2019 IEEE 12th International Conference on Cloud Computing (CLOUD)

Distributing an SQL Query Over a Cluster of Containers

David Holland* and Weining Zhang†
Department of Computer Science, University of Texas at San Antonio
Email: *david.holland@utsa.edu, †Weining.Zhang@utsa.edu

***Abstract*—Emergent software container technology is now available on any cloud and opens up new opportunities to execute and scale data intensive applications wherever data is located. However, many traditional relational databases hosted on clouds have not scaled well. In this paper, a framework and deployment methodology to containerize relational SQL queries is presented, so that, a single SQL query can be scaled and executed by a network of cooperating containers, achieving intra-operator parallelism and other significant performance gains. Results of container prototype experiments are reported and compared to a real-world RDBMS baseline. Preliminary result on a research cloud shows up to 3-orders of magnitude performance gain for some queries when compared to running the same query on a single VM hosted RDBMS baseline.**

***Index Terms*—Software container, deployment, SQL database, query evaluation, intra-operator parallelism, cloud computing**

I. Introduction

Container technology has become universally available on all kinds of cloud platforms. Containers provide an OS-level virtualization [8] that provide applications with an isolated execution environment using cloud host resources. Running applications inside containers, rather than in virtual machines (VMs), on a cloud can provide many benefits: elasticity (fast start-up and quick tear-down), scalability, host independence, multi-tenant isolation, a smaller footprint and provides overall better economy(cloud consolidation).

While many new applications have been designed to run in containers, progress has been slow to adapt existing SQL relational database systems to take full advantage of container platforms. Recent published research on whether container technology can be used for scaling relational queries across many containers to achieve HPC level performance, concludes that containers are more elastic and scalable than VMs[7]. It is our contention that database queries can indeed be containerized and scaled, but containerization requires a special container deployment architecture to meet specific needs of SQL queries [20].

In this paper, the problems of containerizing SQL queries is considered. To our knowledge, this has not been reported in the literature. Specifically, a software framework to containerize and deploy an SQL query is presented, permitting a single SQL query to be executed by a network of cooperating containers. This framework is a general methodology that is cloud and container type agnostic. We build on established distributed query concepts, e.g. partitioning and intra-operator parallelism, (i.e. dividing execution of an operator concurrently across multiple processors and partitions of the original relation [18]), but investigate a new problem, i.e., how to distribute containers that collectively execute a SQL query across a cluster of containers in a cloud. A feasibility study of this with performance analysis is reported in this paper. A containerized query (henceforth CQ) uses a deployment methodology that is unique to each query with respect to the number of containers and their networked topology effecting data flows. Furthermore a CQ can be scaled at run-time by adding more intra-operators. In contrast, the traditional distributed database query deployment configurations do not change at run-time, i.e., they are static and applied to all queries. Additionally, traditional distributed databases often need to rewrite an SQL query to optimize performance. The proposed CQ methodology requires no such SQL-level query rewriting.

Containerizing the execution of an SQL query using multiple distributed containers raises several challenges: 1) how the query's relational operators should be distributed among many containers to obtain in-parallel performance gains; 2) how containers can improve query performance collectively by loading and keeping data relations all-in-memory, including any intermediate results throughout the entire query's evaluation using container share-nothing memory, which provides further query evaluation speed-up by eliminating DBMS disk buffer management overhead of large relations that overflow memory [12]; 3) a CQ must also orchestrate scheduling of multiple cooperating containers across many cloud hosts, while carefully synchronizing their interactions during the query evaluation. These issues are addressed in this paper. Contributions in this paper are as follows:

1) A framework (methodology and container architecture) for a CQ, which for any given SQL query, creates an execution plan for a unique network of specialized cooperating containers, and deploys the execution plan using a container technology specific platform, e.g. Docker, in a cloud.
2) An implementation of a research prototype, including a CQ plan compiler to systematically create CQs.
3) A set of experiments on a research cloud comparing CQ performance metrics against a real-world RDBMS baseline to validate the framework. Preliminary result shows up to 3-orders of magnitude performance gain for containerization over running the same query on the standalone RDBMS baseline hosted on a OpenStack research cloud VM.

The rest of the paper is organized as follows. In Section II a CQ framework is presented. Section III describes an implementation of the framework. In Section IV results from a set of test-bed experiments are discussed. Section V discusses

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00079

important related work, and Section VI concludes with a summary and discussion of future work.

II. A Framework for Query Containerization

For the purpose of this paper, we define an SQL CQ as fully planned when all relational operators in the query tree have been mapped into a cluster of cooperating networked containers that can evaluate the query.

