Containerized SQL Query Evaluation in a Cloud

Dr. Weining Zhang and David Holland
Department of Computer Science
The University of Texas at San Antonio
{Weining.Zhang, david.holland}@utsa.edu

***Abstract*—Recent advance in cloud computing and light-weight software container technology opens up opportunities to execute data intensive applications where data is located. Noticeably, current database services offered on cloud platforms have not fully utilized container technologies. In this paper we present an architecture of a cloud-based, relational database as a service (DBaaS) that can package and deploy a query evaluation plan using light-weight container technology and the underlying cloud storage system. We then focus on an example of how a select-join-project-order query can be containerized and deployed in ZeroCloud. Our preliminary experimental results confirm that a containerized query can achieve a high degree of elasticity and scalability, but effective mechanisms are needed to deal with data skew.**

***Index Terms*—Database, DBaaS, query evaluation, software container, cloud, OpenStack, ZeroVM**

I. Introduction

Recent years have witnessed fast growth of cloud computing. An increasing number of cloud platforms, such as Amazon AWS EC2, Google Compute Engine, and Microsoft Azure[1], are now available to users. A cloud platform provides a combination of services, including infrastructure (IaaS), platform (PaaS), and software (SaaS). These services run on an infrastructure of large numbers of commodity computers connected by high speed networks, delivering economy of scale, elasticity, efficiency, availability, and reliability.

To support data processing for cloud users, it is important to provide scalable, reliable, highly available and highly efficient database services (DBaaS) in cloud. Currently users have three options when it comes to use databases.

Cloud users currently have three DBaaS options to choose from. One) Run a traditional database management system (DBMS) inside a virtual machine (VM). The user must administer most system management activities, including software licensing, installation, configuration, management, backup, and recovery. This option is difficult to scale. Two) Use a NoSQL database, e.g. Apache HBase[1], Google BigQuery[2], and Cassandra[3]. These services are highly efficient and scalable for some types of large data, but the user is forced to deal with the lack of structured data and strong data integrity that is present in relational models. Three) Use a managed SQL database service, e.g. Amazon RDS[4], Google Cloud SQL[5], and MS Azure SQL[6]. User can rent conventional database servers running in the cloud. This is convenient because the DBaaS takes care all system maintenance and storage; existing applications can run without modification. However, while the user may scale-out by adding more servers, the execution of each query is still bounded by a single server and lack of parallelism.

Another noticeable recent technology advance is the rapid adoption of light-weight container technologies, such as Docker[7] and ZeroVM[8]. By running applications in containers, which are easily deployed and quickly instantiated, compute can be moved directly to the data instead of the other way around. Containers have smaller footprints, less start-up overhead, and secure isolation among tenants[9]. Containerized applications often have better performance because of parallelism. These features have prompted a strong push to integrate container technology into cloud platforms. However, the use of container technology in SQL DBaaS has not been reported in the literature.

In this paper, we present a study on using light-weight containers to evaluate relational query plans. We consider a DBaaS that provides the functions

[1]On-line at aws.amazon.com, appengine.google.com and azure.microsoft.com, respectively.

