# Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems

[Research Highlights]

Christina Pavlopoulou
Univ. of California, Riverside
cpavl001@ucr.edu

Michael J. Carey
Univ. of California, Irvine
mjcarey@uci.edu

Vassilis J. Tsotras
Univ. of California, Riverside
tsotras@cs.ucr.edu

## ABSTRACT

Effective query optimization remains an open problem for Big Data Management Systems. In this work, we revisit an old idea, *runtime dynamic optimization*, and adapt it to a big data management system, AsterixDB. The approach runs in stages (re-optimization points), starting by first executing all predicates local to a single dataset. The intermediate result created by a stage is then used to re-optimize the remaining query. This re-optimization approach avoids inaccurate intermediate result cardinality estimates, thus leading to much better execution plans. While it introduces overhead for materializing intermediate results, experiments show that this overhead is relatively small and is an acceptable price to pay given the optimization benefits.

## 1. INTRODUCTION

Query optimization is a key aspect of a database system, as it determines the order of execution for query operators along with the operators' physical algorithms. One of the most demanding operators is the Join, which can be implemented in different ways depending on the sizes of its inputs and outputs. To tackle the join optimization problem, two basic approaches have been introduced.

The first approach (introduced in System R [9]) is *cost-based query optimization*; it performs an exhaustive search (via dynamic programming) among all join orderings until the one with the smallest cost is found and eventually executed in a pipelined fashion. The second approach (introduced in INGRES [38]) instead takes a *runtime dynamic query optimization* approach (later known as Adaptive Query Processing (AQP)), where the original query is decomposed into single-variable (single dataset) subqueries that are executed separately. This takes place by: (1) breaking off components of the query which participate with a single variable (e.g., selections), and (2) implementing joins by substituting one of the join variables with a tuple-at-a-time value. Each subquery result is stored as a new relation that is then considered by the optimizer in optimizing the remaining query. The choice of the "next" subquery to be executed is based on the cardinality of the participating datasets.

The INGRES approach was a greedy, cardinality-based technique, with overhead due to creating indexed (for joins) intermediate results, and the cost-based, compile-time approach of System R became the preferred approach for many years [28, 34, 19, 21, 20, 17, 36]. To assign a cost for each plan (and thus find the best join order and operator algorithms) the cost-based approach depends heavily on statistics. Its accuracy is greatly affected by the existence of multiple selection predicates (on a single dataset), *complex* selection predicates (with parameters or UDFs) and join conditions that are not based on key-foreign key relationships. In such cases, statistics can be misleading, resulting in inaccurate join result estimates. As the number of joins increases, the error worsens as it propagates to future join stages [24]. These issues are exacerbated in today's big data management systems (BDMS) by the sheer volume of data.

In this work, we revisit the INGRES runtime dynamic approach and adapt it (with modifications) to a shared-nothing distributed BDMS (AsterixDB), and experiment (using TPC datasets) with various workloads to evaluate its performance in a real BDMS. With an increased volume of data, even small errors in join ordering can lead to *very expensive* execution plans. A characteristic of the original dynamic optimization approach was that the next subquery to execute was chosen based only on dataset cardinalities. However, the cost-based optimization approach showed that, for good join result estimation, better statistics are needed. Here we take advantage of the materialization stages to collect all needed statistics. This combination of re-optimization and statistics collection leads to superior execution plans.

In our approach, when a multijoin query is executed, all predicates local to a table are pushed down and executed first to gather updated accurate statistics. Intermediate results and updated statistics are fed back to the optimizer to choose the cheapest first join to execute. This process is repeated until only two joins remain; then one of the remaining joins is picked as the best, leaving one final join to process, for which there is no need to collect further statistics. Note that at each stage the optimizer *simply* picks the next join using the statistics collected from the previous stage. We have integrated this approach in AsterixDB [1, 8] which, like many relational database systems, executes queries in a pipelined manner for efficiency. Although with our modified dynamic optimization approach query execution goes through blocking re-optimization points, the extra overhead is relatively small and worthwhile since very expensive plans

The original version of this paper is "Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems" and appeared in EDBT 2022.

are avoided.