Fig. 1 shows the work-flow for a CQ framework. (A) A SQL statement is compiled by a relational query optimizer and output as an optimized query tree. (B) The query tree is then used as input to a mapping function to construct a digraph, representing abstractly a set of specialized networked containers that will evaluate the query. (C) The digraph is further compiled into an evaluation and deployment plan, conveying special container properties, e.g. placement, network configuration, container image type, parameters etc. The plan is output as a data serialization language(DSL) formatted file, e.g. YAML. (D) The DSL file is used as a command stream by the underlying container platform's orchestrations tools to allocate and schedule containers that evaluate the query.

Fig. 2 shows a scenario evaluating a CQ across a cluster of containers hosted across many distinct cloud VMs. Shown are three types of containers: Master, Worker, and Data-Only. Each query is progressed and orchestrated at run-time by a master. The master starts and executes steps (B) and (C) in Fig. 1. The framework work-flow is repeated for every distinct query to create a new digraph, plan and serialized DSL file.

Subsequently, after all containers are started, the master progresses query evaluation and data-flows with control messages to-and-from worker containers. Query relational operations are executed by the cooperating worker containers; each performs a specific query operation, e.g. one of $\{\bowtie, \pi, \sigma, \delta\}$, as well as operations needed to transfer data among workers. Data-Only containers serve workers as a uniform interface for storage and retrieval of data from the underlying cloud storage. All containers are networked together by a virtual private overlay LAN that spans the container cluster's VMs.

III. A Framework Prototype

We now describe one (of many possible) experimental research prototype implementation of the CQ framework using Docker containers, which we use to illustrate and validate the framework. For simplicity, we use off-the-shelf software packages to implement query system functions. Specifically, SQLite, a lightweight SQL engine, is embedded inside each Worker container image to execute relational operators. RabbitMQ, an external messaging service, is used to facilitate message communication among master and worker containers. We use Java to implement plan generation, serialization and query run-time container management functions. The Java Database Connectivity (JDBC) API interfaces containers with the relational operator functions provided by SQLite. This implementation strategy provides a plug-and-play interface for our containers to interact with any off-the-shelf lightweight SQL engine that supports a JDBC driver. The following subsections detail specific issues related to the prototype.

A. Container Images

We build a different image for each of the three types of the containers, namely, Worker, Master, and Data-Only, to run container-specific Java code. These container images are built using a software layer model. The base layer consists of an Ubuntu 14.04 Linux operating system. Subsequent upper layers in an image's software stack include a Java JVM and Java programs (for both Master and Worker container functions), and the RabbitMQ message interface API binaries. The message layer permits run-time control messages to-and-from master and workers to monitor, synchronize and progress the query. The Master uses message queues to communicate tasks to other Worker or Data-only containers (to be further discussed in Section III-G). The Master sends control messages to workers to synchronize the progress of the overall query and to direct data flow between workers, which contains intermediate results (tuples) produced by relational operators. The Master image includes a custom written query plan compiler, which interfaces to Docker container platform orchestration tools: Docker-Compose and Docker-Swarm. Worker functions delegate relational operations to the embedded SQLite engine, but manage intermediate result data flows using special non-relational operators described in Section III-C. The Data-Only image uses the same base layer, but contains only functions to manage data and provide a uniform interface for Workers to access relation data sets.

B. Plan Generation

Fig. 1 illustrates the CQ plan generation process. It begins with an optimized query tree obtained in Step (A). In step (B), the optimized query tree is transformed by a 1-to-1 mapping into a directed graph that represents container execution order and direction of data flows. Specifically, each query tree operator is mapped into a unique digraph vertex abstractly representing a worker. Each worker executes a specific query relational operator, with conditions, predicates and clauses derived from its mapped query tree node. A pre-order traversal of the query tree suffices to map each node into the directed graph.

Data relations must also be mapped to data-only containers. A list of database relations and their partitions, if any, must be available to the query planner. If the relation is partitioned, each partition is managed by a distinct data-only container. This information is usually available from a data catalog. Assigning data-only containers to a relation's data partitions is an additional organization step that proceeds query evaluation. Containerizing the data requires identifying the data's location, schema and permissions. Once the partition size and host data location is known, containerization scheduling also attempts to collocate data-only containers on the same host where the database volume resides.

