Query and Resource Optimization: Bridging the Gap

Lalitha Viswanathan, Alekh Jindal, Konstantinos Karanasos

Microsoft

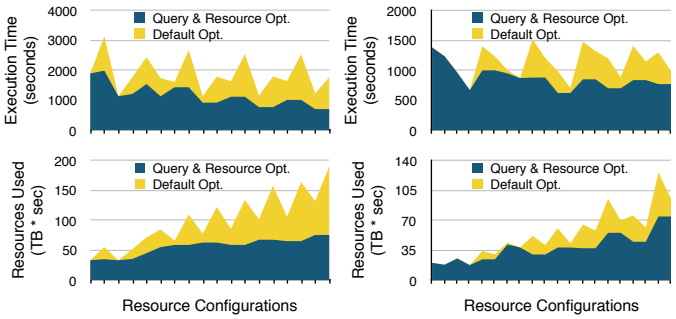
***Abstract*—Modern big data systems run on cloud environments where resources are shared among several users and applications. As a result, declarative user queries need to be optimized and executed over resources that constantly change and are provisioned on demand for each job. This requires us to rethink traditional query optimization designed for systems that run on dedicated resources. In this paper, we show evidence that the choice of query plans depends heavily on the resources that the plan will be executed on. The current practice of determining query plans without accounting for resources could lead to significant performance loss in popular big data systems, such as Hive and SparkSQL. Therefore, we make a case for Query and Resource Optimization (or QROP), i.e., choosing both the query plan and the resource configuration at the same time, and present a research agenda towards this direction.**

I. Introduction

Traditional SQL systems typically rely on query optimizers that determine execution plans assuming a dedicated set of resources or hardware. However, today's cloud computing environments [1], [2] offer a shared pool of resources where per-job resources are provisioned dynamically and on demand, via resource managers, such as YARN [3] and Kubernetes [4]. As a result, SQL-like systems running on these environments, such as Hive [5], SparkSQL [6], and SCOPE [7], need to pick resources *in addition to* the query plan. As an example, Microsoft's Azure Data Lake [8] offers analytics-as-a-service, allowing users to submit their declarative queries without them having to optimize or provision resources for these jobs.

The current practice is to use a *two-step* approach. *First*, a query plan is chosen via a query optimizer [9], [6], [7]. *Then*, the right resource plan, i.e., the resource configuration, is determined through user guesstimates, simple heuristics, or a resource optimizer [10], [11].

Employing this approach means that *the query and the resource optimizer are not aware of each other*, even when the query and the resource plans heavily depend on each other (e.g., when determining the memory size and the join implementation for join processing). Furthermore, the query plans are picked *without considering the current cluster conditions*, which are constantly changing in shared environments. This can result in picking suboptimal combinations of query and resource plans, thereby leading to significant loss in performance or cost.



(a) Hive (b) SparkSQL

Fig. 1. Potential gains of query and resource optimization.

To illustrate the potential gains in case of joint query and resource optimization, we run a join query on the TPC-H dataset on a 10-node YARN cluster, using both Hive and SparkSQL. We experiment with different join implementations (broadcast and shuffle join) and resource configurations (e.g., memory per container, number of containers). Figure 1 reports the execution time and the total resources used for each run. The total resources are measured as the product of the total memory and the total execution time. Our results show that the default optimizer picks the optimal plan for very few resource configurations. In particular, the plans chosen are up to twice slower than those chosen by picking the best plan for the given set of resources. Moreover, they use up to twice more resources, which directly impacts monetary cost.

In this work, we propose a redesign of the optimization stack in big data systems. We show evidence that the current practice of choosing query plans prior to resource configurations leads to significant performance loss and resource wastage (Section III). Therefore, we argue for a more holistic approach, in which the optimizer jointly determines the query *and* the resource plan, while taking into account the current cluster condition (Section IV). We term this approach *Query and Resource Optimization*, and introduce several related open research problems (Section V).

