Dynamically Optimizing Queries over Large Scale Data Platforms

Konstantinos Karanasos
IBM Research - Almaden
kkarana@us.ibm.com

Andrey Balmin*
GraphSQL
andrey@graphsql.com

Marcel Kutsch*
Apple Inc.
kutschm@gmail.com

Fatma Özcan
IBM Research - Almaden
fozcan@us.ibm.com

Vuk Ercegovac*
Google
vuk.ercegovac@gmail.com

Chunyang Xia
IBM Silicon Valley Lab
cxia@us.ibm.com

Jesse Jackson
IBM Silicon Valley Lab
jessejac@us.ibm.com

ABSTRACT

Enterprises are adapting large-scale data processing platforms, such as Hadoop, to gain actionable insights from their "big data". Query optimization is still an open challenge in this environment due to the volume and heterogeneity of data, comprising both structured and un/semi-structured datasets. Moreover, it has become common practice to push business logic close to the data via user-defined functions (UDFs), which are usually opaque to the optimizer, further complicating cost-based optimization. As a result, classical relational query optimization techniques do not fit well in this setting, while at the same time, suboptimal query plans can be disastrous with large datasets.

In this paper, we propose new techniques that take into account UDFs and correlations between relations for optimizing queries running on large scale clusters. We introduce "pilot runs", which execute part of the query over a sample of the data to estimate selectivities, and employ a cost-based optimizer that uses these selectivities to choose an initial query plan. Then, we follow a dynamic optimization approach, in which plans evolve as parts of the queries get executed. Our experimental results show that our techniques produce plans that are at least as good as, and up to 2x (4x) better for Jaql (Hive) than, the best hand-written left-deep query plans.

Categories and Subject Descriptors

H.4 [**Database Management**]: Systems

Keywords

query optimization; large-scale data platforms; adaptive query processing; pilot runs

1. INTRODUCTION

Large scale data processing platforms, such as Hadoop, are being extensively used by enterprises in order to store, manage, analyze and exploit their "big data". These large scale data platforms are popular for various different types of applications, including text analytics on unstructured data or log analysis over semi-structured data, while they are also increasingly used for relational-like processing with many joins over semi-structured and structured data, which is the focus of this paper. In addition to the large data volumes, there are other important characteristics that distinguish this environment from traditional relational query processing. First, nested data structures, such as structs, maps, and arrays, are pervasive as users are commonly storing data in denormalized form. Second, users push more complex business logic closer to the data, resulting in heavy usage of user-defined functions (UDFs) in queries.

High-level declarative languages for large scale data platforms, such as Hive [36], Pig [18] and Jaql [5], gained popularity due to their ease of use, and increased user productivity. As with any declarative query language, query optimization is a key challenge, as the system has to decide in which order to execute various query operators, as well as which implementation method to use for each operator. Join is a particularly important operator with alternative implementations that have very different performance characteristics depending on cardinalities of inputs and output. As a result, many techniques of relational query optimization, such as selectivity and cardinality estimation, join method selection, and join ordering are still applicable to the new large scale data languages.

Obviously, not all aspects of traditional query optimization fit well in these new environments. Traditional query optimizers rely on data statistics to estimate predicate selectivity and result cardinality, and use these estimates to decide on join method and ordering. Even in the relational setting, optimizers are plagued with incorrect cardinality estimates, mainly due to undetected data correlations, existence of UDFs and of external variables (in parameterized queries). Various solutions have been proposed to capture data correlations, such as CORDS [26], but these require very detailed and targeted statistics. Collecting such statistics on all datasets may be prohibitively expensive in large clusters.

In this paper, we address the query optimization problem for large scale data platforms, by continuously collecting statistics during query execution, and feeding these statistics to a cost-based query optimizer to decide the next query sub-expression to execute. We have implemented our techniques in the context of Jaql. When a query is submitted to Jaql, we first identify operations that are local to a table[1], such as selections, projections, and UDFs. We utilize "pilot runs" to execute these local plans over a sample of each base dataset, until enough results are produced to collect meaningful statistics. Then, we present each local sub-plan, along with the collected statistics and the corresponding join graph to a simple

*Work done while the author was at IBM Research.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610531.

[1]An operation is local to a table if it only refers to attributes from that table.

cost-based optimizer to decide the initial join order. Our optimizer, built on top of the open-source extensible Columbia optimizer [12], estimates join result cardinalities using textbook techniques, however, it operates on very accurate input cardinality estimates for local sub-queries, as these are collected either during pilot runs or previous execution steps. Our optimizer chooses the appropriate join methods and order, as well as marks the joins that can be executed together in the same MapReduce job. We provide this join order as input to the Jaql compiler, which generates the MapReduce jobs to execute the query plan. As the query gets executed, we collect statistics on the intermediate results, which we can use to re-optimize the query, if needed. In our current implementation, we re-optimize after each job, but the infrastructure is rich enough to accommodate more complex conditions. For example, one can decide to re-optimize if the difference between the observed result cardinality and the estimated one exceeds a certain threshold.

There has been a lot of work on adaptive query processing [15, 28, 31, 22] to address the shortcomings of traditional optimizers. These adaptive techniques use various run-time solutions that try to adjust the query plan dynamically, similar to ours. However, these works did not consider pilot runs, which provide the means to accurately estimate the result sizes when there are UDFs, and correlations between different attributes in the query. Although pilot runs introduce overhead, they amortize it by avoiding really bad plan choices. Earlier approaches on adaptive query processing can also benefit from pilot runs, by avoiding mistakes in the first optimization decisions. Further, in a relational setting queries are mostly pipelined and it has been a challenge to identify the correct re-optimization points. On the contrary, in a distributed large scale cluster, checkpoints are typically present, since the system has to account for failure recovery; restarting a long running query may not be a viable option. These checkpoints also enable the system to better utilize resources, as it greatly simplifies load balancing.

In our setting, we exploit the fact that MapReduce jobs always materialize their output and use them as natural re-optimization points, but our techniques are not restricted to MapReduce systems. Pilot runs are applicable to any massively parallel data processing system, provided that the query needs to scan large enough data to amortize its overhead. We believe that intermediate result materialization, which provides us with the opportunity to re-optimize a query, will always be a feature of many large scale data platforms designed for long running queries (e.g., Spark, Tez), not only MapReduce. It is also important to note that although our dynamic techniques are developed within the context of Jaql, they are also applicable to other large-scale query languages, such as Hive.

Our experimental results show that our techniques produce plans that are at least as good as, and often even better than, the best hand-written left-deep query plans. They are also most of the time better than the plans produced by a state-of-the-art relational query optimizer for a shared-nothing DBMS. Pilot runs introduce a small overhead, but provide accurate result size estimation, especially when the query contains UDFs and data correlations. Accurate result size estimation not only leads to better execution plans, but also prevents the optimizer from making fatal mistakes. Re-optimizing the query as it executes proves to be very beneficial as we observed many plan changes in some queries (see Figure 2).

In this paper, we make the following contributions:

- We propose pilot runs, which execute local plans over a sample of the base data, and provide accurate result size estimation, which is essential to choose the right join method and order.
- We utilize a traditional query optimizer to determine the join methods and global join ordering. The optimizer is provided with the statistics of local plans, as if they are base tables, and does not estimate local predicate selectivities.
- We collect result statistics, which can be used to re-optimize the remaining of the query and hence provide a means to adapt the execution plans at runtime.
- Our optimizer produces both left-deep and bushy plans. We exploit bushy plans and explore novel strategies for scheduling multiple jobs in parallel to run in a distributed environment.
- We provide detailed experiments, where we assess the overhead of our techniques, study different execution strategies for bushy plans, and demonstrate that pilot runs and re-optimization can be very beneficial for Jaql, but also for other systems such as Hive.

The paper is organized as follows. Section 2 provides an overview of Jaql, whereas Section 3 describes the architecture of our system. Section 4 presents the pilot runs, and Section 5 details the dynamic re-optimization of execution plans. In Section 6 we present our experimental results, in Section 7 we review related work, then we conclude.

2. JAQL OVERVIEW AND QUERY EXECUTION

In this section we provide a general overview of Jaql [5], and describe its join processing capabilities.

2.1 Jaql Overview

Jaql is a system for analyzing large semistructured datasets in parallel using Hadoop's[2] MapReduce framework [14], and is part of IBM BigInsights[3]. It shares common features with other data processing languages that were also developed for scale-out architectures, such as Hive [36], Pig [18] and DryadLINQ [41]. These common features make the techniques developed in this paper applicable to those systems with slight modifications, as will be discussed in Section 3. Jaql consists of a declarative scripting language, a compiler, and a runtime component for Hadoop, which we briefly present below.

Query language Each Jaql script consists of a set of statements, which are either variable assignments or expressions to be evaluated. Each expression gets an input and produces an output, which can then feed another expression. Jaql supports all relational expressions, such as filter, join, sort, and group by, as well as other expressions, such as split and tee. It also supports a SQL dialect close to SQL-92; SQL queries submitted to Jaql are translated to a Jaql script by the compiler. Finally, users can provide their own UDFs, either as Jaql scripts or in an external language (e.g., Java).