Planner compiled output is formatted as a version 3.3 Docker-Compose YAML file. The planner feeds the compiled

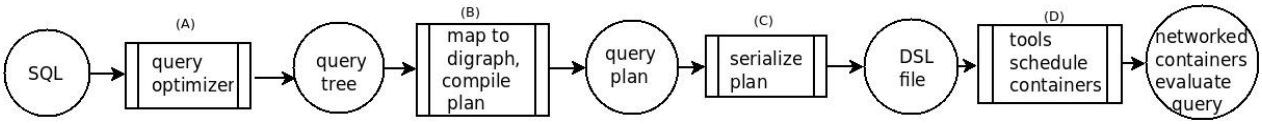

Figure 1. Query Containerization Framework Work-Flow

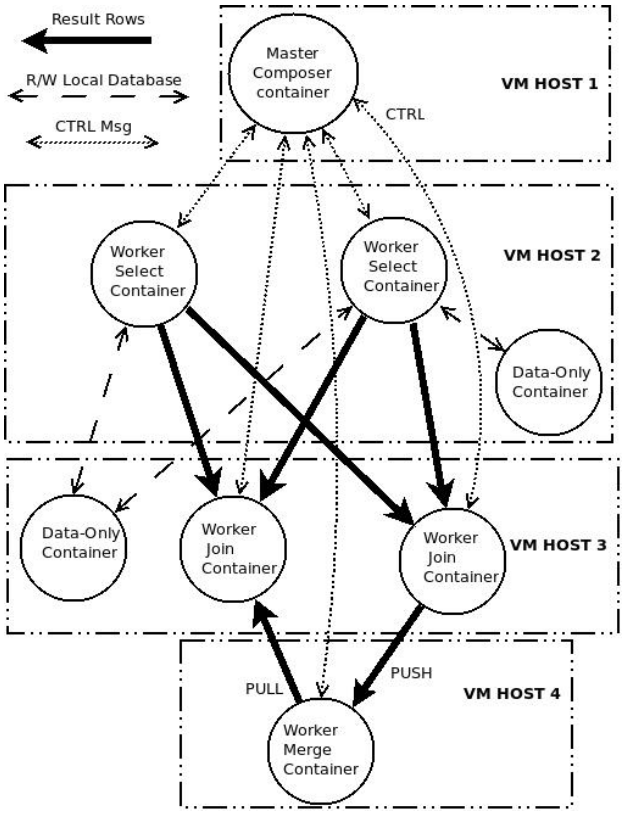

Figure 2. Containerized Query across many VMs

YAML file to a container orchestration tool, Docker-Stack, that deploys the containers across a cluster of VMs. Each worker container executes its distinct relational operator on its data using the embedded SQL engine. A simple SQL SELECT command with appropriate predicate and clauses e.g. *"SELECT t.a, t.b FROM intermediateTable AS t"* suffices to execute the operator, in this example a projection operator π. Data-Only containers are concerned with marshaling a query's initial database relations for workers. JDBC intermediate rows sets are transferred between containers along shared directed edges as specified in the query's digraph using streaming sockets. Sockets implement special data transfer operators described in section III-C.

C. Data Flow Management Operators

In addition to the query's relational operators executed by containers, four non-relational operators are used to transfer intermediate results among containers.

- Pull: is used by the down-stream worker to retrieve relation or intermediate data from an up-stream worker.
- Push: is used by the up-stream worker to send relation or intermediate data to a down-stream worker.
- Exchange: redistribute tuples between containers, using One-To-Many or Many-to-One distribution patterns.
- Merge: collates down-stream results of up-stream parallel operations.

Whether a Push or a Pull is used to transfer data between two workers will depend on the relational operators the workers are assigned to execute. For example, as shown in Fig. 2 a worker executing a join may Pull tuples from one worker that produces tuples, while allowing another worker to Push its tuples.

The Exchange operator is introduced to further organize how Push and Pull are used to transfer intermediate results. This operator redistributes output data tuples from up-stream workers or data-only containers to multiple down-stream workers. Exchanging involves selecting a distribution pattern, e.g exchanging by hash key, equal-size block re-distribution or round-robin among recipients etc.

For example, in Fig. 3 (C) an Exchange operator is used to distribute tuples from a data-only container managing relation R to the three workers that perform a join operation in parallel, possibly using a join key hash mapping from R to the distinct join containers. In that case the distribution pattern of the tuples is based upon the hash values of join attribute $R.B3$. As alluded to, the Exchange is a general redistribution operator, not always a hashing function.

D. Adding Intra-Operator Parallelism Statically

In Fig.1 step (B), the initial digraph may need to be revised by adding more vertices and edges whenever a vertex's operation needs to be distributed across many intra-operator vertices in parallel. The number of parallel nodes for a relational operators is referred to as the degree of parallelism (DoP) of the operator[18]. An operator's DoP is assigned by the planner. As parallel operators are added, Exchange operators III-C are also added to redistribute data to and from the new vertices.