II. Background

Below we describe a few key trends that can be observed in the evolution of big data systems over the past decade.

High-level languages. To avoid writing low-level platform-specific code (such as MapReduce [12], [13]), most systems

provide users with SQL-like *declarative language abstractions* [5], [6], [7], which get translated to an intermediate DAG representation and share the same execution engine. Consequently, efficiently translating the declarative queries to the underlying DAGs, via a *query optimizer*, becomes crucial.

Query optimization. Early systems optimized jobs at the dataflow level, using black-box optimizations coupled to the specific dataflow engine [14]. More recently, big data systems use query optimizers, resembling traditional relational optimizers, to translate a given SQL query to an efficient DAG of operators supported by the underlying dataflow engine. For instance, Hive queries get translated to Tez DAGs [15] and SparkSQL queries to Spark DAGs [16], using either rule- or cost-based approaches [6], [7], [9].

Resource optimization. Clusters are now being shared between multiple applications and users, both on-premises and in public clouds, in order to improve resource utilization and application interoperability. This led the *resource management* layer to get abstracted out of the big data systems [3], [4]. Therefore, along with the translation from SQL query to execution DAGs, the dataflow engine now has to choose the resources to request from the resource manager for each DAG vertex, which is the focus of *resource optimization*.

As an example, YARN, similar to other resource managers, allows applications to request resources in the form of containers, which are resource units comprised of a fixed amount of memory and CPU (or other resources). Resource optimization in YARN involves determining the container size (resources per container), the maximum number of concurrent containers (actual degree of parallelism), and the total number of containers per DAG vertex[1] (total tasks per vertex).

Most systems rely on user configurations (e.g., container size in Hive or degree of parallelism in Spark) or simple heuristics for making such decisions. Early works in resource tuning studied the problem of provisioning Hadoop workloads [17], but their applicability is limited to MapReduce systems. Also, they do not consider dynamically changing resources. Recently, Ernest [11] and PerfOrator [10] focused on the resource optimization problem, relying on executing the job over samples of the input to pick resource configuration.

Lack of joint optimization. Note that all existing resource optimizers take as input a *fixed execution DAG*, produced by a *previously-performed* query optimization step. Furthermore, they do not take into account the per-operator resources nor the current condition of the cluster.

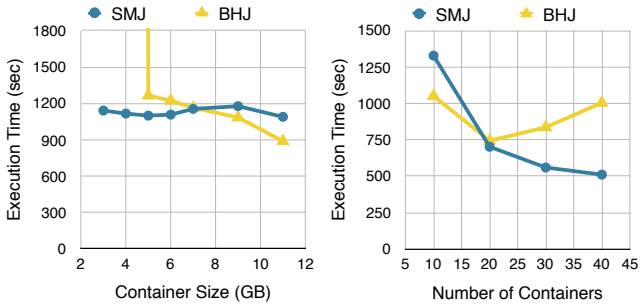
III. Cost of Ignoring Resources

In this section, we study the impact of ignoring resources during query optimization, in particular when choosing operator implementations and join orders.

Setup. For our analysis we deploy Apache Hive (version 2.0.1) on a 10-node YARN (version 2.7.2) cluster. Hive queries get translated to Tez (version 0.9.0) DAGs. Each cluster node has 4 cores at 2.2 GHz, 16 GB of RAM, and a 3 TB data drive. Nodes are connected through a 10 Gbps network. We ran the same experiments on SparkSQL and observed similar trends, but focus on our Hive results here due to space limitations.

We use the TPC-H dataset [18] with scale factor 100, and create Hive tables in ORC format. We measure execution times, excluding the overhead of materializing the join output, and report an average of three runs.

We consider the following resource configurations: (i) container size, and (ii) maximum number of concurrent containers. To simplify our analysis, we consider container sizes in terms of memory, but other resources can be used instead. Finally, in the presented results we use a split size of 256 MB to determine the number of mappers, and enable Hive's feature that automatically determines the number of reducers, since those gave us close to optimal performance.



