2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)

Using Containers to Execute SQL Queries in a Cloud

David Holland and Weining Zhang

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

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

***Index Terms*—Containers, SQL database, query evaluation, operator parallelism, cloud computing**

I. Introduction

Container technology, such as Docker [1], has become universally available on all kinds of cloud platforms. While many new applications have been designed to run in containers, progress as been slow to adapt existing SQL database systems to take full advantage of container platforms.

This paper considers the problem of containerizing SQL queries [3]. A framework to containerize an SQL query is presented, so that a single SQL query is evaluated over a distributed network of cooperating containers with a high degree of operator parallelism (DOP). We also present preliminary results of our experiments.

The use of many containers to evaluate a single query also allows distributed containers to load data into and keep intermediate results all-in-memory during the query's life time. This provides a significant speed-up by eliminating DBMS heap buffer management of the data [2].

II. A Framework for Query Containerization

We define "query containerization" as a process that maps the relational operators of an SQL query plan into a cluster of cooperating networked containers that collectively evaluate the query's plan.

Fig. 1 shows a work-flow for our query containerization framework process. (A) The SQL statement is first compiled by a relational query optimizer into an optimized query tree. (B) The query tree is then used as input to construct a digraph, representing a set of specialized networked containers. (C) The digraph is further compiled into an execution plan, which is output as a data serialization language (DSL) formatted file. (D) The DSL file is used by the underlying container platform's orchestrations tools to allocate and schedule containers that evaluate the query.

Additionally, to speed-up query evaluation, three non-relational operators are defined to transfer intermediate results among worker containers: 1) *Pull*: is used by a down-stream digraph worker to retrieve data from up-stream workers, 2) Push: is used by an up-stream digraph worker to transfer intermediate results to down-stream workers, 3) *Exchange*: redistributes tuples between containers, using one-to-many or many-to-one distribution patterns; in operator parallelism.

The framework process requires three types of containers: Master, Worker, and Data-Only. Each query is orchestrated by one master, which executes query planner steps, then dispatches the plan to container platform tools to schedule worker containers. Subsequently the master progresses query evaluation with control messages to-and-from worker containers. Query operations are executed by cooperating worker containers, each worker executing a specific query tree relational operator such as join, selection or projection, as well as other non-relational operations needed to transfer and reload data among workers. Data-only containers serve workers as a uniform interface for storage and retrieval of data from the underlying cloud storage.

III. Implementation of a Prototype

A reference prototype of the container query framework was implemented using widely available off-the-shelf software. Docker container images built for the Master and Worker container images use the Ubuntu 14.04 operating system as the base layer. Top layers include a lightweight embedded SQLite engine to execute container relational operators, the RabbitMQ messaging service for queued communication among master and workers, and Java programs using the JDBC interface to interact with the embedded SQL engine and which implement other various functions needed by master and worker containers.

A YAML configuration file representing the final query plan is used to schedule and deploy the digraph's cluster of query containers using the Docker container platform. Platform deployment tools include: Docker-Compose and Docker-Swarm. Throughout the query evaluation, the master container monitors the query's overall progress and synchronizes worker inter-container order execution as specified in the digraph.

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00028

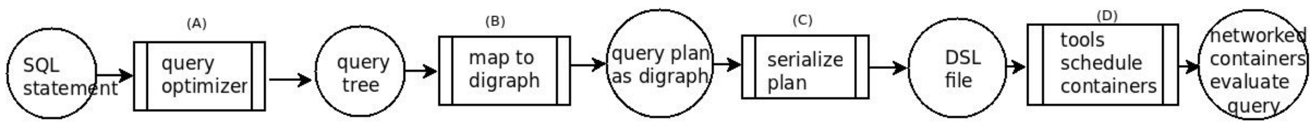

Figure 1. Query Containerization Framework Process Work-Flow

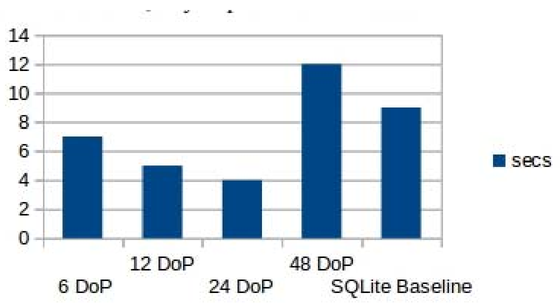

Figure 2. Experiment One

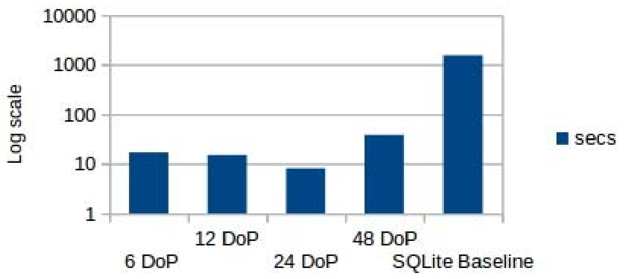

Figure 3. Query Experiment Two

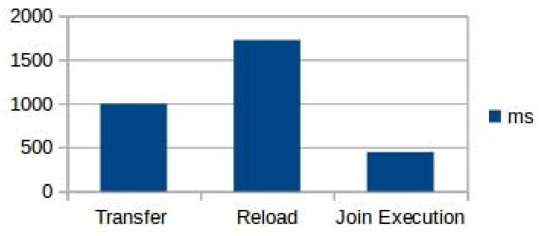

Figure 4. Experiment Three

IV. Experimental Results

Experiments compare the performance of containerized queries with a control baseline SQLite database running on a single cloud VM. The experiments were performed using Docker containers scheduled across a Docker Swarm cluster of six OpenStack VM hosts on the Chameleon research cloud provided by TACC[1]. Three experiments are reported here. Each experiment is repeated multiple times with operators assigned different DOPs to investigate how parallelism effects performance. We tested three distinct complex Select-Join-Project-Order(SJPO) queries using a large public relational data set provided by the MySQL community.

Experiment One tests an SJPO query that contains an equi-join clause. For this query the control baseline SQLite was able to utilize indexing and sorting. The results in Fig. 2 shows starting from 6 DOP that execution times are only reduced marginally compared to the control baseline time; up until DOP increases to 48, an apparent inflection point, when network and container multi-thread overhead begins reducing performance.

Experiment Two tests an SJPO query that contains no equi-join, rather a range join clause using BETWEEN, which eliminates indexing and sorting optimizations to the control baseline planner; forcing full-scans. The result in Fig. 3 shows an 8 secs running time at 24 DOP for the containerized query as compared to 1538 secs running time for the control baseline; a 3-orders of magnitude improvement.

Experiment Three observes 3 operation execution times: 1) transferring intermediate result between workers, 2) reloading intermediate result into downstream workers and 3) execution of container relational operators. A SJPO query with 6 DOP was tested. The result in Fig 4 shows that time used to transfer and reload data in a downstream worker is 7 times greater than needed to execute the query relational operator; clearly a bottleneck, needing further improvement from future research.

[1]Texas Advanced Computing Center at the University of Texas System's Research campus in Austin Texas

V. Conclusions

This paper presented a framework for containerizing a relational (SQL) query, and an experimental reference implementation of the framework using off-the-shelf open source software, e.g. Docker and SQLite. We also reported preliminary results of prototype experiments running complex SQL queries using a large public database on a research cloud.

References

[1] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of ACM*, 53(4):50, April 2010.

[2] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Sigmod*, page 981, 2008.

[3] 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.