Modifying the graph by adding additional parallel vertices is performed based on Algorithm 1, which takes as parameters: (G the query's digraph, v the current digraph vertex and d the DoP). Two helper sets are needed: 1) a set In_v of vertices for all input edges of v, 2) a set Out_v of vertices for all output edges of v. Steps 1-5 iterate $(d-1)$ times to add $d-1$ new intra-operator vertices to augment the original vertex v. In each iteration Step 2 creates a new vertex v' by duplicating v. Steps 3 and 4 add directed edges to v' based

Algorithm 1 Adding Parallel Vertices

Input:
$G =< V, E >$: a digraph view of containers
v: current digraph vertex
d: operator DoP
Output:
$G' =< V', E' >$: updated digraph
Method:
1. For $(d - 1)$ iterations Do
2. create new graph vertex v'
3. add inEdges to E $\{(u, v')$: $\forall(u) \in In_v\}$ using ψ_1
4. add outEdges to E $\{(v', u)$: $\forall(u) \in Out_v\}$ using ψ_2
5. add v' to V
6. return $G' =< V', E' >$

upon the same directed edges from v using vertex sets In_v and Out_v. Exchange functions ψ_1 and ψ_2 are added to v'. Exchange functions (see Section III-C) may create new edges not common to v, depending on how the data is redistributed by and to the new intra-operator vertices. Step 5 adds the new vertex v' into V, the digraph's modified list of vertices. Step 6 returns the updated digraph G' with an updated list of intra-op vertices and edges V' and E'.

Fig. 3 (B) and (C) show an example that transforms the original join operator into intra-operators with a combined DoP 3. In this case, because the operator is an equi-join, the Exchange operator may choose to implement a hashing function based upon the join key that maps a tuple to one of the three intra-operator containers.

E. Serialization

Next, the digraph representing the completed CQ plan is serialized by the planner into a configuration DSL (YAML) file. An example configuration file is shown in Fig. 3 (D). This file communicates any semantics and relationships between containers executing the query and other configuration input needed to deploy and orchestrate the query. The file is composed of a hierarchical list of container properties for each container in the digraph, reflecting query semantics, such as, the types of containers, their inter-relationships, cluster placement preferences, environment variables, memory size, volumes, network identifiers, and other artifacts needed to execute the overall query across a cluster of containers.

F. Plan Deployment

Query evaluation is started by the master by sending the YAML configuration file as input to the platform's container orchestration tool, Docker-Compose. The orchestrator tool then schedules containers described in the configuration file assisted by the prototype's cluster manager, Docker-Swarm. Container server/daemons on each VM respond to the orchestrator to start container execution. In addition, orchestration also dynamically creates a virtual network of data communication channels among all containers. The private networked messaging service, RabbitMQ, is also started at deployment to enable subsequent communications between the master and worker containers during query evaluation.

G. Synchronization

Once containers are deployed the master orchestrates the evaluation and progress of the query using control messages. Evaluation task control messages are passed from master to other containers with instructions to progress the query, e.g. specific relational operator instructions. The master makes use of its knowledge of the digraph using edge direction to synchronize both container execution order and operator result data flow between containers. Fig. 2 shows a scenario in which the master orchestrates the evaluation of the query with control messages. The master progresses the query by sending control messages to a down-stream worker to start execution of its relational operation, e.g. a join $\bowtie$. The query plan instructs the master what type of control message and instruction payload to send to each worker. Each worker then executes its control message instructions.

In general, the master can be replicated for fault tolerance. Workers are delegated responsibility to handle autonomously some inter-container data transfers between their peers to help distribute overall control and avoid bottlenecks at the master. For example, a worker may sends a message to an up-stream or down-stream worker to request pulling or pushing its result data.

Workers provide their adjacent peers and master with their current state information. When an up-stream worker completes transferring data to a down-stream worker, both containers notify the master of any data state content changes. To aid specifying worker synchronization, the following container data states are defined:

- Valid Input State (VIS). A container is in VIS when its input satisfies conditions to start execution of its designated relational operator.
- Valid Output State (VOS). A container is in VOS when intermediate results can be sent to their down-stream workers.

With data state transitions occurring in the workers, execution progress of the overall query can be viewed as data flow engine, i.e. the execution of query operations proceed only along directed edges in the digraph to successive containers coincident with the progress of operator result data flow states. Until a container's input data flow state (VIS) is valid, the query cannot progress along an adjacent directed edge. This helps to simplify control of the evaluation. Each container tracks its data input state and reports state transitions to the master. This allows the master to monitor the overall state of the query's progress. For example, in the case of a hash-join, the *build phase* data flows are completed before the *probe phase* flows begin. In the prototype, a hash-join container's *probe* relation input state (VIS) remains invalid until its *build* state input data flows are also completed.