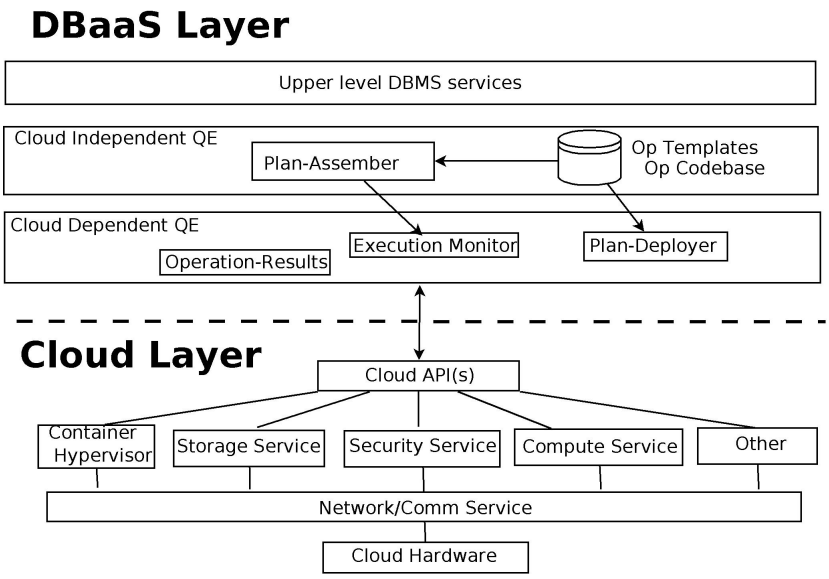


Figure 1: A Layered Architecture of a DBaaS in Cloud

of a traditional SQL database but run queries in containers inside a cloud storage system. Specifically, the DBaaS stores data using the cloud storage system, which automatically partitions, replicates, and distributes the data. When processing a query, the DBaaS first accepts an SQL query from a client and generates an optimized query evaluation plan in a traditional way. It then containerizes the query plan by identifying a network of compute nodes and assembling executable programs for the compute nodes to run inside containers. The containerized query plan is then deployed to the cloud storage system so that each container executes at the data storage node near its data when feasible. Intermediate results are pipelined into other containers. The execution of containers is load balanced and monitored by the cloud's job scheduler. The final result may be either returned directly to the client, or stored into the cloud storage system.

We focus on the containerization and deployment of a query evaluation plan in such a DBaaS. We present a layered system architecture and show by an example how a query can be containerized and deployed in the ZeroCloud[10]. Our preliminary experiments indicate that a containerized query has the potential of achieving scalability for big data sets. However, effective mechanisms are needed to deal with data skew.

The rest of the paper is organized as follows. In Section II, we present the layered architecture of the DBaaS. In Section III, we present a containerizable algorithm example. In section IV, we present a method to containerize a plan and deploy it into ZeroCloud. In Section V, we present experimental results obtained from running containerized join query plans on ZeroCloud. Finally, we briefly discuss related work in Section VI and conclude the paper in Section VII.

II. A Layered Architecture of a DBaaS

As shown in Figure 1, the DBaaS is a set of layers built on top of cloud services. The layers can be loosely coupled in the sense that upper layers use lower layers only through the provided service interface. Thus, different implementations of a service layer will not affect the function of an upper layer.

The Cloud Layer manages the cloud hardware, including compute nodes, storage nodes, and high speed networks, as well as a suite of other cloud services such as compute scheduling, storage replication, security access, networking, messaging, and container hypervisor services. Without loss of generality, we assume that this layer will provide high availability (by data replication), multi-tenant isolation, load-balancing, some consistency guarantee, security, and container management (including creation, scheduling, monitoring, disposal of contain-

ers). These services can be accessed by public cloud platform APIs.

The DBaaS Layer provides the functionality of relational database management, including the SQL query, optimization, transaction processing, and ACID consistency. We assume that the user data and system catalog are stored in the Cloud Layer using its storage service. The DBaaS Layer is itself divided into three sub-layers: a top sub-layer for higher level abstraction data management functions and two lower sub-layers for containerized query evaluation. The two lower sub-layers are: Cloud-Independent QE (Query Evaluation) and Cloud Dependent QE.

The top sub-layer receives user queries and generates optimized query evaluation plans. It also manages transaction processing and guarantees the ACID consistency. Optimized query plans represented in standard formats, e.g. an XML representation such as DXL[11], are passed to the Cloud-Independent QE for execution.

The Cloud Independent QE layer containerizes a given query plan by mapping plan operators into a network of cooperative compute nodes and assemble an executable program for each compute node using code from a library of relational operators. A repository of libraries is maintained by the DBaaS for different query operators.

The Cloud Dependent QE translates and packages the containerized query plan into an executable archive specific to the Cloud Layer consisting of programs, dependent libraries, and configuration meta data. The configuration meta data specifies deployment details such as number of containers, programs to be executed in containers, and communication topology among containers. The Cloud Dependent QE then deploys the package through the Cloud Layer's Container Hypervisor's service API.

III. Containerization of a Query Plan

In this paper, we say that a query plan is containerized if it consists of a network of compute nodes such that each compute node performs a subsequence of basic query operations of the plan, and can be executed inside a container, possibly in parallel or in pipeline with other compute nodes running in cooperating containers. The containerization of a query plan is to identify this network of compute nodes and assemble algorithms for these compute nodes.