Starting with [25], various works have used dynamic techniques to alleviate problems with static cost-based optimization [11, 18, 10, 35]. While similar to our approach, they have mostly not leveraged information coming from correlated selectivities and predicates with parameters and UDFs. Moreover, they assume a query processor with a cost-based optimizer that enumerates the space of possible query plans, augmenting it in various ways; they estimate new statistics after mid-query execution (using information gathered from intermediate results) and use the new information to re-calibrate their query plans. Our approach is architecturally simpler, being based instead on the INGRES dynamic one-step-at-a-time idea (i.e., we look again at a "road not taken"). By executing local predicates first, we gain accurate cardinality estimates early that can lead to improved query performance (despite materialization overhead). Dynamic optimization has also been explored in multi-node environments [25, 29, 26, 27, 7] (see Section 2).

We offer two main contributions in this paper. First, we adapt INGRES-inspired dynamic optimization to a shared-nothing BDMS setting. At each stage (re-optimization point), we only consider and execute the next cheapest join. Second, we assess the approach via detailed experiments that reveal its superiority compared to traditional optimizers. We also evaluate the overhead due to its multiple re-optimization points and materialization of intermediate results.

## 2. RELATED WORK

Runtime dynamic optimization first appeared in INGRES, where a query was decomposed into single-variable queries and then, based on updated cardinalities, the next query fragment to execute was chosen. Unlike INGRES, our work does not rely solely on cardinalities; we collect more information about base and intermediate data based on statistics. Since INGRES, there has been other work on runtime dynamic optimization in a single-server context. LEO [35] calibrates the original statistics using feedback acquired from historical queries and uses them to optimize future queries. In Eddies [10] the selectivity of each query operator is calculated as records are being processed; more selective operators are eventually prioritized in the evaluation order.

Dynamic optimization is more challenging in a shared-nothing environment since data is distributed across multiple nodes. Optimus [27] leverages runtime statistics to rewrite execution plans. It performs a number of optimizations but does not address multi-way joins, which as [27] points out, can be "tricky" since the data may need to be partitioned in multiple ways. RoPE [7] leverages historical statistics from prior plan executions to tune future executions, e.g. the number of reduce tasks to schedule, choosing appropriate operations, including order. Follow-up work [15] extended RoPE's approach to general query workloads in Scope [40]. That strategy generates a full initial query plan from historical statistics, but collects fresh statistics during execution that it uses to adjust the plan's remaining operators. To avoid wasting work, reoptimization occurs after a threshold and the initial plan is based only on the base datasets, which can lead to suboptimal plans. In contrast, our approach blocks a query after each join stage is done and uses the result to optimize subsequent stages; no join work is wasted. Also, we estimate the selectivity of predicates by pushing down their execution, thereby avoiding possibly misleading initial calculations. Regardless, learning from past query executions is an orthogonal idea that could be used to further optimize our approach in the future.

Another runtime dynamic optimization approach uses *pilot runs*, as introduced in [26]. To alleviate the need for historical statistics, pilot runs of the query are run on sample data. There are two main differences from our work. First, statistics from pilot runs are not very accurate for joins that do not have a primary/foreign key condition, as sampling can be skewed in such cases. In contrast, we gather statistics on the base datasets that lead to better join result estimates for those joins. Secondly, we exploit AsterixDB's LSM ingestion process to obtain initial statistics for base datasets and employ materialization of intermediate results to get more accurate estimates, thereby avoiding the overhead of pilot runs. Another approach is ROX [4], which gathers statistics (only) on samples instead of on base datasets. In contrast to pilot runs, ROX gathers statistics on samples even for intermediate results. In our work we gather statistics on the base datasets that lead to more accurate join result estimates. Finally, Monsoon [33] uses a probabilistic model to decide if preprocessing is needed for cardinality estimation. That technique could be used to further optimize our approach when we focus on more costly predicates.

RIOS [29] also employed runtime incremental optimization. Unlike Optimus, RIOS assumes that the potential repartitioning overhead is amortized by the efficiency of their approach. Statistics are collected during a pre-partitioning stage in which all datasets in the query are partitioned according to an initial lazy plan based on their raw byte sizes. A disadvantage of the prepartitioning phase is that if the statistics collected indicate that it is not the correct plan, RIOS re-partitions the data. This is done if the difference between the lazy plan and the better one exceeds a given threshold. In that case, the remaining query is optimized according to the feedback from intermediate results. Unlike RIOS, our method avoids potentially expensive repartitioning since accurate statistics are collected before the query is analyzed by the optimizer. This way, we can choose the right join order from the outset and thereby the right partitioning scheme. Avoiding the overhead of faulty partitioning can be very important for large data volumes.