To communicate a container's valid state transitions, the message sent by the worker contain various fields depending on the message type: [timestamp, containerID, inputstate,

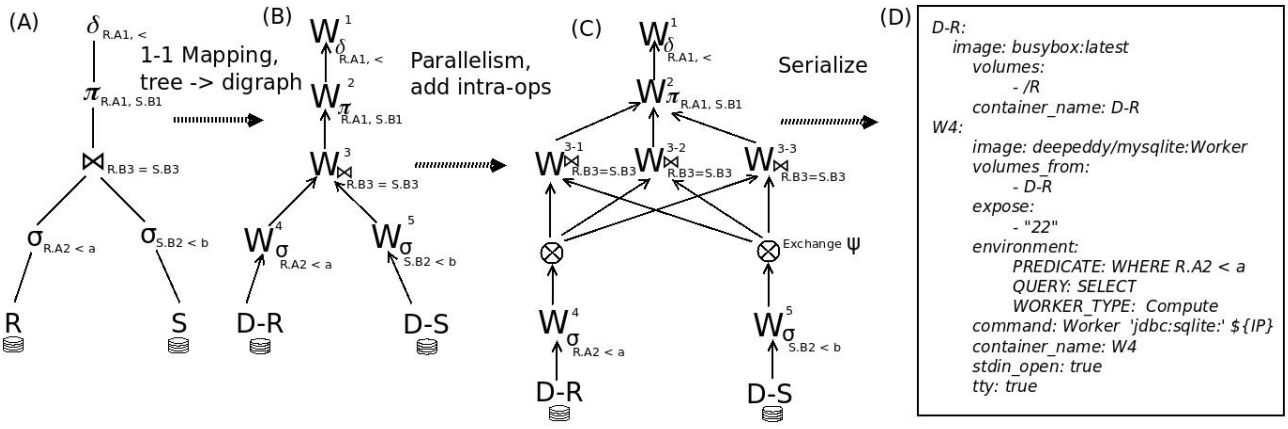

Figure 3. An Example of Query Containerization.(A) query tree; (B) mapped 1-1 into digraph; (C) add parallel intra-operators into digraph; (D) compile serialized deployment file.

outputstate, operatorAction, adjacenyList]. Because workers can help manage their data state progress autonomously, the master can focus on progressing the overall query with control messages; handling state transitions exceptions or failures. Failures include a container crash or undelivered (or lost) state transition message.

H. An Extended Example of Query Containerization Plan Generation

Fig. 3 illustrates the work-flow of containerizing an example SelectJoinProjectOrder(SJPO) SQL statement denoted in relational expression form: $\delta_{R.A1,<}(\pi_{R.A1,S.B1}((\sigma_{R.A2\leq a}R) \bowtie_{R.A3=S.B3} (\sigma_{S.B2=b}S)))$, where $R(A1, A2, A3)$ and $S(B1, B2, B3)$ are two relations; $\{\bowtie, \pi, \sigma, \delta\}$ are join, projection, selection, order-by operators, respectively. The query tree in (A) represents the optimized query plan that would be generated by a conventional query optimizer. Query tree nodes are mapped 1-1 into a directed graph (B), whose vertices represent worker containers that execute the query's relational operators. Digraph edge direction represents both operator execution order and data flow. The planner chooses to further modify the digraph by dividing a relational operator (here, the original join) among three intra-operator vertices (for a DoP of 3) that execute in parallel (C). Parallelism is a principle performance benefit of CQ evaluation. Also, shown in Fig. 3 (C) is the Exchange operator (Section III-C) that redistributes input to multiple containers; in this case redistributing tuples from data relations to parallel intra-operator join workers. Finally, the digraph is compiled into a YAML (DSL) file that specifies execution semantics for a container specific platform orchestration tool (D), e.g. network id, data volumes, placement directives, container image type etc.

IV. Experimental Results

A. Experiment Setups

This section describe prototype experiments and discuss preliminary results. To ensure experiment transparency and repeat-ability, a sufficiently large public relational database [1] was transcribed from MySQL to a SQLite database, which serves as test-bed data relations for comparing different CQ configurations. Two relations from the database, *salaries* and *employees*, are used in the queries: *salaries* with $2,844,047$ tuples, *employees* $300,024$ tuples. Both relation's tuples were randomly permuted before experiments to avoid any skew in the data. The experiments use several types of queries: SJPO, Aggregate and Simple Select, Fig. 4. Operator predicate clauses were varied for different effects in each experiment.

