Serverless Query Optimizer (SQO)

Jeremy Bogle and Tim Kralj

W December 11th, 2018

Abstract

All modern data management systems (DBMS) have a set of parts that allow it to read, write, and store data as well as perform operations on its data. Since microservices and microservice architectures have become increasingly popular in application development, we were wondering if it would be possible to apply this same idea to a database system to make it more modular and allow for independent development and processing. Since the query optimizer is a specific module that often gets overlooked, we decided this module would be the best to iterate on and improve. The query optimizer takes queries that are already parsed by the query parser, and attempts to find the most efficient logical plan to perform this query. To do this, the query optimizer first tries to search the space of all possible logical plans and estimate the cost of each plan and pick the one that should run the fastest. Often, more exhaustive query searches lead to slower query speeds, but companies need to be competitive on query execution speed and therefore they are not investing money into researching better ones. There has been a lot of work in attempting to prune the search space for query plans so that the optimizer has to consider a smaller number of logical plans and thus can find the best logical plan faster[1]. We think that, because this specific part of the database is very modular, it could be implemented in a lambda function in AWS so that these functions could be performed separately from the actual DBMS. Then, this query optimizer could be iterated on its own as its own module of the larger DBMS. In addition, we believe it is possible to have multiple lambda functions searching different parts of the logical plan space at the same time so that the space can be searched in parallel allowing for less aggressive search space pruning or simply faster logical plan generation. In order to have this happen, lambdas should be treated as if they were actual threads and not separate lambdas. Similar thread-safety issues found on concurrent programs apply to concurrent lambdas; however, one main issue is that lambdas do not have a shared data source to get information from.

I. INTRODUCTION

Cloud databases are becoming a more widely offered service among cloud computing companies. With the world revolving around data collection, being able to store data from anytime - anywhere - all the time - has become a commonplace among any tech company. Both Tim Kralj and I were interested in experimenting with cloud database systems. We were wondering if it would be possible to make a fully distributed database that lives on microservices in the cloud. With this idea being broad, we tried to implement specific parts of the database in AWS lambda functions. Doing this was be an interesting experiment to be able to take different parts of a database and put them together to create a working DBMS. To start narrow, we decided to try to implement just the query optimizer in AWS lambda and compare it to locally running query optimizer.

The query optimizer is an often overlooked part of the DBMS. Until recently, projects involving parallel query plan search, or modular query optimizers, have scarcely been attempted. We imagine a query optimizer that would sit on top of the rest of a DBMS and take queries parsed by the query parser and the find the correct logical plan and return that plan back to the DBMS to takeover the actual query computations. A first attempt at this has come out of Greenplum DB (GPDB) [2] with its project called Orca [3]. Orca is now implemented in GPDB as well as Apache HAWK [4]. With our project SQO, we have learned from Orca, as well as our our simpleDB query optimizer and attempted to turn GPORCA into a serverless function. With a serverless query optimizer living in the cloud, any DBMS could use this state of the art query optimizer to find the proper logical plans before beginning its computations. This also would allow for jobs to be running on the DBMS while query planning is running on lambda functions separately. In addition, the query optimizer could even be shared by multiple DBMs systems. This opens the door for more parallelism as the same way a single thread could search the entire plan space, many query optimizer nodes could compute the best plan for different parts of the plan space and recursively aggregate to find the single optimal plan.

II. Project Details

A. Amazon Web Services (AWS)

1) Lambda: AWS Lambda is a serverless function that runs in the cloud. The architecture contains code that only runs when it is triggered by a specific event. This type of architecture has no underlying servers the user sees. In addition, the user is only billed for actual compute time, not constantly (like a cloud server). AWS lambda has many different runtimes that can be used including Java, python, and most recently C++. This will be used to do computing for the project.

2) API Gateway: AWS API Gateway is a tool to create APIs. They are integrated with the Lambda service very well and make it much easier to integrate and test different AWS services together. This will be used to trigger the lambda functions and return a result to the caller.

3) Elastic Cloud Compute (EC2): AWS EC2 is the service for virtual machines in the cloud. It is used for our testing to compare the results of the lambda output vs regular servers.

B. SimpleDB

Before beginning to tackle a large scale project like GPORCA, we wanted to work with something that we knew. For this, we started with our current implementation of simpleDB. This is a small database we built ourselves. In essence, this is a mini db that is able to run queries and joins but does not have much logic in the form of the query optimizer. It does use histograms but does not do the best job in estimating the selectivity of a filter. We started off with this as our base to try to improve the basic implementation before starting with something new and harder.