Among the aforementioned works, the closest are RIOS [29] and pilot runs [26] since neither requires the use of historical statistics. For our experiments we implemented the pilot runs approach [26] in AsterixDB (see Section 7). We did not implement RIOS [29] since its prepartition phase is quite different than the query workflow in AsterixDB, where each dataset is stored partitioned and repartitioned if needed at the first join stage that it is involved in.

## 3. ASTERIXDB BACKGROUND

Apache AsterixDB is a shared-nothing platform for managing large volumes of semistructured data. To process a query, AsterixDB compiles it into an Algebricks [13] program called a logical plan. This plan is then optimized via rewrite rules that reorder the Algebricks operators and introduce partitioned parallelism for scalable execution. After this rule-based step, a code generation step translates the resulting physical query plan into a job that uses the Hyracks engine [14] to compute the query results. The job

is distributed across the system and executed on every cluster node. AsterixDB takes a dataflow-based query execution approach, as in many parallel RDBMSs and runtimes for big data (i.e., Pig, Hive, Impala, Apache Tez, and Spark [12]).

All AsterixDB layers are involved in the integration of our work, but the query optimizer, which is mainly in the Algebricks layer, is the core focus. Currently, the AsterixDB optimizer considers many data properties, such as the data partitioning and ordering, and decides according to a set of heuristic rules (which are the core of Algebricks) how the query should be executed. These heuristic rules are applied without any statistical information. For multi-join queries, the join order in AsterixDB currently depends on the order of the datasets in the *FROM* clause of the query (i.e., datasets are joined in order of appearance). Generally, the compiler produces right-deep joins; if a user wants to generate bushy-joins, it is feasible by joining the datasets together using subqueries. This option is unpleasant for naive users.

Another aspect of join query optimization is the choice of join algorithm. AsterixDB supports several algorithms:

**Hash Join:** If the join inputs are not partitioned in a useful way, they are redistributed by hashing both on the join key(s) and doing local joins using dynamic hash join. The join's "build" side is first re-partitioned and fed over the network into the build step of a local hash join. The "probe" side is then re-partitioned similarly, yielding a pipelined-parallel dynamic hash join. If either input is already partitioned on the join key(s), re-partitioning is skipped for that input.

**Broadcast Join:** This strategy also uses a local dynamic hash join, but one (small) join input is broadcast (replicated) to all partitions of the other input. The broadcast input is the build input for the join, and when the build phase is done the participating partitions then each probe their local partition of the other (larger) input to perform the join.

**Indexed Nested Loop Join:** Here, one input is broadcast to all partitions of the other input, which is a base dataset indexed on the join key(s). As objects arrive at each partition, they are used to probe the index of the indexed dataset.

Hash join is the default unless a query hint asks the optimizer to choose one of the other algorithms. However, when a broadcast join is applicable, joins can be much faster, as expensive shuffling of the larger dataset is avoided.

**Optimizer Limitations:** The current rule-based optimizer in AsterixDB has several significant limitations. First, there is no selectivity estimation for predicates, thus missing opportunities for choosing the best join algorithms. Broadcast joins, in particular, will not be considered, even if a dataset is small enough to fit in memory after applying selective filters. Second, there is no cost-based join enumeration. The performance of a query relies largely on how it was written by the user (i.e., the dataset ordering in the FROM clause).

The work here targets AsterixDB, but any big data query platform that supports joins should be able to incorporate our techniques with the following changes: (1) instrument its query runtime to collect dynamic statistics over attributes; (2) incorporate a cost model like ours into its query optimizer; (3) create or use operators to materialize intermediate results to scan during the remainder of a query; and (4) identify potential query plan re-optimization points.

# 4. STATISTICS COLLECTION

At each re-optimization point, we collect statistical information about the base and intermediate datasets to help the optimizer pick the best join order and join algorithm. These statistics are later used to estimate the actual join result size by using the following formula, as described in [32] (where $S(x)$ is the size of dataset $x$ and $U(x.k)$ is the number of unique elements for attribute $k$ of dataset $x$):

$$A \bowtie_k B = S(A) * S(B)/max(U(A.k), U(B.k)) \quad (1)$$

The size of a dataset is the number of qualified records in the dataset immediately before the join operation. If a dataset has local predicates, result cardinality is traditionally computed by multiplying all individual selectivities [32]. As described in Section 5.1, we use a more effective approach.