1	SELECT e.empNo, s.salary FROM salaries AS s, employees AS e WHERE (CAST(julianday(s.toDate) - julianday(e.hireDate) AS INT) < 730 AND (s.empNo == e.empNo) AND (s.salary < 200000) AND (e.empNo < 50000) ORDER BY e.empNo DESC; **[*Query* One: *SJPO*]**
2	SELECT e.empNo, s.salary FROM salaries AS s, employees AS e WHERE (CAST(julianday(s.toDate) - julianday(e.hireDate) AS INT) BETWEEN (1 AND 7) AND (s.salary < 40000) AND (e.empNo <30000) ORDER BY e.empNo DESC; **[Query Two: *SJPO*]**
3	SELECT salary, emp_no FROM salaries WHERE salaries < 200000; **[Query Three: *Simple Select*]**
4	SELECT SUM(salary - emp_no) FROM salaries WHERE salary < 200000; **[Query Four: *Aggregate*]**

Figure 4. Table of Experiment Queries

Each experiment records and compares a query's performance when 1) executed as a cloud hosted CQ vs. 2) executed on a VM hosted standalone SQLite baseline. Each experiment is repeated multiple times with operators assigned different DoPs to investigate how parallelism effects per-

[1] https://dev.mysql.com/doc/employee/en/sakila-structure.html

# Partitions	⋈ DoP	δ DoP	π DoP	σ DoP
6	6	1	1	6
12	12	2	2	12
24	24	4	4	24
48	48	8	8	48
SQLite Baseline	1	1	1	1

Figure 5. Experiment Partitions & Intra-Operator DoPs

formance. The *salaries* relation for each query experiment is run with different number of partitions, matching intra-operator DoP for some operators, Fig. 5, to observe effects of parallelism. All relational operators $\{\bowtie, \pi, \sigma, \delta\}$ are configured with zero or more parallel intra-operators. DoP for intra-operators $\{\bowtie, \sigma\}$ match the experiment's number of partitions for relation *salaries*. DoP for $\{\delta, \pi\}$ also increase with number of partitions to distribute compute overhead needed when merging intermediate results from parallel intra-operator vertices lower(upstream) in the digraph. This permits observing speed-up in the experiments due to intra-operator and disk parallelism, and further effects performance by loading and keeping relations all-in-memory in each container's embedded SQLite engine. JDBC's database driver specification: *<DB_URL>:memory:* supports an all-in-memory configuration. Relations and intermediate results are only constrained by a container's memory size, which in turn is constrained by the VM host memory; no SQLite memory overflow errors were observed. This avoids costly performance degrading due to heap buffer paging overhead for large relations [19]. More to the point, the Docker workers written in Java are initially started with 4 Gigs of Heap memory (distinct from SQlite memory). Java garbage collection has a significant impact on worker performance, when large JDBC ResultSets generated by the SQL engine operations deplete container JVM heap memory. DoP and number of relation partitions is summarized for the SJPO experiments. Queries 3 and 4 also matched their principle relational operator DoP with the same number of partitions.

The experiments were performed using Docker containers scheduled across a Docker Swarm cluster of six OpenStack Ubuntu 14.04 VMs hosted on a research cloud. The experiments can be conveniently repeated by cloning the experiment's BitBucket repo using the following shell command: *git clone https://deepeddy@bitbucket.org/deepeddy/databases.git*. A README file details suggested VM configurations on any cloud and step-by-step instructions for experiment set-up and test query execution. Each experiment configuration and the baseline were run at the same time to avoid the impact of different multi-tenant cloud workloads; network rates also fluctuated depending on workloads. However, the relative performance between the baseline and the Containerized Query is reported regardless of multi-tenant workload. A CQ planner/compiler was coded for the prototype to enable the Master to organize the deployment and evaluation of each query. Special optimizations were avoided by the Query Planner to simplify analysis and emphasize the essential CQ methodology. Java executables were instrumented to record operator performance metrics . Total query execution time for each experiment is reported and compared.

B. Query Experiment One

The purpose of experiment-one Fig. 6 (see query-1, Fig. 4) is to compare the evaluation of a SJPO CQ to the same query evaluated on the SQLite baseline, when indexing and sorting optimizations are allowed by the baseline's query plan. Results show that a configuration of 6 partitions and DoP 6 for $\bowtie$, results in a only a slightly smaller query execution time compared to the SQLite baseline running the query on unpartitioned relations. The associated table shows slightly improving performance up until 48 partitions, an apparent inflection point, where network and container scheduling overhead begins reducing performance. Experiment-one includes a lengthy join clause, but its trailing segment *"AND (salaries.empNo == employees.empNo)"*, permits the SQLite baseline engine's query plan to optimize the $\bowtie$ by first sorting, then indexing on field *empNo* in both relations being joined. This optimization avoids a full-scan of both relations by the control baseline, resulting in only marginal speed-up in the CQ over the baseline.

C. Query Experiment Two

The purpose of experiment-two Fig. [7] (see query-2, Fig. 4) is to compare a SJPO CQ to the baseline version, when indexing and sorting optimizations are prohibited, forcing full-scans by the baseline. Join selectivity is also intentionally chosen to be poor by adding a BETWEEN clause. The clause *(salaries.empNo == employees.empNo)* in query one is now absent from the join, forcing a full-scan by the baseline SQLite engine. This effectively creates the same overhead as a full Cross-Join, and demonstrates the advantage of the distributed Containerized Query in cases of large relations with intense relational operator overhead. Experiment-two shows a 3-order of magnitude improvement over the baseline, 8 secs for 24 partitions as compared to 1538 secs for the baseline. In this case demonstrably, real-time OLTP is possible with a CQ. Again, Experiment-two again shows a performance inflection point at 48 partitions, where inter-container communication and container scheduling overhead begins degrading overall query completion time.