Compiler Jaql's compiler includes a heuristics-based rewrite engine that applies a set of transformation rules in order to optimize the input scripts. Such rules include simplification of the script, translation of declarative expressions (e.g., joins) to low-level operators, various database-style optimizations (e.g., filter push-down), etc. One of the most important rules is the `toMapReduce` rule that, whenever possible, transforms an expression to a `mapReduce` function that can then be evaluated using the MapReduce framework.

Runtime The Jaql interpreter evaluates the script locally on the computer that compiled the script, but spawns interpreters on remote machines using MapReduce, in order to parallelize the execution. Expressions that cannot be parallelized (due to the nature of the script or the limitations of the compiler) are executed locally.

[2]http://hadoop.apache.org

[3]www.ibm.com/software/data/infosphere/biginsights

2.2 Join Processing in Jaql

In this section, we discuss the join algorithms and join optimizations supported by Jaql, as well as their limitations.

2.2.1 Join Algorithms

There are three possible join strategies in any distributed query processing system regarding data movement: shuffle none, one or both join tables. Shared-nothing databases implement all three strategies [16, 4], whereas most query processors over large scale data platforms implement only the second and third option (broadcast and repartition join, respectively) [7]. Jaql currently implements a flavor of both the repartition and broadcast joins.

Repartition join This join is implemented in one MapReduce job, comprising a map and a reduce phase. Every map task operates over a split from one of the two input tables, tags each record with the table name, and then outputs the extracted join key value and the tagged record as a (key, value) pair. The outputs are partitioned, merged and sorted by the MapReduce framework, and all records from both tables having the same join key are sent to the same reducer. At every reduce task, for each join key, the corresponding records are separated in two sets according to the table they belong, and then a cartesian product is performed between the two sets.

Broadcast join This join method is implemented in a map-only job and can be used if one of the two input tables fits in memory. Let S be the small table and R be the bigger one. In that case, S is broadcasted to all mappers, and a memory hash join is executed between S and each split of R. Thus, S is the build input and each split of R the probe input of the hash join. In the current Jaql implementation spilling to disk is not supported. Hence, if the build side of the join does not fit in memory (e.g., after the application of a UDF that produces more tuples than its input), the execution of the join, and hence the query fails due to an out of memory error.

As expected, the broadcast join is faster, because it avoids sorting the small relation during the map phase, and, most importantly, avoids reshuffling the big relation over the network.

2.2.2 Join Optimizations

The default join algorithm in Jaql is the repartition join. Hence, the compiler translates each join operator to a repartition join, which is executed as a single MapReduce job, that is, an n-way join is translated to n MapReduce jobs. To improve the performance of the join expressions, the user can provide a hint stating a relation is *small*. In this case, the compiler creates a broadcast join having the small relation at the build side of the join. Moreover, Jaql uses a rewrite rule that checks the file size of the two join inputs, and, in case one of them fits in memory, produces a broadcast join plan.

In case of n-way joins, the compiler is capable of producing only left-deep plans. The exact order of the joins depends on the order of the relations in the `FROM` clause of the query. In particular, the relations are picked in the order they appear in the `FROM` clause, as long as this does not lead to cartesian products; in that case, a relation that avoids the creation of cartesian products is picked first.

Finally, when there are more than one consecutive broadcast joins, and the relations that appear in the build side of these joins simultaneously fit in memory, the join expressions can be *chained* in order to be executed in a single map job in a pipelined way.

2.2.3 Limitations of Current Join Processing

The current join processing in Jaql has several limitations:

- There is no selectivity estimation for the predicates and UDFs. As a consequence, the application of broadcast joins is limited only to relations that already fit in memory. Thus, we are missing opportunities for broadcast joins in case of relations that could fit in memory after the application of a selective filter. On the other hand, even in cases when relations fit in memory, a UDF could increase the size of the relation, hence we have to be conservative in the application of broadcast joins.
- There is no cost-based join enumeration. Therefore, the performance of a query heavily relies on the way it has been written by the user (order of relations in the `FROM` clause).
- Even if the order of relations in the `FROM` clause is optimal, Jaql only produces left-deep join plans. In the traditional (centralized) relational setting, these plans are usually close to the optimal ones, since they maximize the pipelining of operators. In the MapReduce setting though, where the result of a job is always materialized to disk and pipelining is not useful, the basic optimization goal is to reduce the size of intermediate results. To this regard, bushy plans may be much more efficient, while they also allow executing parts of the plan in parallel. The importance of bushy plans is also pointed out in [39].

The above limitations are present in most existing large scale data platforms. Thus, the techniques presented in this work would also be beneficial for those systems. We discuss how our techniques can be extended to other systems in Section 3, while in Section 6.6 we experimentally show how Hive benefits from our techniques.

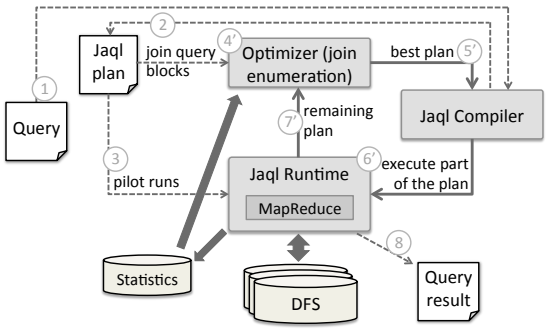

Figure 1: System Architecture.

3. SYSTEM ARCHITECTURE

In this section we introduce DYNO, a system we built for optimizing complex queries over Hadoop data. The architecture of the system is shown in Figure 1. The dashed lines denote actions that are performed only once, whereas the solid lines (associated with primed numbers, 4'-7') refer to repeated actions. When a query arrives in the system (step 1), the Jaql compiler is invoked and some logical heuristic rules, such as filter push-down, are applied [34] (step 2). Once this is done, multiple *join query blocks* are created, which are expressions containing n-way joins, filters and scan operators. Join blocks are separated from each other by aggregates, grouping and ordering operators. Before further optimizing and executing each join block, we first perform the *pilot runs* (step 3) in order to collect statistics that take complex predicates and UDFs into account. Hereafter, we will refer to such predicates/UDFs that are applied right after a scan, simply as *local predicates*, distinguishing them from the non-local ones that are applied on results of joins and cannot be pushed-down to the leaves of the operator tree. For each scan in the join block, along with its local predicates (pushed down by the compiler), we execute the corresponding expression over a sample of the data and collect statistics. More details about this process are given in Section 4.

After the pilot runs, in each join block we consolidate the scans with their local predicates, since the pilot runs statistics capture the

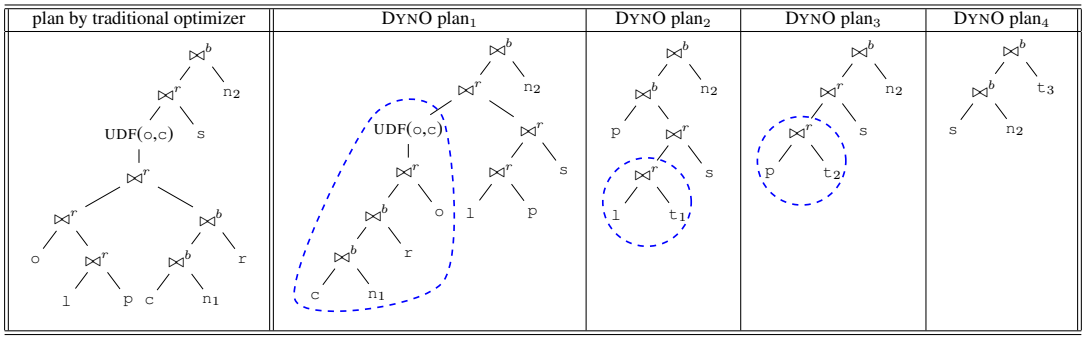

Figure 2: Execution plans for TPC-H query Q8′.

local predicate selectivities. Then, we feed each such join block and the collected statistics to our cost-based optimizer, which now only needs to focus on join enumeration. The best plan output by the optimizer is then passed to the Jaql compiler that translates it to a set of MapReduce jobs (step 5'). Note, however, that instead of executing all the MapReduce jobs at once, we execute only a subset of them (step 6'). The exact subset and the way the jobs will be executed (possibly in parallel) is decided by the compiler. During execution, we collect join column statistics over the output of the jobs and update the existing statistics. Our optimizer is then re-invoked over the part of the join block that was not yet executed (coming back to step 4'). If the statistics of the previous step were not accurate, a new plan, different than the previous one, may be selected by the optimizer. This iterative process is continued until the whole plan gets executed, and is detailed in Section 5. Finally, the result is returned to the client that posed the query (step 8).