**Statistics Types:** To measure the selectivity of a dataset we use quantile sketches. Following Greenwald-Khanna [37], we extract quantiles that represent the right border of a bucket in an equi-height histogram. The buckets help us identify estimates for different ranges when filters appear on the base datasets. To find the number of unique values needed for formula 1, we use Hyperloglog [30] sketches. The HLL algorithm can identify with great precision the unique elements in a stream of data. We collect these statistics for each field of a dataset that may participate in any query. The gathering of these two statistics types happens in parallel.

# 5. RUNTIME DYNAMIC OPTIMIZATION

The main focus of our approach is to utilize the collected statistics from intermediate results in order to refine the plan on each subsequent stage of a multi join query. To achieve this aim, there are several stages that need to be considered.

As described in Algorithm 1 lines 6-9, the first step is to identify all the datasets with predicates. If the number of predicates is more than one, or, there is at least one complex predicate (with a UDF or parameters), we execute them as described in Section 5.1. Afterwards, while the updated query execution starts as it would normally do, we introduce a loop which will completes when there are only two joins left in the query. In that case, there is no reason to re-optimize the query as there is only one possible remaining join order. This loop can be summarized in the following steps:

- A query string, along with statistics, are given to the **Planner** (line 12) which is responsible for figuring out the next best join to be executed (the one that results in the least cardinality) based on the initial or online statistics. As a result, the Planner does not need to form the complete plan, but only to find the cheapest next join for each iteration.
- The output plan is given as input to the **Job Construction** phase (line 14) which actually converts it to a job (i.e. creation of query operators along with their connections). This job is executed and the materialized results will be rewired as input whenever they are needed by subsequent join stages.
- Finally, if the remaining number of datasets is more than three, we return to the **Planner** phase with the new query as formatted in the **Query Reconstruction** phase (line 13); otherwise the result is returned.

## 5.1 Selective Predicates

Filtering can occur in the WHERE clause of a query in several forms; we focus on selection predicates. If a dataset has only one local selection predicate with a fixed value,

**Algorithm 1** Dynamic Optimization

```
1: J ← joins participating in the original query
2: D ← collection of base datasets (d) in the query
3: Statistics ← quantile and hyperloglog sketches for each field of D that is a join key
4: Q(σ, D, J) ← original query as submitted by user ▷ σ is the projection list
5:
6: if |J| > 1 then
7:     for d in D do
8:         P ← set of selective predicates local to d
9:         if |P| > 1 then
10:            D − {d} ⋃ PushDownPredicates(d, P)
11:
12:    while |J| > 2 do
13:        j ← Planner(J, Statistics)
14:        Q(σ, D, J) ← QueryReconstruction(j, Q(σ, D, J))
15:        intermediateResults,Statistics ← ConstructAndExecute(j) ▷ collect statistical sketches on intermediate data and integrate them on the statistics collection framework
16:        J ← joins in Q(D)
17:
18:    j ← Planner(J, Statistics)
19:    return ConstructAndExecute(j)
20:
21: function PushDownPredicates(d, P)
22:    Q(σ, {d}, ∅) ← query consists only of d with its local predicates ▷ σ is filled by fields participating in joins
23:    d′, Statistics ← Execute(Q(σ, {d}, ∅)) ▷ update original Statistics with the sketches collected for the new d
24:    return d′
25:
26: function Planner(J, Statistics)
27:    minJoin ← ∅, finalJoin ← ∅
28:    for j in J do
29:        minJoin← min(minJoin,JoinCardinality(j, Statistics))
30:    if |J| = 2 then
31:        finalJoin← BestAlgorithm(minJoin)⋈BestAlgorithm((J-{minJoin}))
32:    else
33:        finalJoin ← BestAlgorithm(minJoin)
34:    return finalJoin
35:
36: function QueryReconstruction(j(d1, d2), Q(σ, D, J))
37:    d′ ← CreateDataset(j(d1, d2))
38:    D ← (D ⋃ {d′}) − {d1, d2}
39:    J ← J − {j(d1, d2)}
40:    return Q(σ, D, J)
```

we exploit the equi-height histogram's benefits. Depending on the number of buckets predefined for the histogram, its range cardinality estimation can reach high accuracy.