D. Query Experiment Three

The purpose of experiment-three Fig. 8 (see query-3, Fig. 4) is to compare a simple Select CQ to the baseline evaluation. Results shows a order of magnitude improvement over the baseline, 4 secs for DoP 48 as compared to 69 secs for the baseline. In this case demonstrably again real-time OLTP is improved by query containerization.

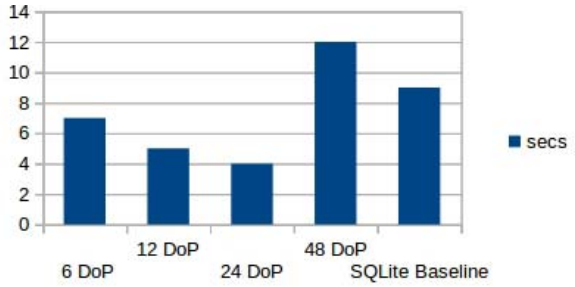

Figure 6. Experiment One: SJPO with indexing

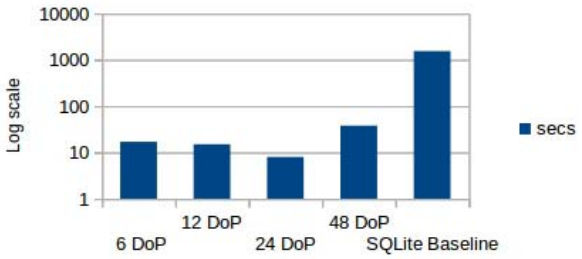

Figure 7. Query Experiment Two: SJPO with full scan

E. Query Experiment Four

The purpose of experiment-four Fig. 9 (see query-4, Fig. 4) is to compare an aggregate CQ to the baseline evaluation. Results show only marginal improvement over the baseline. An inflection point occurs again at DoP 48, when container scheduling and container-to-container network communication overhead begin to degrade performance. Clearly, containerizing a query in not always an imperative.

F. Experiment Five: Transfer & Re-Insert Latencies

The purpose of experiment-five Fig. 10 is to observe three critical operation times: 1) data ResultSet transfer time between adjacent digraph workers, 2) re-Insert/reloading latencies of intermediate results sets, and 3) relational operator execution times. An SJPO query run with 6 partitions and 6 DoP suffices to reveal latencies transferring and re-Inserting intermediate results between worker containers. The chart shows that these latencies are a factor of 7 times the average relational

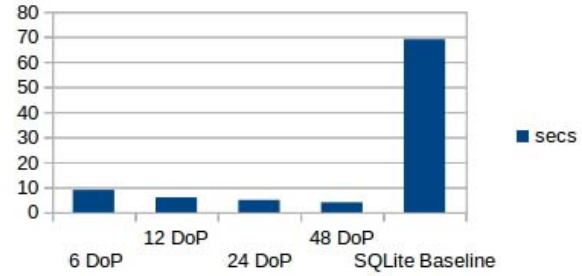

Figure 8. Query Experiment Three: Simple Select

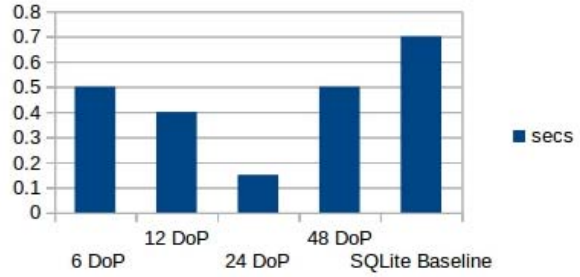

Figure 9. Query Experiment Four: Aggregate (Sum)

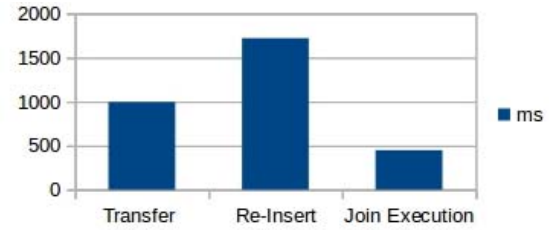

Figure 10. Experiment Five: Transfer & Re-Insert Latencies

operator execution time. This poses a significant bottle-neck for containerizing a query. Re-Inserting is an artifact to the prototype's design implementation; using an embedded SQL engine. An implementation that does not rely on an embedded SQL engine, but executes relational operators directly would eliminate database reloading. Re-Insert database latencies observed could be mitigated by the development of an API that supports direct native intermediate result set reloading. Extending the JDBC to accommodate transfer of rows sets using native internal data structures(ResultSetInternal) at the driver level that can be reloaded directly into a SQL database engine avoid the overhead of SQL Inserts and Journaling. Transfer latency of intermediate results between nodes is unavoidable with distributed containers and can best be addressed by extending low-level container kernel support for inter-container communication, instead of relying on application level channels, e.g. sockets.