There are several methods for the Plan-Assembler to generate a containerized query plan. For example, given a query plan, the Plan-Assembler may take each query operator of the plan as a compute node and assign a parallel algorithm to the compute node to explore intra-operator parallelism. It can then optimize the containerized plan by consolidating some adjacent compute nodes into a single compute node. Alternatively, the Plan-Assembler can assign compute nodes according to a set of translation rules that match a sub-plan with a specific pattern to a specific type of compute node. It can then assemble algorithm based on the type of the compute node. In either case, the Plan-Assembler must preserve the query plan's overall inter-operator execution order, but parallelize intra-operator whenever feasible. To do that, the Plan-Assembler needs to specify intra-operator as well as inter-operator network data flows. Once the algorithms for compute nodes are assembled, the program code for each compute node is then composed, compiled, linked, and packaged. At the run-time, the program codes for compute nodes will execute inside containers and the data flow among compute nodes will be realized by communications among containers.

A comprehensive treatment of methods to containerize an arbitrary query plan is beyond the scope of this paper. In the rest of this section, we present an example containerized query plan.

A. Network Topology of a SJPO Query

We consider the following simple Selection-Join-Projection-Ordering (SJPO) query:

$$\delta_{R.A_1,<}(\pi_{R.A_1,S.B_1}((\sigma_{R.A_2\leq a}R) \bowtie_{R.A_3=S.B_3} (\sigma_{S.B_2=b}S)))$$

where $R(A_1, A_2, A_3)$ and $S(B_1, B_2, B_3)$ are two relations, σ, $\bowtie$, π, and δ are selection, join, projection, and order-by operators, respectively. The major concepts presented in this example can be applied to other relational query plans as well.

Figure 2 shows the network of compute nodes (or simply node thereafter) for a containerized SJPO query plan. There are three types of nodes: ShuffleNode, JoinNode and MergeNode. Nodes of the same type form a tier. In Figure 2, the tiers are laid out from left to right. The algorithm for each type

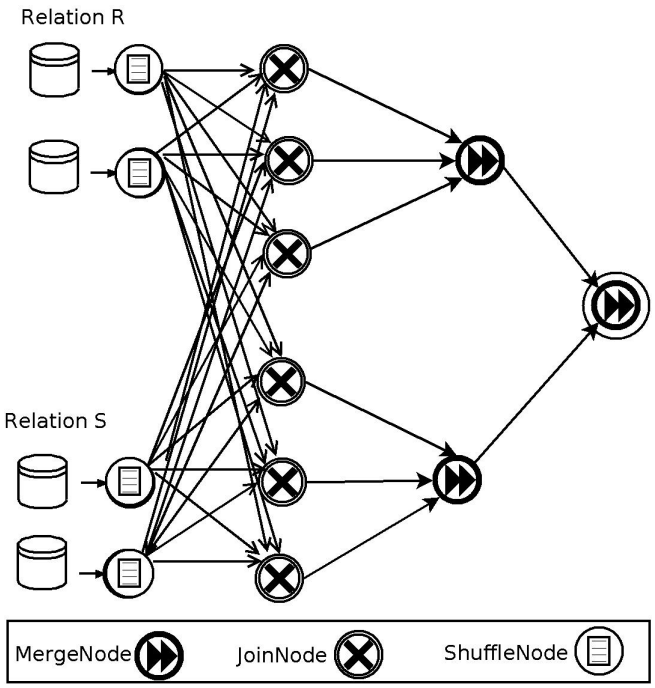


Figure 2: Network Topology of SJPO Query Plan

Algorithm 1 A ShuffleNode Algorithm

Input: T: data table,
θ: selection condition,
A: join attribute,
JN: a list of output streams
Method:
1. initiate scan of T, open streams in JN for write;
2. repeat for each tuple t scanned from T do
3. if select(t, θ) is true then
4. $i = hash(t.A)$;
5. transmit t through stream i in JN;

of nodes is assembled to perform a sequence basic query operators, such as file scan, selection, data exchange, join, projection, sorting, and merging.