As an example, consider TPC-H [37] query Q8, which includes a 7-way join block and to which we added a filtering UDF on the result of the join between `orders` and `customer` tables. On the lefthand side of Figure 2 we depict the plan that we obtained from a state-of-the-art relational optimizer for a shared nothing DBMS[4]. In the same figure, we also give the plans we obtained from our optimizer as the execution of Q8′ proceeds. In particular, plan_1 is the plan we got after the pilot runs. Then, during execution, we have three re-optimization points that in turn produce plan_{2-4}. Note that the relational optimizer's plan will be executed with one map-only and four map-reduce jobs. On the other hand, plan_1 includes four map-reduce jobs, and after re-optimizations Q8′ finally gets executed in three map-reduce and one map-only job.

Applicability to other systems Any query processing system over a large scale data platform that supports the typical join and group-by operators together with UDFs can incorporate our techniques with the following changes:

- Instrument the runtime (e.g., the map and reduce tasks) to collect statistics over attributes.
- Implement the pilot runs by executing the scans together with their local predicates over a sample of the data.
- Our cost-based optimizer for join enumeration can be used as is, since similar join algorithms are supported by most systems, such as Hive and Pig (as discussed in Section 2.2).
- Create a new dynamic join operator that calls the optimizer at runtime to re-optimize the remainder of the query.
- Identify re-optimization points in the plan. If the query already contains checkpoints (e.g., MapReduce job boundaries, Tez stages, explicit checkpoints in Spark), those can be used. Otherwise, strategies similar to the ones proposed for the relational context [28, 2, 31, 22] can be used to identify the re-optimization points.

[4]For readability, in both Figures 2 and 3, we use the initials of the TPC-H tables, e.g., `p` denotes `part` and $\texttt{n}_x$ denotes `nation`.

4. PILOT RUNS

In this section, we describe the *pilot runs*, which we use to collect the initial statistics for a given query q, taking into account the predicates and UDFs that participate in q.

4.1 The PILR Algorithm

Most traditional relational optimizers rely on statistics for estimating the selectivity of operators, costing alternative execution plans and finally choosing the minimum-cost plan. Some of these optimizers use very detailed data distribution statistics and hence can accurately estimate the selectivity of simple predicates. When multiple predicates are applied on a relation, they usually rely on the independence assumption and simply multiply the selectivities of each simple predicate to compute the final selectivity [35]. However, the independence assumption does not hold when there are correlations between the attributes used in predicates, and simply multiplying the selectivity of individual predicates leads to overly underestimated result sizes. Underestimation can result in extremely poorly performing query plans, as the optimizer will choose operators that will work well when its operands fit in memory, such as sort-merge-join, or broadcast join. Incorrect selectivity estimation is exacerbated in the presence of more complex predicates or UDFs, or when the predicates involve complex data types, such as arrays and structs.

As an example, consider the following query Q_1 that asks for the names of the restaurants in California with zip code 94301, having positive reviews (a UDF is used for the sentiment analysis of the review). Each restaurant can have multiple addresses (thus the `addr` is an array type), but we are interested only in the primary address. The reviews are joined with tweets and a second UDF (`checkid`) is applied to verify the identity of the users:

```
SELECT rs.name
FROM restaurant rs, review rv, tweet t
WHERE rs.id=rv.rsid AND rv.tid=t.id
AND rs.addr[0].zip=94301 AND rs.addr[0].state=CA
AND sentanalysis(rv)=positive AND checkid(rv,t)
```

First, notice the correlation between the two adress predicates: the predicate on the state is redundant, since all restaurants with the given zip code are in CA. In this case, even if the optimizer could deal with the array datatype, the independence assumption would lead to the wrong selectivity estimation. Moreover, the selectivity of the two UDFs cannot be computed by a relational optimizer.

To account for such non-trivial predicates/UDFs in the queries, we employ the *pilot runs*. The basic idea is to apply the predicates/UDFs of each relation R over a sample of R and collect

Algorithm 1: Pilot Runs Algorithm (PILR)

Input : query q, $|\mathcal{R}|$ relations in q, dataset $\mathcal{D}$, number of records k to collect statistics per relation, m map slots in the cluster
Output: Statistics over a sample of the data after applying local predicates and UDFs.

1 $queryExpr \leftarrow$ `jaqlParse`(q)
`// Push down predicates/UDFs`
2 $queryExpr \leftarrow$ `filterPushDown`$(queryExpr)$
`// Get table scans with predicates/UDFs`
3 $leafExprs \leftarrow$ `getLeafExprs`$(queryPlan)$
4 **foreach** $lexp_R \in leafExprs$ **do**
5 **if** $lexp_R \notin Statistics$ **then**
6 $mapJob \leftarrow$ `toMapReduce`$(lexpr_R)$
 `// Pick` $m/|\mathcal{R}|$ `random splits for the corresponding relation`
7 $splits \leftarrow$ `reservoirSample`(`input`$(mapJob), m/|\mathcal{R}|)$
 `// Collect statistics over` k `records`
8 $stats \leftarrow$ `execute`$(mapJob, splits, k)$
9 `addToStatistics`$(stats)$

statistics based on the output. The corresponding algorithm (PILR) is given in Algorithm 1. Given a query q, with $\mathcal{R}$ being the set of relations in q, we first use the Jaql compiler to parse the query and perform all the possible predicate/UDF push-downs (lines 1-2). Then, we collect the leaf expressions of q, that is, the scans for each relation in $\mathcal{R}$ together with the predicates/UDFs that appear immediately above the scans (line 2). Clearly, there is one leaf expression for each relation $R \in \mathcal{R}$, denoted $lexp_R$. Subsequently, we transform each $lexp_R$ to a map-only job and execute it over a sample of R until k tuples have been output by the filter or we have finished scanning R (lines 6-8). More details about the sampling process and the execution of the pilot runs (lines 7-8) are provided in Section 4.2. The statistics that are collected during the execution of $lexp_R$ are added to the statistics metastore. In this paper, we store the statistics in a file, but we can employ any persistent storage, including key-value stores, and relational DBMSs.

Coming back to the above query Q_1, three pilot runs need to be executed: one for relation `rs` (applying the two predicates on `addr[0]`), one for `rv` (applying the `sentanalysis` UDF), and one for t (with no predicates). As a second example, consider TPC-H query Q9, which includes a 5-way star join. We have added various UDFs to it, so that all dimensions fit in memory. On the lefthand side of Figure 3 we give the plan that is picked by a state-of-the-art relational optimizer for a shared nothing DBMS. The optimizer cannot estimate the selectivity of predicates involving UDFs and produces a plan where all joins are expensive repartition joins. On the other hand, after performing the pilot runs and feeding our optimizer with the updated statistics, we get a plan having only broadcast joins (righthand side of figure).

Reusability of statistics To avoid performing unnecessary pilot runs in cases when we have already collected statistics for a specific relation R with the same predicates, we associate the statistics in the metastore with an expression signature (corresponding to the leaf expression $lexp_R$). Thus, before executing the pilot runs, we look-up for existing statistics using the corresponding signature and perform the run only in the absence of statistics (line 5). This is useful in cases of recurring queries, or when the same relation and predicates appear in different queries.

Moreover, note that if there are no predicates/UDFs participating in a leaf expression $lexp_R$ and there are already available statistics for relation R, we can avoid executing $lexp_R$ and instead use the existing statistics for R. For instance, in Q_1 above, since no predicates are applied on t, we do not need to execute the third pilot run if we already have statistics for t.

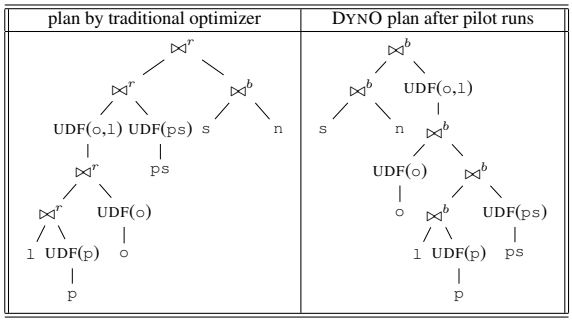

Figure 3: Execution plans for TPC-H query Q9′.

Optimization for selective predicates When the predicates applied on a relation are very selective, the whole input relation may be consumed before outputting k tuples. In this case, we use the output of the corresponding map job (that is in any case written to HDFS) during the actual execution of the query, instead of running it again. This is very beneficial especially for expensive predicates/UDFs that take a considerable amount of time to be executed. Taking this a step further, in such cases, if the job is close to completion when k tuples have been output (e.g., 80% of the base relation is scanned), we can let the job finish in order to be able to reuse the output when the query gets executed.