Initially, we wanted to improve simpleDB and test the viability of our idea by running it on a serverless function. Starting off, we had to start with fixing simpleDB query optimizer. To do this, we changed the given enumerateSubsets method and improved upon the existing join cardinality estimations. To change enumerate subsets we recursively computed a bitmap of all possible joins and returned iterator over those joins instead of computing all possible joins in multiple nested loops. Then, we used our updated cardinality estimator that used a seqscan to calculate selectivities on different ranges and a cost estimator to estimate the cost of all found plans and return the best plan. With this, we then moved onto deploying this query optimizer on lambda.

The lambda function we created takes in a query

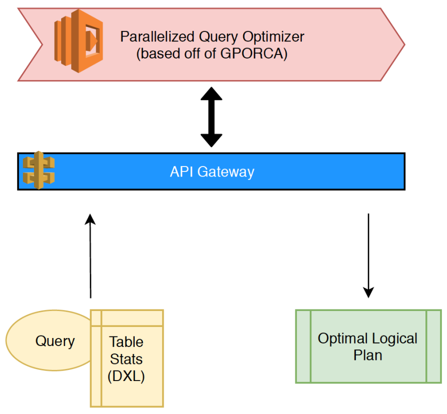

Fig. 1. A diagram showing how a query along with its table statistics, in DXL format, can be passed to API gateway, processed by lambda, and a plan is returned.

from the database, searches for the plan that it believes to be best, and then returns that logical plan. We fronted this serverless function with an API so that any machine is able to query our simpleDB query optimizer. This implementation of the query optimizer did have some downsides, however. For example, all the data from the query from has to already be stored on the AWS lambda function. This is something that we found was not compatible with our simpleDB implementation to fix, since we could not pass data files into our lambda through an API request without the underlying data structures. In addition, there was a lot of unnecessary dependencies within our simpleDB implementation that were needed in order to have the serverless function working but should not be needed for a standalone query optimizer.

https://3xs0c504hi.execute-api.us-east-1.amazonaws.com/test/test

We quickly realized to get this kind of function to run without giving the API request data tables, the lambda would have to communicate with the DBMS in order to receive table statistics to compare logical plans. We saw how GPORCA creates a data exchange language, or DXL, that the DBMS could use to communicate with GPORCA (Figure 2). However, in order to just run a proof of concept, we hosted the entire data tables on the lambda, allowing it to compute statistics locally. We were able to host this optimizer with an API and pass it queries and successfully return a query plan to be used by the locally running DBMS. We saw minimal

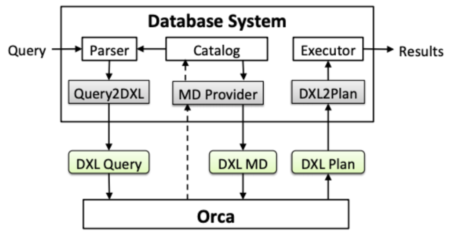

Fig. 2. An illustration of communication between a DBMS and query optimizer using DXL. It is important to note how it requires an initial query and then a request for data from the catalog DXL (MD) before it can return the optimal plan

latency and an effective way to offload some processing from the local DB and decided to move forward with the exploration by tackling GPORCA.

C. GPORCA

Because GPORCA was published and open sourced in 2016, we had the privilege of being some of the first users of the project. This came with a lot of debugging headaches, especially when deploying to lambda. However, before moving to lambda we were able to run GPORCA with standard query benchmarks and analyze its performance.

GPORCA is an multi-threaded parallel query optimizer written in C++. It was created to be one of the first optimizers that makes use of modern multi-core architectures with multiple threads. To do this, the system has a scheduler that adds tasks onto a stack that are then taken off and used. The tasks are added to a dependency graph that determines what parts of the query optimization can be done in parallel and which cannot. There are 7 main steps that are done with orca are: Exp(g), exp(gexpr), Imp(g), Imp(gexpr), Opt(g, reg), Opt(gexpr, req), Xform(gexpr, t) as seen in Figure 3. The scheduler uses the dependency graph to determine which can be done in parallel and batch them out accordingly on various threads. Each of the queries is analyzed using these steps and then it finds the optimal query plan.

Another component of GPORCA is how the system interacts with the database itself. Being a modular query optimizer, GPORCA does not contain the data so it must rely on asking the catalog for the statistics. All of the is done using a format called DXL. This is used to receive the query, ask for data statistics and then to return the final result from the query optimizer (to the system itself). This is an important concept for our own system since we were able to leverage the DXL solution with our own implementation of a query optimizer (Figure 1). In addition, this means that the host DBMS of GPORCA must implement DXL capabilities within the catalog.