B. Algorithms for Compute Nodes

In general, a node performs a sequence of query operators, such as data scan, selection, sorting, etc. The algorithm for a node can be obtained by chaining together the algorithms of the composed query operators.

As shown in Algorithm 1, each ShuffleNode performs a file scan in Step 2 that reads in one partition of a relation (either R or S), a selection in Step 3 that filters tuples based on the selection condition ($R.A_2 \leq a$ or $S.B_2 = b$), and finally a data exchange in Step 5 that sends each selected tuple across the network to a distinct JoinNode using a mapping (in Step 4) based upon the value of the tuple's join attributes (A_3 for R and B_3 for S).

Algorithm 2 A JoinNode Algorithm

Input: RN, SN: input streams
θ: join condition, L: attributes to project,
A: attributes to ordering, o: the required order,
m: output stream
Method:
1. open streams in {RN, SN} for read, m for write;
2. initialize hash table H, list U, and sorted list V;
3. while exists open input stream do
4. for each tuple s received from SN do
5. insert s into U;
6. for each tuple r received from RN do
7. insert r into H using join attribute of r;
8. for each tuple s in U do
9. probe H to find matching R-tuples;
10. for each <r, s> that satisfies θ,
11. $w =$ project <r, s> on L;
12. insert w into V in order o of $w.A$;
13. transmit V through m;

Algorithm 2 shows that a JoinNode performs a hash-join (Steps 2 to 10) followed by a projection (Step 11), and then a sort (Steps 12). The inputs RN and SN are lists of up-stream ShuffleNodes. The parameter m is the reference to a down-stream MergeNode, which merges the join output. The join is performed according to the match of join attribute values, using a hash-join with R-tuples for build (Step 7) and S-tuples for probe (Step 9). Before the first probe can be performed, it buffers S-tuples in a temporary list, U (Step 5). The join result, after projection and sorting, are sent to the downstream MergeNode in Step 13.

As shown in Algorithm 3, each MergeNode performs a merge on the tuples received from its input streams in Steps 3 to 5, and sends the results in Step 6, possibly to another down-stream MergeNode, depending on the depth of the merge tiers in the network. The MergeNodes may also return the final result directly to the client in sorted order or to the

Algorithm 3 A MergeNode Algorithm

Input: IN: a list of input streams,
A: attributes to base the sorting,
o: the sort order,
m: an output stream
Method:
1. open streams in IN for read and open m for write;
2. initialize a sorted list L;
3. while not all streams are closed do
4. for each tuple t received do
5. insert t into L in order o of $t.A$;
6. transmit L through m in sorted order;

cloud's storage system.

Once the algorithms for the compute nodes are assembled, the program codes that implement these algorithms will be composed, compiled, and linked on-the-fly. This can be achieved using pre-built fragments of program codes maintained by the Cloud Dependent QE layer. We emphasize that the context of these algorithms is limited to their container's boundaries. For example, the JoinNode cannot access schema meta-data for tuples outside the container's local file system.

IV. Deployment of Containerized Query

In this section we briefly describe an example of how a containerized query plan can be deployed using an open-source cloud service, ZeroCloud, a compute and storage platform that integrates ZeroVM, a light-weight container technology, into OpenStack's Swift object storage. We leave containerization of queries on other cloud and container platforms for future research.

A. The ZeroCloud Platform

ZeroVM is open-source and based on Google's Native Client (NaCl), a sand-boxing technology. It uses sand boxing to guarantee execution isolation. ZeroVM is light-weight in the sense that there is no virtualization. Compiled code is simply restricted to a trusted and limited set of system kernel calls. The runtime environment in a ZeroVM container includes a C99 or Python environment and an isolated in-memory local file system. All programs running in the containers are single-thread. Each program is built using the ZeroVM tool-chain and can only access external data using IP sockets, Unix pipes, and POSIX-like streaming files such as stdin, stdout and stderr.

ZeroCloud orchestrates the creation and deployment of ZeroVM containers. Programs are packaged into tar balls and deployed to ZeroCloud through its RESTful API. All packages contain a manifest/configuration file, and a set of program codes. The manifest is a JSON document that specifies the disposition of one or more groups of containers; their program codes and dependencies, inter-connect topology, shared environment variables, and other execution parameters. ZeroCloud collocates containers with their data on Swift storage nodes, sets up network channels among containers, starts the containers, and monitors their execution. Containers will stop and be discarded upon the completion of their programs.