4.2 Execution of Pilot Runs

We have implemented two different ways of executing the PILR algorithm: PILR_ST and PILR_MT. First, we started by implementing PILR_ST, where we submit the i-th leaf expression $lexp_R$ only after the (i-1)-th leaf expression has finished executing. For each $lexp_R$, all map tasks (or to be precise, at least the first wave of map tasks) are started. Each map task increases a global counter (that is stored in ZooKeeper[5]) as it outputs records. When the global counter becomes bigger than k, the job gets interrupted. This was the simplest approach to implement, since at the time Jaql did not have support for submitting multiple MapReduce jobs in parallel. However, it has various drawbacks. First, by not submitting the leaf jobs simultaneously, we pay the startup cost of the MapReduce jobs (which could be as high as 15-20 seconds) for each relation in the query, and we underutilize the cluster. Moreover, as each job terminates early, its tasks are interrupted before they finish processing their input data blocks. This may lead to a bias in our estimates, due to "inspection paradox", as described in [32]. Essentially, tasks that produce smaller outputs are likely to process records faster and thus skew output statistics. We avoid this "inspection paradox" by finishing processing each data block that we already started.

To address these drawbacks, we implemented the second variant, PILR_MT, where we have modified the input format so that, given a query with $|\mathcal{R}|$ relations, we pick exactly $m/|\mathcal{R}|$ random splits for each relation, where m is the number of map slots in the cluster. We also modified the execution engine of Jaql to submit multiple jobs at a time. To this end, we are now able to start all leaf jobs together, avoiding the job startup cost $|\mathcal{R}|$ times, and better utilizing the resources of the cluster. Note that if the $m/|\mathcal{R}|$ splits are not sufficient for getting our k-record sample, we pick more splits on demand with a technique similar to [38]. Our ability to dynamically add random splits to the sample allows us to solve the main problem of block-sampling, i.e., sample sizing [10]. Note that

[5]http://zookeeper.apache.org

Query	SF100-ST	SF100-MT	SF300-MT	SF1000-MT
Q2	100%	27.7%	26.2%	27.5%
Q8′	100%	16.9%	19.1%	16.0%
Q9′	100%	19.4%	18.9%	20.6%
Q10	100%	24.8%	24.2%	24.4%

Table 1: Relative execution time of PILR for varying queries and scale factors.

requiring k output records is a generic yet imperfect method of assessing the sample quality. However, using a shared data structure in ZooKeeper, we can easily plug in other such methods, such as cross-validation used in [10].

To show the impact of PILR_MT, we have used four TPC-H queries to compare PILR_ST using scale factor 100 with PILR_MT for increasing scale factors (100, 300, 1000). The results are given in Table 1, where the numbers are normalized to the execution time of PILR_ST for SF=100. PILR_MT brings an average performance speedup of 4.6x over PILR_ST. Importantly, the performance of PILR_MT does not depend on the size of the dataset, but only on the size of the sample k.

In the rest of the paper, whenever we refer to PILR, we will be using the PILR_MT variant. As will be shown in Section 6.2, the overhead of PILR on the overall query execution is in most cases about 3%, which is not significant, especially given its advantages.

4.3 Collected Statistics

During the pilot runs we keep global statistics for each table (namely, table cardinality and average tuple size), as well as statistics per attribute (min/max values, and number of distinct values). In order to reduce the overhead of statistics collection, we only collect statistics for the attributes that participate in join predicates.

For each pilot run over a relation R, let R_o be the tuples of R that are output after the application of the predicates, $|R_o|$ be the corresponding number of tuples and $size(R_o)$ the size (in bytes) in disk (these numbers are computed by the counters exposed by Hadoop). Then, the average record size[6] is computed as $rec\_size^{\epsilon}_{avg} = \frac{size(R_o)}{|R_o|}$. Moreover, the table cardinality of R (after the predicates) is estimated as $|R|^{\epsilon} = \frac{size(R)}{rec\_size^{\epsilon}_{avg}}$.

Collecting the minimum and maximum value for each attribute is done by finding for each HDFS split the corresponding values and combining them to find the min and max values for the whole sample. This combination is done in the Jaql client to avoid a reduce phase in the pilot run jobs. Estimating the number of distinct values of an attribute is more involved and is detailed below.

Distinct values estimation via KMV synopsis Let R be one of the relations of the dataset $\mathcal{D}$ and A be one of its attributes for which we want to compute the number of distinct values DV_A. Scanning R and keeping track of all the distinct values of A can incur a prohibitive cost for big relations, especially in terms of memory. Instead, a synopsis for A can be created and then used to estimate DV_A (we denote the estimation by DV^{ϵ}_A). In particular, we opted for the KMV synopsis [6], which allows to create one synopsis for each partition R_p of R and then easily combine the partial synopses to get a global synopsis for the whole relation. Thus, the process can be executed in parallel for each HDFS split.

Let $\mathcal{S}^A_{R_p}$ be the KMV synopsis for partition R_p, and h be a hash function with domain $\{0, 1, \ldots, M\}$. The synopsis $\mathcal{S}^A_{R_p}$ is a set containing the k (with k>0) minimum hash values for the values of A that appear in R_p. For a dataset with D distinct values, the size of the synopsis is $\mathcal{O}(k \log D)$, and the cost of creating it is $\mathcal{O}(|R_p| + \log k \log D)$, where $|R_p|$ the number of tuples in R_p. Creating the global synopsis $\mathcal{S}^A_R$ is equivalent to computing the union $\bigcup_{R_p \in R} \mathcal{S}^A_{R_p}$ of all partial synopses and then keeping the k minimum values of the union. Let h_k be the biggest value in $\mathcal{S}_R$. Then, as shown in [6], we can estimate the number of distinct values $DV^{\epsilon}_{A,R}$ of A in R by using the unbiased estimator via the following formula: $DV^{\epsilon}_{A,R} = \frac{(k-1)M}{h_k}$.

In our setting, we compute the synopsis for each HDFS split that is processed at the map phase of the pilot runs, and then compute the global synopsis out of the partial ones in the Jaql client. Using this estimation, for $k = 1024$, the upper bound on the error of DV^{ϵ}_k is approximately 6%.

Given that during our pilot runs we consume only a sample R^s of each relation R (i.e., $R^s \subseteq R$), we need to extrapolate the distinct value estimation we got over R^s to the whole relation R. Computing the number of distinct values based on a sample of the dataset still remains a challenging task, especially when data is skewed or when there are many distinct values, each having a low frequency [9]. In this work, we used the following simple formula for estimating the number of distinct values for R, given the distinct values for R_s: $DV^{\epsilon}_{A,R} = \frac{|R|}{|R_s|} DV^{\epsilon}_{A,R_s}$. We plan to focus on more precise extrapolations as part of our future work.

Additional statistics Further statistics can be collected, including frequent values, percentile and histograms. This would lead to more accurate cost estimations and possibly better plans, but would increase the overhead of statistics collections. We chose to collect the aforementioned statistics, since these are currently supported by the cost-based optimizer we are using (see Section 5.2).

4.4 Discussion

Shortcomings of pilot runs Despite the significant advantages of the pilot runs that we already discussed in this section, there are still cases for which they may not be sufficient for accurately estimating the costs of the join operators in a plan:

- The joins are not on primary/foreign keys.
- The query contains complex predicates/UDFs that are not local but are applied on results of joins (hence, they cannot be pushed down and are not considered by PILR).
- There are correlations among the join columns.

As we explain in the following section, for these cases it is beneficial not only to perform the pilot runs, but also to adapt the execution plan at runtime, as parts of it get executed.

Placement of predicates in the plan In our approach, in case of multiple predicates per relation, we do not consider an order in their application, although this could have a considerable impact on the execution time. Moreover, the heuristic of pushing selections/UDFs down does not always lead to optimal plans, when expensive predicates are present [24, 11]. Unlike our approach, these works assume that the user provides the selectivity estimation of such predicates, and then focus on finding their optimal place in the execution plan. In practice, such selectivity is difficult to compute, as it also depends on the data, and this is exactly what we try to tackle with the pilot runs. Along the same lines, a recent development in the Stratosphere system [25] focuses on the reordering of expensive predicates, based either on user annotations or static code analysis. We see these works as complimentary to ours. Once we compute the selectivity of a predicate/UDF using the pilot runs, we can use these algorithms to find the right place and order to evaluate those predicates. For instance, if a predicate involving UDFs turns out to be expensive and not selective, we could reverse our decision to push it down, and pull it back up in the query plan.

[6]We use the superscript ϵ to highlight that a value is an estimation.

Algorithm 2: Dynamic Optimization Algorithm (DYNOPT)

Input : join block expression jb, a set of relations $\mathcal{R}$
Output: $jb(\mathcal{R})$

```
1 while true do
2     bestJoinPlan ← optimize(jb)
3     mapredPlan ← toMapReduce(bestJoinPlan)
4     jobsToRun ← pickLeafJobs(mapredPlan)
5     results ← execute(jobsToRun)
      // If the whole current plan will be
         executed, exit and return result
6     if mapredPlan \ jobsToRun = ∅ then return result
7     updateStatistics()
8     jb ← updatePlan(jb, jobsToRun, results)
```

5. DYNAMIC EXECUTION OF PLANS

As described in Section 3, after performing the pilot runs and gathering the initial statistics, we start the execution of the query's join blocks. In the first iteration, we use the statistics collected in the pilot runs, and find the best join order. After we execute a subset of the joins, we re-optimize the remaining part of the join block using the statistics collected in this execution step, and dynamically adapt the query plan as needed. This corresponds to the loop including steps 4'-7' in Figure 1. In this section we present the corresponding DYNOPT algorithm.