(a) Varying Container Size (b) Varying #Containers

Fig. 2. Comparing BHJ and SMJ over varying resources.

A. Physical Operators

We look into two commonly-used join implementations in Hive, namely the shuffle sort merge join (SMJ) and the broadcast hash join (BHJ)[2]. Unlike SMJ that shuffles both join relations, BHJ broadcasts only the smaller of the two. BHJ is picked by the optimizer if the size of the smaller relation is below a threshold (determined through a parameter). The default threshold is 10 MB.

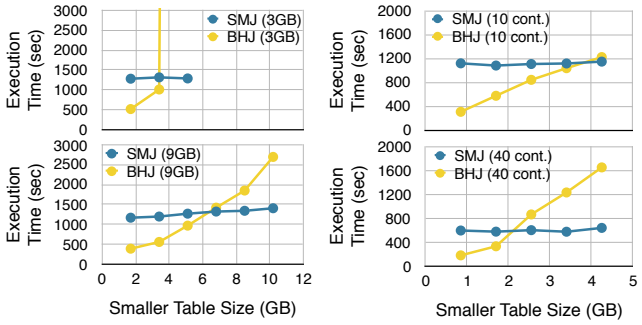
We use the following query: `select * from orders, lineitem where o_orderkey = l_orderkey`. It is based on TPC-H query 12, from which we removed the aggregates and additional filters to focus on the join behavior. In our experiments below, we adjusted the smaller `orders` table size[3] proportionally to the resources we used each time.

Fixed data, varying resources. First, we study the impact of resource configurations on execution time using the two join implementations. Figure 2(a) shows the results for our single-join query (with a 5.1 GB `orders` table), using 10 YARN containers of varying sizes. SMJ outperforms BHJ for container sizes up to 7 GB, while BHJ is better for bigger container sizes. This shows that BHJ benefits from larger

[1]In systems like Spark-on-YARN that reuse containers, following the executor model, this is not applicable, although applications still need to determine the number of tasks.

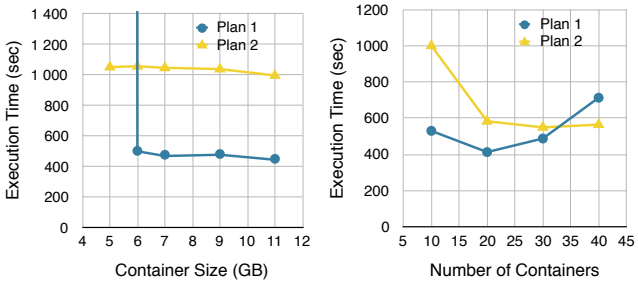
[2]After contacting contributors at Hive, we decided to omit our results for Hive's shuffle hash join, as it is not yet stable.

[3]To select a portion of the `orders` table on demand, we used a uniform sampling filter on `o_orderkey`.



(a) Container Size (b) #Containers

Fig. 3. Comparing BHJ and SMJ switch points over varying data size.



(a) Container Size (b) #Containers

Fig. 4. Join order decisions over varying resources.

memory, whereas the performance of SMJ remains relatively stable. Note that below 5 GB containers, BHJ is not an option as it runs out of memory with the default Hive settings.

Figure 2(b) shows the impact of the number of concurrent containers on the execution times, while keeping the size of each container fixed at 3 GB (using a 3.4 GB `orders` table). We see that while BHJ is better than SMJ for less than 20 containers, SMJ benefits more from increased parallelism and is twice as fast as BHJ for 40 containers.