B. Generation of a Configuration File

Although it is possible to generate a manifest file on-the-fly from scratch, we find it much easier to create it from predefined templates. In this approach, the DBaaS maintains a set of template manifest files for various types of query plans, such as SJPO. The templates specify types of needed containers with place holders for various parameters, such as numbers of each container type, its program codes, names of input/output files, and query-specific information such as selection conditions and join attributes. At the time of manifest generation, the Plan-Assembler in the QE layer retrieves one or more templates and parameterizes them based on query plan specifics.

C. Deployment of a Containerized Query Plan

To deploy a packaged containerized query plan for execution on ZeroCloud, the Plan-Deployer will issue a command to the ZeroCloud service API. For example, the following command executes a query packaged in a "query.tar" file.

```
$ curl -i -X POST
   -H "Content-Type: application/x-tar"
   -H "X-Auth-Token: $OS_AUTH_TOKEN"
   -H "X-Zerovm-Execute: 1.0"
   --data-binary @query.tar
  $OS_STORAGE_URL
```

where $OS_AUTH_TOKEN and $OS_STORAGE_URL are environment variables providing an authentication token and the link to the ZeroCloud service.

Internally, ZeroCloud authenticates the request, instantiates ZeroVM containers with specified program codes, places containers at storage nodes where the required data objects are located, sets up channels for each container, and executes the specified program codes inside containers. The program codes can then use the channels to access local data and to stream data through the network of containers.

V. Experiments and Results

We performed simple experiments to verify the feasibility and benefit of containerizing relational query plans. In this section, we report preliminary results of two experiments performed on a simple join query.

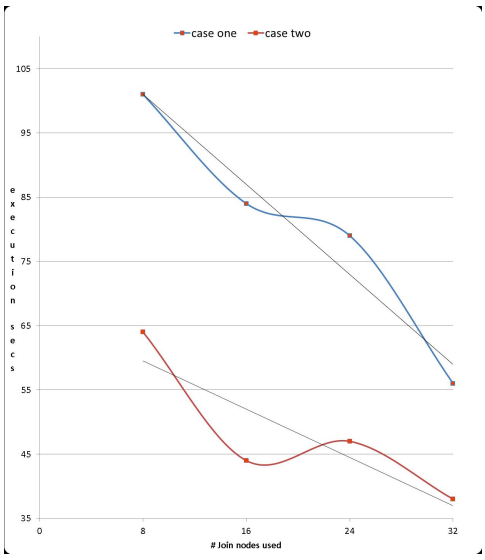
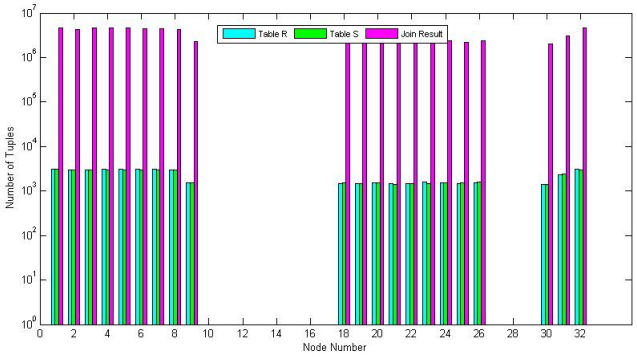
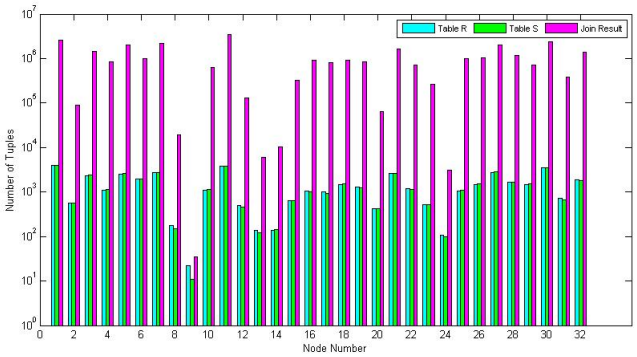


Figure 3: Execution time vs # of JoinNodes

The data-set used in our experiments is the Natality database downloaded from Google's BigQuery portal. A segment of the data-set containing 91,819 tuples is imported as a table. Each row has 32 attributes and is 96 bytes on average. The overall size of the table is 8 MB. For the experiments reported here, the table is partitioned randomly into four equal-sized partitions; two assigned as relation R and the remaining two assigned as relation S. Each partition is 2 MB in size and contains 22,955 tuples.