5.1 The DYNOPT Algorithm

Our DYNOPT algorithm (Algorithm 2), takes a join block jb and a set of relations $\mathcal{R}$, and returns $jb(\mathcal{R})$, i.e., the result of evaluating jb on $\mathcal{R}$. It iteratively executes parts of jb, dynamically adapting the query plan as needed, until the whole jb is executed.

At each iteration, we first optimize the current join block using a cost-based optimizer (line 2), which we detail in Section 5.2. Interestingly, when we optimize the current join block, we do not need to consider its local predicates, since local predicates have already been accounted for in its input statistics. The first iteration uses the statistics gathered in the pilot runs, and optimizes the join block where each node is a base relation. In subsequent iterations, the nodes in the join block are the results of previous steps and the statistics are collected during the execution of the jobs that compute the new nodes. Hence, optimizing a join block is tantamount to performing join enumeration, choosing the best join method for each join and picking the join plan with the minimum cost. As an example, the join block corresponding to query Q_1 (introduced in Section 4.1) that gets processed by the cost-based optimizer is the following Q_1':

```
SELECT rs.name
FROM restaurant' rs, review' rv, tweet' t
WHERE rs.id=rv.rsid AND rv.tid=t.id
```

In Q_1' the predicates have been removed, and each relation R is replaced by a new (virtual) relation R' that is the result of applying its local predicates. Although we do not materialize R', the statistics given to the optimizer correspond to R'.

Once the best plan is chosen by the cost-based optimizer, it gets transformed to a workflow of MapReduce jobs (line 3). Then, based on an *execution strategy*, we choose the leaf job(s) of the plan that will be executed in this iteration (lines 4-5). We discuss execution strategies in Section 5.3. During job execution, statistics are collected for specific attributes (details in Section 5.4). If there are no more jobs to execute, we output the result of the last job (line 6). Otherwise, we update the join block by substituting the part of the plan that just got executed with the results of the jobs' execution (line 8), and start the next iteration. For instance, in Figure 2, in the first iteration we execute the job corresponding to the circled part of plan_1 and save its results in relation $\texttt{t}_1$. After re-invoking the optimizer, we get plan_2, in which the previously executed partial plan is substituted with $\texttt{t}_1$.

In our current implementation, we re-optimize in each iteration, since the added overhead is not significant (see Section 6.2). However, the decision to re-optimize could be conditional on a threshold difference between the estimated result size and the observed one.

Executing the whole query For a given query q, DYNOPT gets invoked as many times as the number of join blocks in q. If there are grouping and ordering operators in the query, they get executed after the join that they depend on. Grouping and ordering operators are inserted into the execution plan by the Jaql compiler, hence are not considered by our cost-based optimizer, which focuses on join optimization. When executing the join blocks through DYNOPT, we need to respect the dependencies between blocks: a block can be executed only after all blocks it depends on have already been executed (or when it does not depend on other blocks). As long as dependencies are respected, the exact order of execution of the join blocks does not matter. However, if there are additional available resources in the cluster, there are cases where we can execute join blocks in parallel (by simultaneous invocations of DYNOPT for different blocks). This inter-block parallelism resembles the inter-operator parallelism of a bushy plan that is discussed in Section 5.3.

5.2 Cost-Based Join Optimizer

We built the cost-based optimizer that we use for join enumeration, as discussed in Section 5.1, by extending the Columbia optimizer [12]. Columbia is an open source top-down optimizer, based on the Cascades query optimizer framework [21], focusing on extensibility and efficiency. It takes as input a query, written in the form of a logical operator tree and outputs the minimum-cost physical operator tree, along with the cost of each operator.

For the enumeration of equivalent plans and their translation to executable operators, it relies on two types of rules: transformation and implementation ones. The former transform a logical expression to an equivalent one, whereas the latter replace a logical operator with a physical one. Each rule has a priority, which defines the order of application of the rules. A branch-and-bound algorithm is used to search through the alternative plans for a given query, and the execution cost of each plan is calculated using statistics for the relations, selectivity estimations and cost formulas. The supported statistics are the ones discussed in Section 4.3.

Since we are only interested in join enumeration, we deactivated all other logical operators but the binary join and the scan. The selectivity estimation of a join is computed based on the typical relational formulas [35]. Notice that Columbia is capable of producing bushy plans, whenever those are the cheapest ones. This is a desired feature that enables us to execute parts of the plan in parallel, as will be discussed in Section 5.3. However, the physical join operators that Columbia supports (hash-join, merge-join, etc.) are different from the two join operators that Jaql's runtime supports, namely the repartition ($\bowtie^r$) and broadcast ($\bowtie^b$) joins. Thus, we deactivated the physical joins of Columbia and added two new ones. Given two relations R and S, the cost of the repartition join consists of the cost of reading the relations, reshuffling them to the reducers, doing the join in the reducers, and outputting the results. Hence, the cost of the join is mostly linear in the size of the inputs (plus the cost of outputting the result). The cost formula we use is the following, where c_{rep}, c_{out} are constants, with $c_{rep} > c_{out}$:

$$C(R \bowtie^r S) = c_{rep}(|R| + |S|) + c_{out}|S \bowtie R|$$

As for the broadcast join, assuming S is the small relation, its cost includes loading S and building the corresponding hashtable,

reading R and probing each tuple to the hashtable, and finally outputting the results. We use the following cost formula, where $c_{probe}, c_{build}, c_{out}$ are constants, with $c_{probe} > c_{build} > c_{out}$:

$$C(R \bowtie^b S) = c_{probe}|R| + c_{build}|S| + c_{out}|R \bowtie S|$$

Importantly, $c_{rep} >> c_{probe}$, which reflects the fact that $C(R \bowtie^r S) > C(R \bowtie^b S)$, as long as S fits in memory. Note that although the formulas rely only on the size of the relations and not on the characteristics of the cluster or the available resources, they serve the basic purpose of *favouring broadcast joins over repartition joins*. When applicable, broadcast joins are usually preferable, since they get executed in map-only jobs and can also be chained, decreasing the total number of map-reduce jobs needed for a query.

As discussed in Section 2.2.2, Jaql is capable of chaining broadcast joins. To do so, it solely relies on the size of relations in the file system, without taking into account the filters that may be applied prior to the join. To avoid relying on this heuristic rule, we added a new rule to our optimizer that takes as input the maximum memory M_{max} that will be available during execution, and dictates which joins should be chained. Assume we want to join relations $R, S_1, S_2, \ldots, S_k$, and that all $S_i, i = 1, \ldots, k$ fit in memory at the same time. If we chain the k broadcast joins, we avoid the cost of writing to and reading from disk the $k-1$ intermediate results of the joins. The corresponding formula is the following:

$$C((R \bowtie^b S_1) \bowtie^b \ldots) \bowtie^b S_k) = c_{probe}|R| + c_{build}(|S_1| + \ldots + |S_k|) + c_{out}|R \bowtie S_1 \bowtie \ldots \bowtie S_k|$$

Our new rule traverses the operator tree and when it discovers two or more consecutive broadcast joins whose build sides fit in memory, it updates the cost of the first join of the chain using the above formula, setting the costs of the other joins of the chain to zero. Jaql's compiler has also been modified accordingly, in order to use the chain information during the translation to MapReduce jobs.

5.3 Execution Strategies: Choosing Leaf Jobs

Bushy plans provide the opportunity to execute multiple leaf joins in parallel. Various strategies can be used for choosing which joins to execute at the same time. For instance, in plan_1 of Figure 2, one could execute first the leftmost (circled) job j_1 or the repartition join j_2 between `l` and `p` or both of them in parallel.

In static optimization where the query plan does not change, the execution order of the (leaf) jobs does not impact query execution time. In this case, we only need to determine which jobs to run in parallel, as these jobs will strive for the same cluster resources [20].

On the contrary, when the execution plan can change at runtime, deciding which leaf job(s) to execute first can have significant impact on execution time, since it determines the way the plan dynamically adapts. For instance, in the aforementioned plan_1, executing first j_2 instead of j_1 would lead to a different plan than plan_2 of the figure. Even more interestingly, executing more than one job at a time has the additional side effect of changing the number of re-optimization opportunities. For example, if we execute j_1 and j_2 in parallel, we will have one less re-optimization point, since we will only re-optimize at the end of the two jobs. Such an order of application can be determined using an *execution strategy*. To the best of our knowledge, we are the first to consider such strategies, which can also be applicable in the relational setting.