However, for multiple selection predicates or complex predicate(s), the prediction can be very misleading. In the case of multiple (fixed value) predicates, traditional optimizers assume predicate independence and thus the total selectivity is computed by multiplying the individual ones. This approach can easily lead to inaccurate estimates [23]. In the absence of values for parameters, and given non-uniformly distributed data (which is the norm in real life), an optimizer cannot make any sort of intelligent prediction of selectivity, thus default values are used as described in [32] (e.g. 1/10 for equalities and 1/3 for inequalities). The same approach is taken for predicates with UDFs [22]. Most works dealing with complex predicates [16, 22] focus on placing such predicates in the right order and position within the plan, given that the selectivity of the predicate is provided. In our work, we exploit the INGRES [38] approach and we push down the execution of predicates (lines 21-24 of Algorithm 1) to acquire accurate cardinalities of the influenced datasets.

As a complex predicate example consider the following query $Q_1$, which has four datasets, two of which are filtered with UDFs and then joined with the remaining two.

```
select A.a
from A, B, C, D
where udf(A) and A.b = B.b
and udf(C) and B.c = C.c
and B.d = D.d;
```

Per line 21 of Algorithm 1, we isolate datasets having local filters and create queries for each one *a la* INGRES. In $Q_1$, datasets $A$ and $C$ will be wrapped around the following single variable queries ($Q_2$ and $Q_3$ accordingly):

```
select A.a, A.b
from A
where udf(A);
```

```
select C.c
from C
where udf(C);
```

Both queries' SELECT clauses are defined by attributes in the remaining query (projections, joins, or other clauses of the main query). Once the queries' construction is done, we execute them and save the intermediate results for use in the remaining query. We also update the statistics (hyperloglog and quantile sketches) of the base unfiltered datasets to depict the new cardinalities. Once this process is done, we need to update $Q_1$ with the filtered datasets (line 10 in Algorithm 1) by removing the UDFs and changing the FROM clause. The final query that will be the input to the looping part of our algorithm (lines 12-16) is illustrated below as $Q_1'$.

```
select A'.a
from A', B, C', D
where A'.b = B.b and B.c = C'.c
and B.d = D.d;
```

## 5.2 Planner

Next is the Planner stage (lines 26-34), where the input is the non-optimized query (in our case $Q_1'$), along with the most updated statistics. The goal of this stage is to output the best plan (since we focus on joins, this is the plan containing the best join order and join algorithm).

The first step in the Planner phase is to identify the join with the least result cardinality, along with its algorithm (lines 28-29). After that, we need to construct the join which will be output. If there are more than two joins in the input, then the cheapest join is the output and we are done (lines 32-33). However, if there are only two joins, the Planner will pick the most suitable algorithm for each. Then, it will combine the two joins by ordering them according to their result cardinality estimation (lines 30-31 of Algorithm 1).

In $Q_1'$ there are three joins, which means that the first case is applied and it suffices to find the cheapest join according to statistics. Assuming that according to formula 1, A' and B lead to the smallest result cardinality, and A' (after the UDF application) is small enough to be broadcast, the plan output is a broadcast algorithm between A' and B ($J_{A'B}$).

## 5.3 Job Construction

Next, we construct a job for the plan (in our example, $J_{A'B}$) output by the previous stage (lines 15 and 19 of Algorithm 1). The details of how we construct a job in AsterixDB

are described in section 6.3. The way a job is executed depends on the number of joins in the plan. If there is only one join, we are still inside the looping part of the algorithm (line 15). There we need to materialize the intermediate results of the job and at the same time gather statistics for them. In our example, plan $J_{A'B}$ has only one join - so the aforementioned procedure will be followed and the joined results of A' and B will be saved for future processing along with their statistics. On the other hand, if the plan consists of two joins, it means that the dynamic optimization algorithm has been completed and the results of the job executed are returned back to the user (line 19 of Algorithm 1).

**Online Statistics:** For the statistics acquired by intermediate results, we use the same type of statistics as described in section 4. We only gather statistics on attributes that participate on subsequent join stages (and thus avoid collecting unnecessary information). The online statistics framework is enabled in all the iterations except for the last one (i.e. the number of remaining datasets is three) since we know that we are not going to further re-optimize.