(a) Case One



(b) Case Two

Figure 4: # of tuples streamed to JoinNodes

The partitions are written to the Swift Object Store, which are automatically saved as three replicas.

ZeroCloud had some limitations. It enforced a 2-minute execution time limit (i.e., containers would be stopped and removed in 2 minutes regardless if the job is completed) and had difficulty executing more than 32 parallel containers for any test query. The containerized query of a simple join $R \bowtie_{R.A=S.A} S$ was executed with different choices of the join attribute, A. Execution times reported by a system monitor were recorded. Each query was repeated multiple times using different numbers of JoinNodes to investigate any performance gained by increasing the number of JoinNode containers working in parallel on the join.

Figure 3 shows execution time (in seconds) of the join for two different join attribute cases, each tested using 8, 16, 24, 32 JoinNodes. Case One uses a join attribute that has only 26 distinct values. Case Two uses a join attribute that has 439 distinct values. For both cases, the same hash function is used to map

tuples from ShuffleNodes to JoinNodes. The query plans, while varying the number of JoinNodes, holds the number of other container nodes static; a total of four ShuffleNodes (two for each relation) and one MergeNode for merging the join results.

As shown in Figure 3, the execution time decreases as the number of JoinNodes increases in both cases. Over the range 8, 16, 24, 32 of JoinNodes tested, Case One achieves an almost linear performance gain. The effect is less certain in Case Two, but still shows overall gain. It is interesting to note that the execution times of Case One is about twice as much as Case Two. Looking closer, we can see this is caused by the way the input tuples are mapped to the JoinNodes; some JoinNodes receive no tuples and are idle during the entire query. Figure 4 shows the distribution of tuples mapped to each of 32 JoinNode as well as the number of tuples joined. A logarithmic scale shows the mapping skew. For Case One, Figure 4a shows that using the standard Python hash function, six JoinNodes receives no tuples and 21 JoinNodes each receives between 1000 to 5000 tuples. Because the simple hash function has no special knowledge of any inherit join attribute skew in the data, six JoinNodes are never mapped. As a consequence these nodes have no work to perform. Since the execution time of the join operation is lower bonded by the execution time of the JoinNode with the heaviest workload, any performance benefit of adding more JoinNodes is lost when the number of JoinNodes exceeds the number of distinct join attribute values. On the other hand, Figure 4b shows that in Case Two, all 32 JoinNodes receive input tuples and have some work to do. Unlike in Case One where each JoinNode that participated in the join operation has to produce in the range of 2 to 6 million joins, JoinNodes in Case Two have much wider variations on workload: most JoinNodes produce less than 1 million joined tuples, but two JoinNodes produce as many as 6 million joined tuples.

Notice that the unbalanced workload is inherent in data skew and cannot be resolved by load balancing methods that simply move containers around. These results point to two important issues in design of join algorithms for further research. One issue is to design ways to use more than one JoinNode to efficiently join tuples that have the same join attribute value. This will allow a graceful scale-out of performance in Case One. The second issue is to design better mechanisms to assign balanced workloads to JoinNodes. These can include better hashing techniques or dynamical reassignment of join keys. This will improve the performance in Case Two.

VI. Related Work

Major cloud providers currently offer a number of SQL database services in the cloud. These include Amazon's RDS[4], Microsoft's Azure SQL Database[6] and Google's Cloud SQL[5]. In addition, a number of academic research groups also proposed cloud DBaaS to support relational database functionality. Examples include VoltDB[12], Postgres-XL[13], Impala [14], and Relational Cloud [15].