There are two dimensions that an execution strategy has to consider: 1) give a priority to each leaf job (higher priority jobs get executed first), which changes as parts of the plan get executed; 2) decide how many jobs to execute at a time. Different metrics for deciding the priority of the leaf jobs can be devised. Two interesting ones are the following:

Cost The basic idea is to favour the cheapest jobs, so that we reach re-optimization points as soon as possible.

Uncertainty With the notion of uncertainty of a job, we want to reflect the expected error in the result size estimation. Note that we use standard join selectivity formulas, and it has been shown that the estimation error increases exponentially with the number of joins [27]. As a result, we define uncertainty as the number of joins that participate in the job. The idea here is that we want to execute uncertain jobs first to gather actual statistics about them, and fix the remaining plan through re-optimization if needed.

By combining the above dimensions we can get several strategy variants. Our experiments in Section 6.3 include a comparison among different strategies. As future work, we plan to determine the most appropriate strategy for each query based on a cost model.

5.4 Online Statistics Collection

During the execution of a MapReduce job, we collect statistics either at the map phase (if it is a map-only job) or the reduce phase (if it is a map-reduce job). We keep the same statistics as the ones for the pilot runs (see Section 4.3). To minimize the statistics collection overhead, we keep statistics only for the needed attributes for re-optimization, i.e., the ones that participate in join conditions of the still unexecuted part of the join block. Moreover, we do not keep statistics if we know we are not going to re-optimize (e.g., during the last query job) or if statistics for the given job are already in the metastore (see Section 4.1 for reusability of statistics).

For each MapReduce job, we first store in the job configuration the list of attributes we will collect statistics for. Assume we execute a map-only job. During execution, each task collects statistics for its input split. Instead of relying on an extra MapReduce job to combine these partial statistics and get the global ones, every time a task finishes it writes its statistics in a file and publishes the file's URL in ZooKeeper. Once all tasks are finished, the Jaql client that submitted the job finds through ZooKeeper the URLs of all partial statistics files, reads and combines them. The same technique is followed when collecting statistics in the reduce phase.

6. EXPERIMENTAL EVALUATION

In this section, we present our experimental results for DYNO.

6.1 Experimental Setup

Cluster setup We performed our experiments on a cluster of 15 nodes, connected with 10 gigabit Ethernet. Each machine has two 2.2 GHz Intel Xeon E5-2430 6-core CPUs, with 96 GB of RAM and 12 SATA disks of 2 TB each. We deployed Hadoop 1.1.1 with each node having 10 map and 6 reduce slots (leading to a total of 140 map and 84 reduce slots). Each slot was given 2 GB of RAM. HDFS was configured to stripe across all disks of each node (excluding one disk dedicated to the OS), and the HDFS block size was set to 128 MB. For the number of map and reduce tasks per MapReduce job, we used the same values that Hive uses by default. All machines are running Ubuntu Linux 12.04 (with kernel version 3.2.0), Java 6 and ZooKeeper 3.4.5.

Datasets There are not many big data benchmarks, and none is suitable for our study, which requires multiple joins. The Pavlo et al. benchmark [33] has only two relations to be joined, the YCSB benchmark [13] targets NoSQL systems, and the data generators for BigBench [19], which would be a good candidate, were not publicly available at the time of writing this paper. As a result, we settled on TPC-H [37], which provides a data generator that allows us testing at different scales, and contains 8 tables and queries with

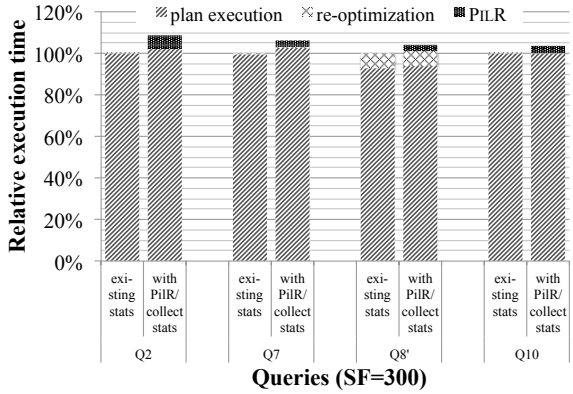

Figure 4: Overhead of pilot runs, re-optimization and statistics collection in DYNOPT.

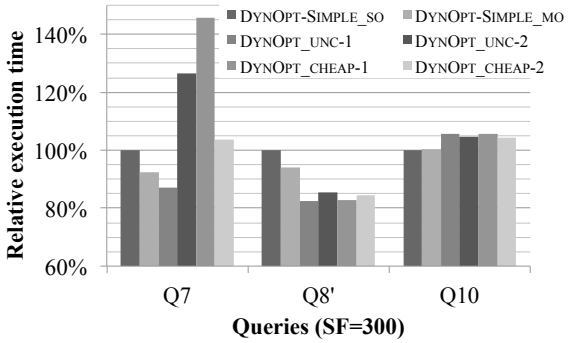

Figure 5: Comparison of execution strategies for DYNOPT.

many joins. We generated datasets of size 100 GB, 300 GB and 1 TB (by using scale factors SF={100,300,1000}, respectively).

Queries From the 22 TPC-H queries, we chose those that include joins between at least 4 relations, namely queries Q2, Q7, Q8, Q9, Q10. We excluded Q5, because it contains cyclic join conditions that are not currently supported by our optimizer. To better assess the performance of DYNO, we used modified versions of Q8 and Q9. In particular, we added a UDF in Q8 that filters the results of the join between `orders` and `customer`. We also added two correlated predicates on `orders` (correlations were identified using the CORDS algorithm [26]). In Q9 we added various filtering UDFs on top of the dimension tables to make them fit in memory. Depending on the experiment, for better readability, we may include only the subset of the above queries that brings interesting insights.

Variants of execution plans We used the following variants of execution plans to evaluate the characteristics of our system:

DYNOPT This is the dynamic execution of a query as dictated by our DYNOPT algorithm (Algorithm 2, described in Section 5). It includes the pilot runs, online statistics collection and re-optimization points.

DYNOPT-SIMPLE This is the plan we get by removing the re-optimization points from DYNOPT, that is, we perform the pilot runs, call our optimizer once and then execute the resulting plan. Apart from the pilot runs, no runtime statistics are collected.

RELOPT Here we employ a state-of-the-art relational optimizer used in a commercial shared-nothing parallel database DBMS-X, which also supports repartition and broadcast joins. DBMS-X is capable of using more detailed statistics (e.g., histograms), and all needed statistics are gathered by the system prior to query execution in this case. The obtained execution plan is then hand-coded to a Jaql script and executed. Note that DBMS-X does not have enough information to estimate selectivity of UDFs during optimization. We used this plan as a representative of the best plan that can be obtained by an existing optimizer.

BESTSTATICJAQL As explained in Section 2, the existing version of Jaql produces only left-deep plans and the join ordering is determined by the order of relations in the FROM clause. For each query, we tried all possible orders of relations and picked the best one; BESTSTATICJAQL is the resulting plan. The rewrite rule for using broadcast joins when relations are small is activated.

In our experiments, we used *k=1024* for the KMV synopsis to estimate distinct values (leading to a worst case error bound of 6%). Moreover, we disabled the optimization that avoids executing the pilot runs when there are already statistics computed for a given expression. All numbers reported below are averaged over 3 runs.

6.2 Overhead of Dynamic Optimization

In this section we assess the overhead introduced to Jaql execution due to our dynamic optimization techniques, namely the pilot runs, the statistics collection during the execution of the MapReduce jobs, and the calls to the optimizer at each (re-)optimization point. To this end, we report execution times of four TPC-H queries for two different executions. In the first one, we provide upfront all needed statistics to the optimizer, and the only overhead is the (re-)optimization time. To do so, we previously execute the query to add to the metastore all needed statistics, and then re-execute it after enabling the statistics reusability optimization. In the second, all statistics are computed at runtime (through pilot runs and online statistics collection). The overhead of the (re-)optimization and the pilot runs are explicitly measured. The overhead of online statistics collection can be computed by the difference in execution time between the two executions, after the execution time of pilot runs and total (re-)optimization time is deducted. The results are given in Figure 4, where execution times are normalized to the execution time with pre-collected statistics.

As can be seen, total (re-)optimization time is less than 0.25% of the execution time in most cases. It is only for Q8′ that it is approximately 7% as this query includes a 7-way join. Note that although Q8′ gets (re-)optimized four times, only the initial call to the optimizer (after the pilot runs) is costly, as it includes all 8 relations and is responsible for 90% of the total (re-)optimization time. The subsequent calls are very fast, because, as part of the query has been executed, less relations are included in the optimization.

Furthermore, the overhead of the pilot runs is between 2.5% and 6.7%. Online statistics collection brings a small overhead of 0.1% to 2.8% to the total execution time, depending on the number of attributes for which we need to keep statistics for. For instance, Q10 that has the smallest number of join conditions, also has the smallest statistics collection overhead. Overall, we observe a total of 7-10% overhead. We believe this is acceptable given the benefits brought by our approach, as will be shown in Sections 6.4 and 6.5.