V. Related Work

Major cloud providers currently offer a number of SQL database services in the cloud, although they do not support distributed queries using containers. These include Amazon's RDS[1], Microsoft's Azure SQL Database[4] and Google Cloud SQL[3]. In addition, a number of academic research groups have proposed cloud DBaaS services to support distributed relational database functionality in the cloud. Examples include VoltDB[19], Postgres-XL[5], Impala [11], and Relational Cloud [9]. A State-of-the-art overview of traditional distributed database architectures in the cloud is reviewed in [16] with a taxonomic overview of partitioning, distributed control and consistency.

In addition to SQL DBaaS, a number of NoSQL database services are also provided by major cloud providers. These

systems provide features like columnar storage and retrieval of large data sets, support nested data structures, and NoSQL query types. These services are generally, like CQs, distributed. However, these systems do not support full SQL queries and ACID data consistency. NoSQL data-stores such as Google's Bigtable[10], Apache Hadoop's HBase[15], Facebook's Cassandra[17] or PrestoDB are instructive for their distributive architectures, high scalability, handling of large data sets and cloud deployments.

Development of light-weight software container technology includes Docker[2], Kubernetes, LCX, and ZeroVM[6]. All are open source projects. Performance of container technology for data intensive applications has been investigated and reported in [14, 13]].

VI. Conclusions

This paper presented a method and software architecture framework for containerizing a relational SQL query, evaluated using cloud-based commodity hosts. The method describes a query planner that compiles the CQ plan. CQ runtime deployment, elasticity and scalability is different from traditional distributed databases configurations. Constructing a CQ plan starts with mapping the query's optimized relational tree into a digraph, which abstractly represents a cluster of cooperating containers that evaluate the query. Some digraph vertex operators may be subsequently distributed across many parallel containers(intra-operators) to increase evaluation performance. This allows any query's relation to be partitioned and processed in parallel. The final evaluation plan is compiled into a data serialization language, e.g. YAML, specific to a container platform technology tool set needed to deploy and execute containers on cloud hosts. Prototype experiments show that for some queries, performance can be greatly improved. Other queries benefit only marginally from containerization. This suggests that an online cloud RDBMS need only containerize queries that benefit from parallelism.

But there are still important future research solutions needed: 1) lower latency distribution of intermediate query results, 2) efficient, optimal run-time parallel intra-operator creation across many containers to progress queries, to include dynamic partitioning, 3) better optimizations including container placement, restart and fail-over, 4) a cost model for comparing Containerized Queries to both non-distributed and distributed baselines, emphasizing container elasticity, portability, and application software development costs. 5) CQ performance needs to be analyzed and compared against a distributed RDBMS baseline.

Acknowledgement

The authors would like to thank Texas Advanced Computing Center at UT Austin for their technical and resource support.

References

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

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

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

[4] MicrosoftAzureSQLDatabase. http://azure.microsoft.com/en-us/services/sql-database/.

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

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

[7] T. Adufu, J. Choi, and Y. Kim. Is container-based technology a winner for high performance scientific applications? In *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, pages 507–510, Aug 2015.

[8] David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[9] Carlo Curino et. al. Relational Cloud: A database service for the cloud. In *Biennial Conference on Innovative Data Systems Research*, pages 235–240, 2011.

[10] Chang Fay et. al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008.

[11] Marcel Kornacker et. al. Impala: A modern, open-source SQL engine for Hadoop. In *Biennial Conference on Innovative Data Systems Research*, 2015.

[12] Michael Stonebraker et. al. OLTP through the looking glass, and what we found there. In *Sigmod*, page 981, 2008.

[13] Paul Rad et. al. ZeroVM: Secure distributed processing for big data analytics. In *World Automation Congress*, pages 882–887, 2014.

[14] Xuehai Tang et. al. Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 415–428, 2014.

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

[16] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *ACM SIGMOD International Conference on Management of Data*, pages 579–590, 2010.

[17] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[18] DeWitt D. J. Mehta M. Managing intra-operator parallelism in parallel database systems. *IBM internal report, White Paper*, 1994.

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

[20] Weining Zhang and David Holland. Containerized SQL query evaluation in a cloud. In *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 1010–1017, 2015.