In addition to SQL DBaaS, a number of NoSQL database services [16], [3], [2] are also provided by major cloud providers. These systems provide efficient columnar storage and retrieval of large data sets, support nested data structure, and NoSQL query types. However, these systems do not support full SQL query and ACID data consistency. NoSQL data-stores such as Google's Bigtable[16], Apache Hadoop's HBase[17], or Facebook's Cassandra[3] are highly scalable, but their limited API and weaker consistency models complicate SQL application development.

Development of light-weight software container technology includes Docker[7] and ZeroVM[8]. Both are open source projects. Not much has been published about these techniques, except documents on the product websites and a number of blogs published by individual developers. Performance of container technology for data intensive applications has been investigated in[9], [18].

VII. Conclusions

In this paper we presented an layered service architecture for a cloud-based, relational database as a service (DBaaS) that can package and deploy a containerized query evaluation plan for execution using the underlying cloud storage system and light-weight container technology. We then focus on the tasks of containerization and deployment of a query evaluation plan and show by an example how a

query can be containerized and deployed for execution in ZeroCloud, an integrated platform based on OpenStack's Swift object storage and ZeroVM light-weight container technology. Our preliminary experimental results confirm that a containerized query evaluation can indeed achieve a high degree of elasticity and scalability, but effective mechanisms are needed to deal with data skew.

There are many interesting directions for future research. This paper investigated ideas that allow a SQL database service to be layered on top of cloud compute and storage services. One important aspect for further research is query optimization in such a service. It is not clear that the conventional cost model in relational DBMSs (based on page I/O) and distributed DBMSs (based on communication) are still appropriate in cloud environment. A principle goal of containerizing a query plan is parallelism. Also like in-memory databases, because distributed container memory scales, intermediate results can be held in memory between successive execution of waves of containers. Like parallel execution, in-memory execution is an architecture design for significant speed-up. Another interesting issue is the composition of algorithms for compute nodes in a containerized query plan. It should be feasible to build containerized queries from libraries of basic query operator codes that can be assembled, deployed and executed inside containers. The assembly and deployment processes use these libraries to compose program code for each type of compute nodes dynamically at plan evaluation and packaging time. Dividing a query into a network of compute nodes executed by cooperating containers is a database architecture model that matches commodity cloud architectures resources.

REFERENCES

[1] Apache HBase. http://hbase.apache.org/.

[2] Melnik, Gubarev, Long, Romer *et al.*, "Dremel: Interactive analysis of web-scale datasets," in *International Conference on Very Large Data Bases*, 2010, pp. 330–339.

[3] Lakshman and Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[4] Amazon RDS. http://aws.amazon.com/rds/.

[5] Google Cloud SQL. https://cloud.google.com/sql.

[6] Microsoft Azure SQL Database. http://azure.microsoft.com/en-us/services/sql-database/.

[7] Docker. https://www.docker.com/.

[8] ZeroVM. http://www.zerovm.org/.

[9] Tang, Zhang, Wang, and Wang, "Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds," in *International Conference on Algorithms and Architectures for Parallel Processing*, 2014, pp. 415–428.

[10] ZeroCloud. http://www.zerovm.org/zerocloud.html.

[11] Soliman, Antova, Raghavan, El-Helw *et al.*, "Orca: A modular query optimizer architecture for big data," in *ACM SIGMOD International Conference on Management of Data*, 2014, pp. 337–348.

[12] Stonebraker and Weisberg, "The VoltDB main memory DBMS," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pp. 21–27, 2013.

[13] Postgres-XL. http://www.postgres-xl.org/.

[14] Kornacker, Behm, Bittorf, Bobrovytsky *et al.*, "Impala: A modern, open-source SQL engine for Hadoop," in *Biennial Conference on Innovative Data Systems Research*, 2015.

[15] Curino, Jones, Popa, Malviya *et al.*, "Relational Cloud: A database service for the cloud," in *Biennial Conference on Innovative Data Systems Research*, 2011, pp. 235–240.

[16] Chang, Dean, Ghemawat, Hsieh *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[17] George, *HBase: The Definitive Guide*. O'Reilly Media, Inc., 2011.

[18] Rad, Lindberg, Prevost, Zhang *et al.*, "ZeroVM: Secure distributed processing for big data analytics," in *World Automation Congress*, 2014, pp. 882–887.