6.3 Comparison of Execution Strategies

In this experiment, we compare various execution strategies both for DYNOPT and DYNOPT-SIMPLE (see Section 5.3 for a related discussion). For DYNOPT-SIMPLE, we use two variants: SO executes only one leaf job at a time, and MO submits all leaf jobs at the same time. For DYNOPT, we use: UNC-1 that executes the single most uncertain leaf job first; UNC-2 that executes the two cheapest most uncertain leaf jobs at the same time (if two leaf jobs exist in the query plan, otherwise one); CHEAP-1 that executes the cheapest leaf job first; CHEAP-2 that executes the two cheapest leaf

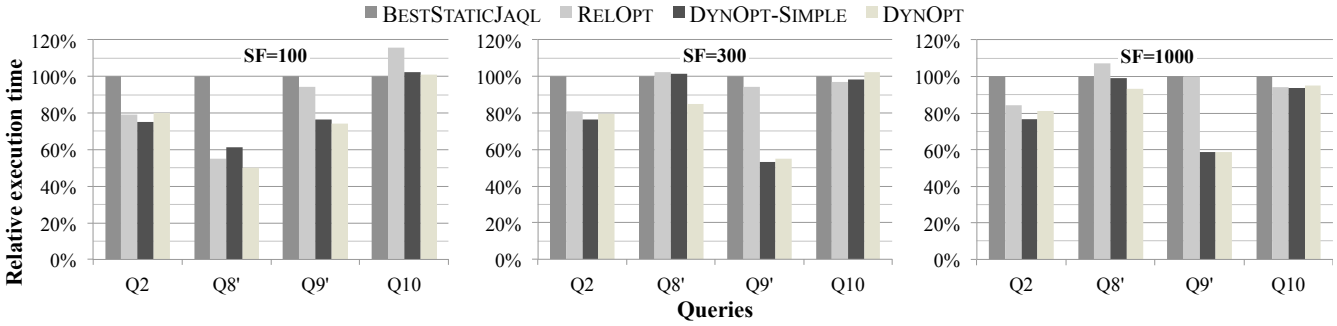

Figure 7: Speedup of execution times for varying queries and scale factors compared to default Jaql execution.

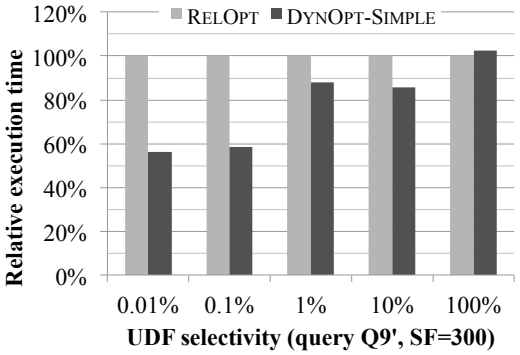

Figure 6: Performance impact of UDF selectivity on Q9′.

jobs first. When executing two jobs j_1 and j_2 in parallel, we use the FIFO scheduler of Hadoop, so as to maximize the utilization of the cluster resources. As future work, we plan to experiment with different schedulers, such as the fair and capacity schedulers. The resulting execution times, normalized to the execution time of DYNOPT-SIMPLE_SO for each query, are given in Figure 6.

As expected, SIMPLE_MO always outperforms SIMPLE_SO, since the former utilizes the cluster better (e.g., when a job is at its reduce phase, another job can start its map phase). The only case the two strategies coincide is for Q10, because a left-deep plan is picked by the optimizer, thus there is no chance for parallelization.

Interestingly, for DYNOPT, parallelization is not always beneficial: executing jobs in parallel increases cluster utilization, but reduces the re-optimization points. For Q10, due to the chosen left-deep plan, all strategies perform the same. However, for Q7 and Q8′, UNC-1 is the winner strategy. For Q7, UNC-1 results in 40% better time over UNC-2 and 13% over SIMPLE_SO. Moreover, executing the most uncertain jobs first is much better than executing the cheapest ones for Q7. In the case of Q8, the cheapest and most uncertain jobs coincide, thus the variants perform the same. Overall, *it is more beneficial for* DYNOPT *to have more re-optimization points than to execute more jobs in parallel*. Due to their better performance, hereafter we will use the MO variant for DYNOPT-SIMPLE, and the UNC-1 variant for DYNOPT.

6.4 Star Join Sensitivity Analysis

In this section we study the performance of DYNOPT-SIMPLE for star join Q9′, as the size of the dimensions changes (by changing the selectivity of the UDFs), in order to see the effectiveness of pilot runs. The results are given in Figure 6, where the execution times are normalized to the execution time of the plan picked each time by RELOPT.

In all but the last case (100% selectivity), all dimensions fit in memory and all joins are broadcast joins. However, not all dimensions fit in memory at the same time, hence the query is executed in multiple jobs. For 0.01% and 0.1% selectivity, all joins are executed in 2 map-only joins (since more joins can be chained), leading to a performance improvement of 1.78x and 1.71x over RELOPT, respectively. For 1% and 10% selectivity, 3 map-only jobs are needed, bringing a (smaller) performance speedup of approx. 1.15x over RELOPT. Finally, when the selectivity is 100%, our optimizer picks the same plan with RELOPT (including 4 jobs), and our execution is slightly worse due to the pilot runs overhead.

The pilot runs enable the optimizer to identify the cases when tables can fit in memory for a broadcast join. Note that if the optimizer's estimate is incorrect and the build table turns out to not fit in memory, the query may not even terminate. As a result, most systems are quite conservative and favour repartition joins to avoid disastrous plans. However, repartition joins lead to worse performance if the table is small enough to fit in memory. Our pilot runs enable making this decision more accurately, avoiding both disastrous cases, as well as not missing good performance opportunities when the build table is indeed small enough to fit in memory. As future work we plan to experiment with more advanced implementations of the broadcast join (e.g., taking advantage of the DistributedCache) to further boost our performance speedup.

6.5 Comparison of Query Execution Times

We now compare the performance of the four execution plans described in Section 6.1 for various queries and scale factors. Figure 7 shows the query execution times normalized to the BESTSTATICJAQL execution time.

First, notice that *both our dynamic techniques (*DYNOPT-SIMPLE *and* DYNOPT*) are at least as good as, and sometimes better than, the best hand-written left-deep plan (*BESTSTATICJAQL*) for all queries and scale factors.* This is not always the case for RELOPT (e.g., Q10 for SF=100, Q8′ for SF=1000). Our techniques produce plans that are also better than the plans generated by the RELOPT in most cases, except Q2 and Q8′ for SF=100, and Q10 for SF=300. Moreover, there are cases for which we are up to 2x faster than BESTSTATICJAQL.

Q2 benefits by considering bushy plans: for all three bushy plans (RELOPT, DYNOPT-SIMPLE, DYNOPT) and all scale factors, there is a performance speedup of approximately 1.2x. This confirms that in a shared-nothing architecture it is crucial to consider bushy plans (to minimize intermediate results) during query optimization. Moreover, given there are no UDFs and joins are on primary/foreign keys, re-optimization does not bring further benefit (in fact, there is a slight degradation due to the added overhead).

Q8′ includes a UDF over the result of a join and in this case re-optimizing proves to be very beneficial: DYNOPT leads to a performance speedup of 2x, 1.17x and 1.07x over BESTSTATICJAQL for the three scale factors. The benefits of re-optimization are not as

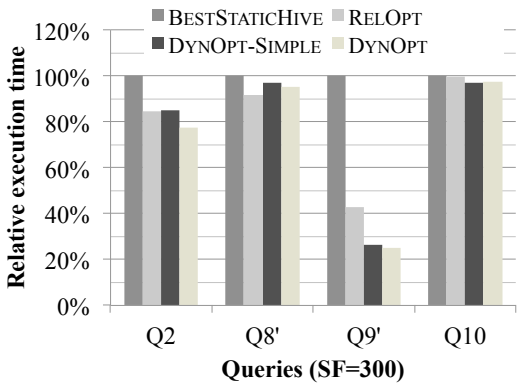

Figure 8: Benefits of applying DYNOPT in Hive.

pronounced for larger scale factors, because as data size increases, there are less opportunities for applying broadcast joins. On the other hand, RELOPT and DYNOPT-SIMPLE manage to speed-up Q8′ only for SF=100. In the case of DYNOPT-SIMPLE, only pilot runs are not sufficient to account for the non-local UDF.

Q9′ is the star join case discussed in Section 6.4. Due to the presence of UDFs, RELOPT does not manage to improve the performance of the query. On the other hand, our techniques manage to make extensive use of broadcast joins, leading to speedups of 1.33-1.88x (depending on the SF). Notice that for larger data sizes, avoiding repartition joins is even more important.

