set of interfaces for providers to follow; the most important method is an ability to select a set of columns from a named PDT, with the results returned as a `ProviderRecordSet`. It also includes two different approaches to defining a provider. In the first, the definition is entirely programmatic: the name, tables, columns, and other values are established based on JClouds API methods. For example, the `ComputeService.listNodes()` method acquires information about compute instances. A lookup function executes the correct code to return a set of records based on the name of the PDT requested.

The second approach allows for the specification of a provider and all of its PDTs using JSON in a prescribed format. No development is required; the PDT is defined based on calls to a RESTful API, with the required prerequisites and optional parameters specified in JSON. The JSON is read into an object, which responds to method invocations by calling

| InstanceSize | CostPerHour | Status | OSFlavour |
|---|---|---|---|
| m1.small | 0.08 | RUNNING | linux |
| m1.medium | 0.23 | RUNNING | mswin |
| m1.large | 0.32 | SUSPENDED | linux |
| t1.micro | 0.02 | RUNNING | linux |

| Joined | CloudyMetrics | JClouds |
|---|---|---|

TABLE I
RESULTS OF SELECT ON A VIEW JOINING INSTANCE INFORMATION FROM JCLOUDS WITH PRICING INFORMATION FROM CLOUDYMETRICS.

the RESTful service with the specified parameters. The result from the service is parsed (it expects a JSON array filled with JSON objects by default, but an optional field can be used to specify where in the response the "row" start.) and the `ProviderRecordSet` populated and returned. This generic JSON Provider can be used for Restful services generally, like OpenStack or DeltaCloud, with little development effort, and simplifies the process of adding providers to Cloud SQL. We defined a JSON-based provider using the CloudyMetrics API [6].

## V. CASE STUDY

As an early assessment of the potential benefits of this approach, we compared the process for acquiring data using Cloud SQL versus an abstraction layer. In two previous projects, we implemented applications that required information from multiple providers using abstraction layers. Here we show how multiple lines of code can be replaced with straightforward queries on pre-defined views.

Our cloud broker project made a resource acquisition decision at runtime, selecting from among multiple IaaS providers and provisioning instances based on the needs of the application [5]. It included a monitoring feature to test the status of provisioned instances, as the startup time of different cloud providers ranges from several seconds to over five minutes. It connects to the various cloud providers using DeltaCloud. The connection requires 25 lines of code (LOC) (excluding error handling). We can replace this code by defining a view that includes the status of the instance from the JClouds provider, and run a SELECT query on the view, limiting the results with a WHERE clause.

Our distributed, scalable cloud monitoring framework includes the ability to monitor cloud resource consumption and overall cost [14], for which we used the JClouds abstraction layer to connect to multiple providers and a RESTful connection to a cloud metadata service. Records were acquired from each source, the running instances were matched with the current pricing information, and the result was returned. Acquiring the data required 97 lines of code, and performing the join required 27 lines of code. In Cloud SQL, we defined a view that joined instance information from the JClouds provider with cost information from the CloudyMetrics provider, and performed a SELECT query. The results are shown in Table I.

## VI. CONCLUSION AND FUTURE WORK

Based on our experience with various cloud abstraction layers, we concluded that they support basic tasks such as launching and terminating on-demand instances, but their support for more complex tasks such as computing the hourly cost of an instance is limited. To ease software development for multi-clouds, we proposed Cloud SQL, a language and an organization of data that allows software developers to map data from service providers to their own knowledge domain. Views based on per-developer semantics are backed by data from providers throughout the cloud ecosystem. A proof-of-concept implementation demonstrates the functioning of such a system for several use cases.

Next steps include defining a complete language specification, an evaluation of developer effort required when working in this SQL-like paradigm, and the inclusion of additional provider information.

## ACKNOWLEDGMENT

This research was supported by IBM Centres for Advanced Studies (CAS) and the Natural Sciences and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network.

## REFERENCES

[1] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," in *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 328–336.

[2] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *ICA3PP (1)*, 2010, pp. 13–31.

[3] SAVI, "Strategic network for smart applications on virtual infrastructure (savi)," http://www.savinetwork.ca/, 2012, last access 31/10/2012.

[4] M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Navigating the cloud with a MAP," in *13th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, To Appear, 2013.

[5] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A cloud broker service," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012, pp. 891 –898.

[6] M. Smit, P. Pawluk, B. Simmons, and M. Litoiu, "A web service for cloud metadata," in *IEEE Congress on Services*, 2012, pp. 24–31.

[7] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A structured English query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. New York, NY, USA: ACM, 1974, pp. 249–264.

[8] A. Bozzon, M. Brambilla, S. Ceri, and P. Fraternali, "Liquid query: multi-domain exploratory search on the web," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 161–170.

[9] D. Konopnicki and O. Shmueli, "Information gathering in the world-wide web: the W3QL query language and the W3QS system," *ACM Trans. Database Syst.*, vol. 23, no. 4, pp. 369–410, Dec. 1998.

[10] E. Spertus and L. A. Stein, "Squeal: a structured query language for the web," *Computer Networks*, vol. 33, no. 16, pp. 95 – 103, 2000.

[11] G. O. Arocena, A. O. Mendelzon, and G. A. Mihaila, "Applications of a web query language," *Computer Networks and ISDN Systems*, vol. 29, no. 13, pp. 1305 – 1316, 1997.

[12] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *Grid Computing Environments Workshop, 2008. GCE'08*. IEEE, 2008, pp. 1–10.

[13] K. Zachos, J. Lockerbie, B. Hughes, and P. Matthews, "Towards a framework for describing cloud service characteristics for use by chief information officers," in *Requirements Engineering for Systems, Services and Systems-of-Systems (RESS)*, 2011, pp. 16 –23.

[14] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring of heterogeneous clouds using stream processing," *Future Generation Computer Systems*, To Appear, 2013. Preprint: http://dx.doi.org/10.1016/j.future.2013.01.009.