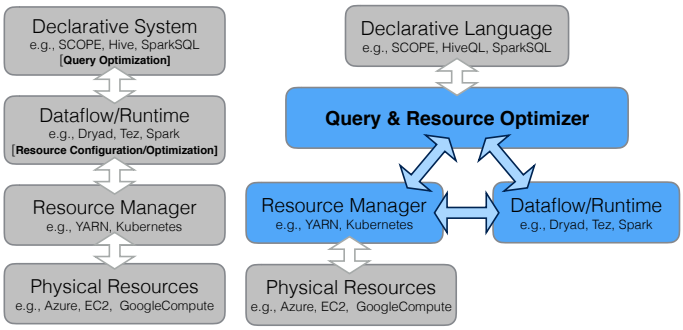
Varying data and resources. So far we saw that there is a switch point for choosing operator implementation when varying resources. But can these switch points be statically determined and hard-coded into the execution engine or are they dynamic? To this end, we also vary the the input sizes.

Figure 3(a) shows the execution times when varying the size of the smaller relation (`orders`) for two container sizes (3 GB and 9 GB). While the switch point between BHJ and SMJ with 3 GB containers is at `orders` size of 3.4 GB (BHJ runs out of memory after that), the switch point shifts to 6.4 GB with 9 GB containers. Figure 3(b) shows the execution times with different number of concurrent containers, keeping the container size fixed. Again, we observe that the switch point between BHJ and SMJ shifts from an `orders` size of 2.1 GB for 10 containers to 3.8 GB for 40 containers.

To recap, the current practice of deciding operator implementations *without accounting for the resources* that will be used can result in significant loss of performance.

B. Join Ordering

We now turn to queries comprising multiple operators to study the impact of resources on different execution



(a) Current Stack (b) QROP Vision

Fig. 5. Current big data system stack with separate query and resource optimization (left), and our QROP vision (right).

plans. We use the following two-way join query, which is a simplified version of TPC-H query 3: `select * from customer, orders, lineitem where c_custkey = o_custkey and l_orderkey = o_orderkey`. We use part of `orders` (850 MB in the first experiment below, 425 MB in the second), so that more BHJs can be used, and make the plan choice more interesting. We compare two plans: (i) *plan 1* first performs a BHJ between `lineitem` and `orders`, and then a BHJ with `customer`; (ii) *plan 2* performs first a BHJ between `orders` with `customer` and then a SMJ with `lineitem`.

Figure 4(a) depicts the execution times for both plans, using 10 concurrent containers and different container sizes, while Figure 4(b) depicts the execution times using 3 GB containers with a varying number of concurrent containers. As shown in the figures, container size does not affect execution times significantly and plan 1 performs better across the board. However, for containers smaller than 6 GB, plan 1 cannot be used as it runs out of memory. On the other hand, the number of concurrent containers does have an impact on execution times. Interestingly, when more containers are available, plan 2 starts outperforming plan 1, with 32 containers being the switch point between the two plans.

IV. The QROP Architecture

Figure 5(a) depicts the current practice in big data systems. First, the query optimizer produces a physical plan for a given query. Then, the resources are picked for executing the selected physical plan and the resource manager (RM) is invoked for acquiring these resources. However, as we saw in Section III, ignoring resources leads to suboptimal plan decisions. Therefore, we propose an alternative architecture that combines Query *and* Resource Optimization (QROP) into a single layer. Figure 5(b) illustrates this new architecture.

The key features of our approach are: (i) we propose a combined optimization for picking both the query plan and the resources at the same time; (ii) the new architecture takes into account the dynamically changing condition of the cluster; (iii) there is a tighter coupling between the optimizer and the resource manager, which enables, for instance, to adapt query plans in case of changing cluster conditions; (iv) given that the

execution time e and the monetary cost c are functions of the emitted query plan p and resource plan r, the optimizer can tune the execution time and the monetary cost that the query will yield when run on the cluster.

The QROP architecture enables several interesting use-cases. We enumerate a few below:

- In case of constrained resources, e.g., due to restricted quota per tenant, we can pick the best plan for a given resource budget: $r \implies p$.
- If a user is satisfied with a given performance, e.g., it meets their SLAs, then they can still optimize their resources for lower monetary cost: $p \implies (r, c)$.
- We can optimize for performance by picking the best query and resource plan combination (p, r). This is useful when there are abundant or even dedicated resources.
- We can pick the resources that yield the best performance for a constraint on the monetary cost: $c \implies (p, r)$.