## 5.4 Query Reconstruction

The last step of the iterative approach is reconstructing the remaining query (line 14 of Algorithm 1). Given that there will be more re-optimization points (more than two joins remaining), we need to reformulate the remaining query since the portion that participates in the job to be executed needs to be removed. When doing so: (a) the datasets participating in the output plan need to be removed (as they will not participate in the query anymore) and replaced by the intermediate joined result (lines 36-37); (b) the join output by the Planner needs to be removed (line 38); (c) any other clause of the original query influenced by the results of the job just constructed needs to be reconstructed.

Following our example, the Planner has picked as optimal the join between A' and B datasets. Consequently this join is executed first; then, the joined result is stored for further processing and is represented by a new dataset that we call $I_{AB}$. In terms of the initial query, this will trigger changes in all its clauses. Particularly, in the select clause the projected column derives from one of the datasets participated in the subjob (A). Hence, after its execution, the projected column will now derive from the newly created dataset $I_{AB}$. In the FROM clause both A and B should be removed and replaced by $I_{AB}$. Finally, in the WHERE clause, the join executed has to be removed and if its result participates in any of the subsequent joins, a suitable adjustment has to be made. To this end, in our example B is joined with C in its c attribute. However, the c column is now part of $I_{AB}$. As a result, $I_{AB}$ will now be joined with C. After these changes the reformatted query will look like this ($Q_4$):

```
select I_AB.a
from I_AB, C', D
where I_AB.c = C'.c and I_AB.d = D.d;
```

$Q_4$ has only two joins, so the looping part of our algorithm has been completed. Once the Planner picks the optimal join order and algorithm, the final job will be constructed and executed and its results will be returned to the user.

**Discussion:** By integrating multiple re-optimization points during mid-query execution and allowing complex predicate pre-processing, our approach can lead to much more accurate statistics and efficient query plans. However, stopping the query before each re-optimization point and gathering online statistics to refine the remaining plan introduces some overhead. As seen in Section 7 this overhead is not significant and the benefits brought by the dynamic approach (i.e., avoiding a bad plan) exceed it by far. Note that here we focus on simple UDF predicates applied on the base datasets. For more expensive UDF predicates, plans that pull up their evaluation need to be considered [22]. Another interesting point unlocked by dynamic optimization is the forming of bushy join plans. They are considered to be expensive since both join inputs need to be constructed before the join begins in a parallel environment, but they can be efficient since they open opportunities for smaller intermediate join results.

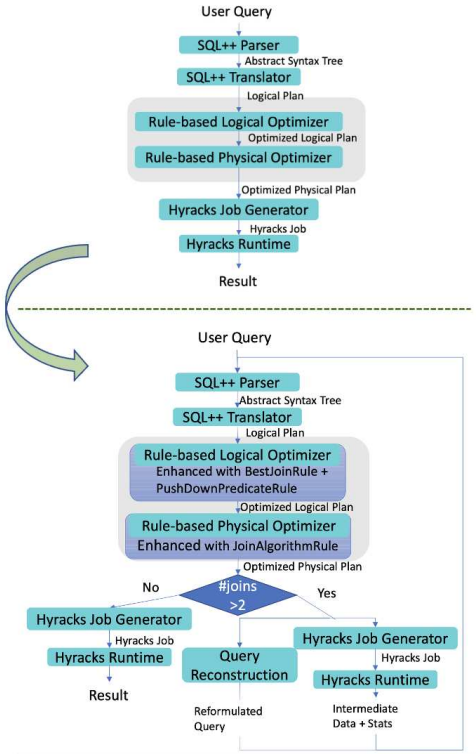



Figure 1: AsterixDB workflow without (top) and with (bottom) the integration of Dynamic Optimization

# 6. INTEGRATION INTO ASTERIXDB

The top portion of Figure 1 shows the current query workflow in AsterixDB, while the bottom shows our changes. In the beginning the workflow behaves similarly, except for a few additional rules in the rule-based (JoinReOrderRule, PushDownPredicateRule) and physical (JoinAlgorithmRule) optimizer (**Planner**). Afterwards, depending on the number of joins participating in the query currently being processed, we either construct and execute the Hyracks job and output the result to the user as usual (only two joins) or we perform the following steps (more than two joins):

- We introduce a **Query Reconstruction** phase where we reformulate the query being processed and redirect it as new input to the SQL++ parser and the whole query process starts from the beginning once again.