III. Challenges

The biggest challenge we saw when trying to modularize the query optimizer is computing table statistics.

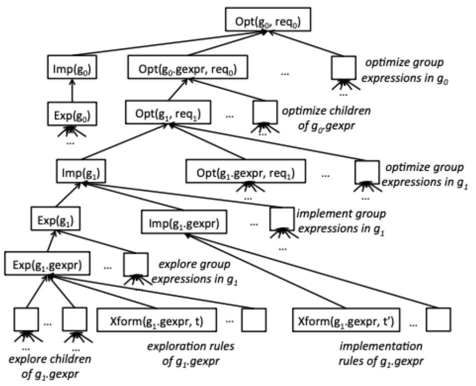
Opt(g0, req0)
Imp(g0)
Opt(g0.gexpr, req0)
optimize group expressions in g0
Exp(g0)
Opt(g1, req1)
optimize children of g0.gexpr
Imp(g1)
Opt(g1.gexpr, req1)
optimize group expressions in g1
Exp(g1)
Imp(g1.gexpr)
implement group expressions in g1
Exp(g1.gexpr)
explore group expressions in g1
Xform(g1.gexpr, t)
Xform(g1.gexpr, t')
explore children of g1.gexpr
exploration rules of g1.gexpr
implementation rules of g1.gexpr

Fig. 3. An illustration of the dependency graph that Orca builds during execution. This is used to determine parallelism within the query optimizer in order to not result in races

With simpleDB, we used small enough tables that we could allow the lambda to compute the table statistics locally so that it could estimate query execution times. To solve this problem, we saw how GPORCA created a DXL language. However, with GPORCA, it required a host DB to also have implemented DXL and be readily available to communicate with the query optimizer. Our spin on this provided the query optimizer with the proper DXL files, or table statistics, up front so that the query optimizer could do the full computation on its own and return the optimal plan, in DXL format. We leveraged GPORCA's DXL in this way so that we could make each query optimization a single-hop process so it could run independently.

Given that GPORCA was so new, it was very difficult to debug this setup. It also has a lot of external dependencies that work differently on different machines. In order to get it to work, it requires specific versions of various build tools including Xerces, Ninja, and Cmake. All of these tools caused hurdles when moving from local machine into a linux based lambda. In addition, the codebase is written in C++ and AWS lambda released support for C++ at AWS re:invent on November 29th, 2018 [5]. While building our C++ deployment package, we had many problems with external libraries and additional test files, to allow for full functionality on the lambda. However, we eventually had our version of GPORCA up and running on lambda with functionality

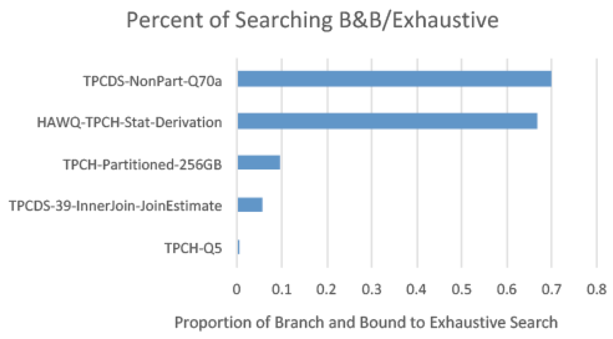

Fig. 4. This graph illustrates the percentage of the query plan space search by branch and bound

for unit tests and query optimization given proper DXL files.

IV. Pruning the Search Space

One specific function of a query optimizer that we wanted to digest was the search space pruning. Every query optimizer has to generate a space of possible logical plans that it want to consider for the query. Then it has to search through that space and evaluate each plan and pick the best plan from those that it evaluates. A widely used method for this is a branch and bound search with a heuristic [6]. GPORCA can do both branch and bound and exhaustive search. One thing it does with its branch and bound is that it tries to put plans that it thinks will be slower later in the search so that when it gets to those it can rule them out more quickly as soon as the estimated runtime exceeds the previously seen fastest runtime.