Finally, for Q10, the left-deep BESTSTATICJAQL is very close to the best plan, so performance does not further improve with bushy plans or dynamic optimizations. Notice, however, that even in that case our techniques are as good as the best hand-written plan.

6.6 Benefits of DYNOPT in Hive

As discussed in Section 3, our techniques can be applied to other systems. To show the benefits of using the execution plans produced by our dynamic techniques on a system other than Jaql, we compare the performance of the four execution plans described in Section 6.1 for various TPC-H queries and scale factor SF=300 using Hive. Note that instead of using the BESTSTATICJAQL plan, we use the BESTSTATICHIVE one, that is, the best left-deep for Hive[7]. We chose Hive for this experiment since it is currently one of the most popular systems for SQL queries over HDFS data.

We took the execution plans of DYNOPT and DYNOPT-SIMPLE that were generated by Jaql for the experiment of Section 6.5 (for SF=300) and hand-coded them in Hive. Note that if we had implemented the same sampling and statistics collection and used our join optimizer in Hive, the same plans would have been generated. Figure 8 shows the query execution times for these plans. These numbers do not include the overheads of our techniques, which are typically about 5% of the execution time (see Section 6.2).

We observe similar trends as the ones for Jaql. DYNOPT and DYNOPT-SIMPLE always produce better plans than BESTSTATICHIVE. It is interesting to note that Q9′ has a significantly higher speedup than Jaql (3.98x instead of 1.88x). This is because broadcast joins in Hive 0.12 exploit the Distributed Cache of MapReduce, leading to better performance. This brings an advantage for queries such as Q9′ that include many broadcast joins.

6.7 Discussion

The following are the highlights of our experiments:

- DYNOPT-SIMPLE always produces better plans than the best left-deep plan. Moreover, it is always comparable and most of the times better than the RELOPT plan (with the exception of Q8′ for SF=300). This shows that although it introduces a small overhead, it provides accurate selectivity and result size estimation, especially when the query contains local UDFs and data correlations, as exhibited by Q9′.
- DYNOPT subsumes DYNOPT-SIMPLE. Hence, when further re-optimization is not beneficial, DYNOPT is a bit worse than DYNOPT-SIMPLE due to the increased overhead it introduces. However, it outperforms DYNOPT-SIMPLE when the join result size estimation contains high uncertainty, due to correlations, UDFs on results of joins (this is the case for Q8′), or non primary/foreign key join conditions.
- There are queries for which the left-deep plan is the optimal (see Q10), but there are others that can benefit significantly by considering bushy plans (such as Q2).
- Our Hive experiments show that our techniques are also beneficial for other systems, exhibiting similar performance trends.

Overall, notice that we get the most benefit from pilot runs, which is an interesting remark as most dynamic optimization techniques so far have focused on re-optimizing the query at runtime and not on carefully selecting the first plan before the first re-optimization occurs.

7. RELATED WORK

Several higher-level declarative languages have been proposed lately for analyzing large datasets, such as Pig [18], Hive [36], Jaql [5], Stratosphere [3], DryadLINQ [41], SCOPE [8], Spark [42] and Shark [40]. Query optimization in this environment is crucial for translating such higher-level scripts to efficient distributed execution plans. However, most of these systems lack a cost-based optimizer and rely on heuristics, e.g., Hive and Pig [17].

Dynamic techniques are natural in this setting and have already started to be adapted by some of the systems. For instance, Shark can switch from a repartition to a broadcast join at runtime. In RoPE [1] recurring queries get optimized based on observations from their previous executions. The most related work to ours is in the context of SCOPE [8], where statistics are collected at runtime and the execution plan gets re-optimized if needed. One difference is that SCOPE relies on Dryad, which does not provide the natural re-optimization points of Hadoop. Moreover, they do not consider a way of automatically costing UDFs before choosing the initial plan (as we do with the pilot runs). Work has also been done on optimizing directly MapReduce jobs [30], by selecting the correct job configuration based on the characteristics of a job or by merging MapReduce jobs whenever possible. We consider such works as complimentary to ours: after our optimizer has picked an execution plan and our compiler has transformed it to a MapReduce workflow, we can use these works to further optimize the workflow.

Adaptive query processing has been extensively studied in the relational setting [15]. Altering the execution plan at runtime has been considered both in centralized [28, 31, 2] and distributed scenarios [22]. One of the important challenges in these works is deciding when to block query execution and re-optimize the plan (possibly throwing away work). Moreover, the initial plan in these works is obtained from the optimizer using the base table statistics, and as such is subject to potential incorrect result size and cost estimation due to data correlations and UDFs, leading to a possibly suboptimal plan (until re-optimization is invoked).

As far as dealing with complex predicates and UDFs is concerned, most existing works [24, 11] focus on placing such predicates in the right order and position in the plan, given that the selectivity of the predicate is provided. In [25], static code analysis and

[7]https://issues.apache.org/jira/browse/HIVE-600

hints are used to reorder UDFs. Few works have focused on UDF cost models that get adapted based on previous executions [29, 23]. To the best of our knowledge, we are the first to consider calculating the UDF selectivity at runtime through the pilot runs.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented DYNO, which utilizes pilot runs, runtime statistics collection, and re-optimization to optimize complex queries over Hadoop data. We introduced pilot runs, which execute the local filters and UDFs over a small sample of each base relation to collect statistics and estimate result sizes. Pilot runs are especially beneficial when the query contains UDFs and data correlations, because traditional static optimizers fail at estimating result sizes correctly under these conditions. Underestimating result sizes can lead the optimizer to make the wrong decision and use a broadcast join, which can be disastrous with a large data set. We provide the statistics collected via pilot runs to a cost-based optimizer to choose the join order and methods. As the query runs, we collect statistics, and at the completion of each MapReduce job, we can re-optimize the remaining parts of the query. Re-optimization is especially beneficial when 1) there are UDFs that need to be applied on the result of a join, and can impact the intermediate result size, 2) the join condition is not on primary/foreign key relationship, and 3) there is data correlation between the join conditions.

We showed that the plans produced by our system, both DYNOPT and DYNOPT-SIMPLE, are at least as good and often better than the best hand-written left-deep plans, and are most of the times better than the plans produced by a state-of-the-art relational optimizer for a shared nothing DBMS. Finally, we identified interesting strategies for choosing leaf jobs to execute at a given point, and proposed some first heuristic choices. Experimenting with more sophisticated (possibly cost-based) strategies is left as future work.

We are also planning to create a new dynamic join operator that switches between a broadcast and repartition join, without waiting for the current job to finish. Moreover, we consider adapting our techniques to other platforms, such as Tez. Although our optimizer currently focuses on join ordering, we plan to extend it to consider grouping and ordering operators.

9. ACKNOWLEDGMENTS

The authors would like to thank Guy Lohman and Ippokratis Pandis for their useful discussions and insightful feedback.

10. REFERENCES

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *NSDI*, 2012.

[2] S. Babu, P. Bizarro, and D. J. DeWitt. Proactive re-optimization. In *SIGMOD Conference*, pages 107–118, 2005.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[4] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. R. Jr. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

[5] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12), 2011.

[6] K. S. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*, pages 199–210, 2007.

[7] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[8] N. Bruno, S. Jain, and J. Zhou. Continuous cloud-scale query optimization and processing. In *VLDB*, 2013.

[9] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[10] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD Conference*, 2004.

[11] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[12] Columbia Query Optimizer. http://web.cecs.pdx.edu/ len/Columbia.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[15] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[16] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[17] A. Gates, J. Dai, and T. Nair. Apache Pig's optimizer. *IEEE Data Eng. Bull.*, 36(1):34–45, 2013.

[18] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of MapReduce: The Pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[19] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, pages 1197–1208, 2013.

[20] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[21] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[22] W.-S. Han, J. Ng, V. Markl, H. Kache, and M. Kandil. Progressive optimization in a shared-nothing parallel database. In *SIGMOD*, pages 809–820, 2007.

[23] Z. He, B. S. Lee, and R. R. Snapp. Self-tuning cost modeling of user-defined functions in an object-relational dbms. *ACM Trans. Database Syst.*, 30(3):812–853, 2005.

[24] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM TODS*, 23(2):113–157, 1998.

[25] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[26] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[27] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD Conference*, pages 268–277, 1991.

[28] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD Conference*, pages 106–117, 1998.

[29] B. S. Lee, L. Chen, J. Buzas, and V. Kannoth. Regression-based self-tuning modeling of smooth user-defined function costs for an object-relational database management system query optimizer. *Comput. J.*, 47(6):673–693, 2004.

[30] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *PVLDB*, 5(11):1196–1207, 2012.

[31] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[32] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large MapReduce jobs. *PVLDB*, 4(11):1135–1145, 2011.

[33] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[34] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD*, pages 39–48, 1992.

[35] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[36] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[37] TPC-H Benchmark. http://www.tpc.org/tpch.

[38] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using situation-aware mappers. In *EDBT*, pages 420–431, 2012.

[39] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, page 12, 2011.

[40] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.