Overall, the QROP architecture opens up new ways of optimizing big data systems, which are more relevant to shared cloud environments and end user needs.

V. Research Landscape

We now present several open research problems related to our QROP vision.

Query/resource space exploration. Jointly optimizing query and resource plans further increases the search space that the optimizer has to explore. Efficient ways for exploring this space have to be devised.

QROP on complex queries. Generalizing our approach to arbitrary DAGs brings both challenges and opportunities: (i) with multiple operators the possible query/resource plans grow exponentially, (ii) the operators may interact (e.g., via sort orders) and the resource plans may interact too, (iii) if resources between operators do not change, containers can be reused, creating a trade-off between picking best resources per operator and resources that minimize resource allocation cost.

Adaptive QROP. From the moment a query gets optimized until its execution begins, the cluster condition might change. Hence, we might need to adapt/re-optimize the query, instead of waiting for resources to free up. Alternatively, QROP could pick plans that are more resilient to cluster conditions.

Interaction with DAG scheduler. With QROP, the submitted jobs now have precise resource requests. This raises new questions for the scheduler in case the exact resources are not available: should it delay the job, should it fail it, or should it consider multiple query/resource plan alternatives and pick the most appropriate at runtime? Moreover, should the scheduling of tasks to resources adapt based on the selected plan (which could for instance affect the DAG's critical path)?

Interface with resource manager (RM). A restricted optimizer-RM API gives less opportunities for optimizations, while, at the other extreme, exposing all the RM details to the optimizer raises security concerns, especially in a public cloud environment.

QROP and pricing. So far we focused mostly the impact of QROP on execution times. Studying the impact on monetary costs is crucial, and can even lead to new pricing models for cloud environments.

Beyond SQL. QROP can be applied to any system that needs to make query and resource optimization decisions, such as streaming or machine learning systems.

Bridging two communities. Overall, this is an initiative to combine efforts being done separately by the database and the systems community. As the different layers of modern big data systems need to increasingly collaborate with each other, so do the corresponding communities.

Redefining the user's role. Finally, we need to reconsider the user's role in a system that supports QROP. Will the user simply provide the declarative queries and let the system run on autopilot? Are there still control knobs they need to handle? What about troubleshooting and debugging? How will the "explain" command look in such systems?

To conclude, this paper opens the book for combining query and resource optimization in big data systems. This is a major departure from current systems that treat query optimization as an upfront process, while resource optimization is a dynamic runtime activity. We argue that there is a strong interplay between query plans and resource configurations and that the former cannot be chosen independently from the latter. Consequently, the query optimizer, the runtime engine and the resource manager need to be aware of each other in order to produce efficient query plans and avoid wastage of resources.

References

[1] "Amazon Web Services," http://aws.amazon.com.
[2] "Microsoft Azure," http://azure.microsoft.com.
[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *SoCC*, 2013.
[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, 2016.
[5] "Apache Hive," http://hive.apache.org.
[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *SIGMOD*, 2015.
[7] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib, "SCOPE: parallel databases meet MapReduce," *VLDB J.*, 2012.
[8] "Microsoft Azure Data Lake," http://azure.com/datalake.
[9] "Apache Calcite," http://calcite.apache.org.
[10] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan, "PerfOrator: eloquent performance models for resource optimization," in *SoCC*, 2016.
[11] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large scale advanced analytics," 2016.
[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI*, 2004.
[13] "Apache Hadoop," http://hadoop.apache.org/.
[14] H. Lim, H. Herodotou, and S. Babu, "Stubby: A Transformation-based Optimizer for MapReduce Workflows," *PVLDB*, 2012.
[15] "Apache Tez," http://tez.apache.org.
[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud*, 2010.
[17] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the tenant-provider gap in cloud services," in *SoCC*, 2012.
[18] "TPC-H Benchmark," http://www.tpc.org/tpch.