We compared the percentage of query plan space searched, and time it took, by both branch and bound and exhaustive search by our lambda function. The results are shown in Table I. We can see in this table the in some cases branch and bound aggressively prunes the search space. Other studies, including a paper called ”Exploiting Upper and Lower Bounds” from 2001 have shown that proper, cautious pruning can be done relatively safely, but many widely used heuristic techniques can return a non-optimal plan [1]. With certain queries like TPCH-Q5, we see very aggressive pruning with only .06% of the entire space searched. This pruning successfully decreases runtime. However, the benefit of running

TABLE I

COMPARING BRANCH AND BOUND TO EXHAUSTIVE SEARCH ON THE LAMBDA FUNCTIONS

Query	Exhaustive Search		Branch and Bound		Plans B&B/Exhaustive
	Number of plans	Time (ms)	Number of plans	Time (ms)	Percentage
TPCH-Q5	4997013580	5853	3092440	820	0.000618858
TPCDS-39-InnerJoin-JoinEstimate	18	34	1	30	0.055555556
TPCH-Partitioned-256GB	409239	1963	38580	1064	0.09427254
HAWQ-TPCH-Stat-Derivation	3879264	256	2584454	261	0.666222768
TPCDS-NonPart-Q70a	1.16541E+12	709	8.16541E+11	699	0.700647696

the query optimizer in lambda is that we can offload this processing off of the DBMS, and with this change, have more processing power for efficiently searching this space. In addition, with the multi-threading done by GPORCA's scheduler, it can maximize parallelism by running on a large core machine or on slave lambda functions - explained more in section VI.

The parallelism that is presented by lambdas could increase the search space from branch and bound to a different type of search that prunes less aggressively. This would allow the percent of searches done by our search algorithm to increase, and ultimately improve our final logical plan outputs.

V. COMPARISON OF RUNTIMES

To further evaluate our proof of concept, we wanted to test runtimes to verify that a serverless architecture could provide fast enough speeds to compete with a typical query optimizer. We saw that lambda stacked up well against all machines we tested. The only machine that outperformed the lambda was a 32-core EC2 from AWS. However, these machines can cost up to 3.5$/hour as long while they are up and running so would not feasible or nearly as cost effective for smaller scale projects.

Lambda functions provide a lot of benefit in terms of cost and rival top single node machines in computational power. We can see Figure 6 that lambda can be much cheaper in many use cases. In addition, this graph does not include the fact that the EC2 might not be able to handle that many queries per second, whereas lambdas can scale more as they are not running on a single machine but always run in parallel, in a new lambda,

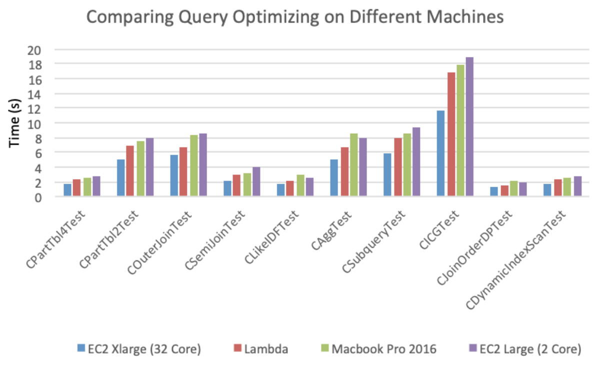

Fig. 5. A comparison of runtimes for sets of queries on various machines. These were all running the same lambda query unit tests. The time for lambda does not include latency (0.2s) per query

when triggered. This idea of concurrent lambdas is explained more in section VI.

VI. APPLICATIONS

A. Concurrent Lambdas

Lambdas are a service that is always available and can scale to whatever amount of requests per second that is required by a system [7]. This is one benefit that a serverless query optimizer has over a regular one. If a system had to handle 1,000,000 requests a second, a server may not be able to handle the requests but 1,000,000 lambda functions a second could be created and maintained at scale for as long as the system needed it to. While there is a breakpoint of price vs requests where lambdas become more expensive; however, there is another cap point that is where a server system cannot physically handle all the requests coming in at once. This is a spot where having a serverless database or a hybrid database with the query optimizer located in a serverless function could be beneficial. These lambdas could help dynamically scale depending on the requests per minute that were being made, and handle large variations in demand with ease. It is important to note that the rest of the database system also has to be able to keep up with such scaling.

B. Multiple Lambdas

A future concept for a lambda based query optimizer is treating lambdas as if they were their own threads. In this case, we could have a master lambda that receives an API call to do some computing. This master node begins the execution and uses its scheduler to spawn off new lambdas instead of threads when it deems necessary (valuing the trade offs of cost and computation speed/power). The dependency graph in Figure 3 shows what parts of a query optimzation call can be optimized with threads. Since the children of a branch do not rely on one another, in theory, one could spin off a new lambda at each point of the graph. This could be used to speed up the same search that would have been done by a server with a limited number of possible cores/threads to use.

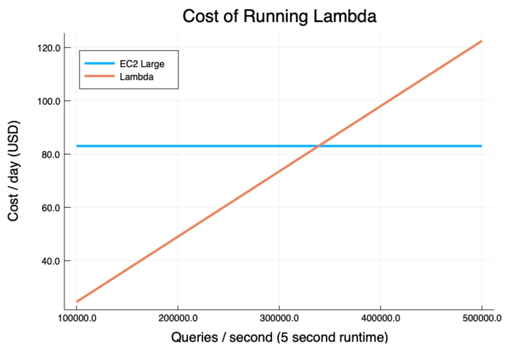

Fig. 6. This figure shows how lambda can be much more cost effective than running large cloud instances. We assume each optimization takes about 5 seconds, which is long for most queries we tested, and also assume that the lambda is running at max compute speed. Equation (1) represents the $.02\$/100000$ requests summed with compute cost for a 3GB lambda for 5 seconds * $hits/second$. The large EC2 cost in equation (2) costs $3.46\$/hr$. We see that as long as your are running less than 350,000 queries per day, lambda is more cost effective
(1) $C_e = (.02 + 0.00024485 * 100000) * (q/s)$
(2) $C_l = 3.46 * 24$

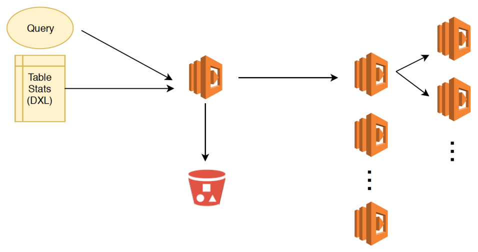

Fig. 7. A potential architecture involving lambdas, spawning other lambdas to parallelize its search. We use S3 to store the table statistics so all other lambdas have access.

C. Lambda Drawbacks

One of the main issues with working with lambdas is they have no persistent state, which makes it difficult to communicate between different lambdas and a data store. On a server, there is a hard drive attached which can be accessed by all threads on that machine; however, a lambda can be thought of as one of those threads without the information behind it. To achieve concurrent lambdas, there needs to be some sort of data stored about the database. There are two logical solutions to this issue. First, when an API call is made to the lambda to process a query, data information can be passed along with it. The information can be moved to AWS S3 (Simple Storage Service) that will hold the information while the concurrent lambdas execute. This way, each of those separate lambda functions can access the information that is needed as shown in Figure 7. Another solution to this problem is to have a lambda make a different API request back to the operating database for the data. This would be similar to the original architecture of ORCA, however, instead of querying the data within the cluster of servers, it would have to send another HTTP request. This would add latency with every subsequent request to the system. Because of this, the second option does not seem viable unless there is a depth limit placed on the number of lambdas that are allowed to be spawned. With this, the rest of the work can be done using the threads on the lambdas itself.

VII. Conclusion

Overall, with in this project we have built a modular, serverless query optimizer (SQO), that can be used anytime from any DBMS. We have evaluated its performance and explored the space of cloud computing. We know that offloading this computation can not only increase the speed of the DBMS, but also increase the speed of the query optimizer as well, all at a lower cost compared to other cloud solutions. The benefits of lambda are just starting to be uncovered as its incredible scalability can be leveraged in many areas of computer science. With SQO we believe that processing queries immediately every time this lambda is triggered rather than waiting for the DBMS to schedule the query is a feature that could be used to improve many database systems today. In a world where data collection is everywhere and cloud computing is still being explored, a lightweight query optimizer like SQO could be a future component for database systems.

REFERENCES

[1] Exploiting Upper and Lower Bounds in Top-Down Query Optimization . https://15721.courses.cs.cmu.edu/spring2018/papers/15-optimizer1/shapiro-ideas2001.pdf.

[2] Greenplum Database: GPDB. https://github.com/greenplum-db/gpdb.

[3] Orca: A Modular Query Optimizer Architecture for Big Data. https://15721.courses.cs.cmu.edu/spring2016/papers/p337-soliman.pdf.

[4] Apache HAWK: Hadoop Native SQL. Advanced, MPP, elastic query engine and analytic database for enterprises. http://hawq.apache.org/.

[5] Introducing the C++ Lambda Runtime. https://aws.amazon.com/blogs/compute/introducing-the-c-lambda-runtime/.

[6] Multiple Query Optimization with Depth-First Branch-and-Bound and dynamic query ordering. https://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article-1184&context=sis_research.

[7] AWS Lambda FAQs . https://aws.amazon.com/lambda/faqs/.