- We construct a Hyracks job (**Job Construction**) by using operators introduced to allow materialization of the results of the query currently being processed along with connection of previously (if any) executed jobs.

## 6.1 Planner

If a dataset has more than one filter, the PushDownPredicateRule is triggered. This rule will push the filters down to their datasource and will remove the rest of the operators from the plan, leading to a modified plan of a simple select-project query (like $Q_2$ and $Q_3$ in section 5.1). On the other hand, if there is only one filter, we estimate the filtered dataset cardinality based on histograms built on the base dataset. Afterwards, the Planner stage will decide the optimal join order and algorithm. In order for the Planner to pick the join with the least cardinality, we enhanced the rule-based logical Optimizer (part of the Algebricks framework) with the JoinReOrderRule (see Figure 1). To further improve the efficiency of the execution plan, we integrated a rule in the rule-based physical Optimizer (Figure 1) that picks the most suitable join algorithm.

***Join Ordering:*** The main goal of this rule is to identify the join with the least cardinality. We identify all of the individual joins along with the datasources (post-predicate execution) of their predicates. Here we focus only on joins formed in the WHERE clause of a query. In the future, we plan to infer more possible joins according to correlations between join predicates. Next, we apply formula 1 based on statistics (see Section 4) collected for the datasets and predicates involved in the join. Traditional optimizers based on static cost-based optimization form their complete plan from the beginning, so they search among all possible combinations of joins which can be very expensive depending on the number of base datasets. However, for the incremental optimization, it suffices to search for the cheapest join because the rest will be taken into consideration in the next iterations of our algorithm. In Figure 2, for $Q_1$ the join between post-predicate $A$ (shown as $A'$) and $B$ will be estimated as the cheapest and will be output from the Planner stage.

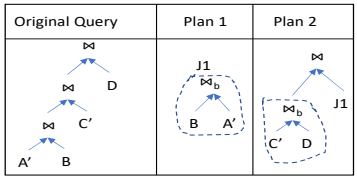



Figure 2: Planning Phase with Dynamic Optimization

The second feature of this rule is triggered when there are only two joins left in the query and hence the statistics obtained up to that point suffice to figure out the best join order between them. Specifically as depicted in Plan 2 of Figure 2, in this case a two-way join (between three datasets) is constructed whose inputs are (1) the join (between two of the three datasets) with the least result size (estimated as described above) and (2) the remaining dataset.

It is worth noticing that in the first iteration of the approach the datasets that are joined are always among the base datasets. However, in the rest of the iterations, one or both of the joined datasets may be among the results from previous iterations. An example of that is shown in Plan 2 of Figure 2, where the right dataset of the final join is the result of the first iteration (J1) of our algorithm.

***Join Algorithm:*** While hash join is the default algorithm, by having accurate information about the datasets participating in the corresponding join, the optimizer can make more efficient decisions. If one of the datasets is small enough, like $A'$ and $C'$ in our example (see Figure 2), then it can be faster to broadcast the whole dataset and avoid potential reshuffling of a large dataset over the network.

Knowing that the cardinality of one of the datasets is small enough to be broadcast also opens opportunities for performing the indexed nested loop join algorithm as well. However, two more conditions are necessary to trigger this join algorithm. The first one is the presence of a secondary index on the join predicate of the "probe" side. The second condition refers to the case of primary/foreign key join and dictates that the dataset that gets broadcast must be filtered - thereby during the index lookup of a large dataset there will be no need for all the pages to be accessed.

## 6.2 Query Reconstruction

This stage is entered in one of the following cases: (1) the Planner has output a simple projection plan (predicate push down) or (2) the Planner output is a select-project-join plan (cheapest join). In both cases, we follow the process described in section 5.4 to reformulate the clauses of the input query and output the new query that will be given as input to the optimizer for the remaining iterations.

## 6.3 Job Construction

There are three different settings when creating a job: (1) When there are still re-optimizations to be scheduled (more than 2 joins), the output of the job has to be materialized for future use. (2) If one or both inputs of a job is a previously materialized job output, we need to form a connection between the jobs. (3) When the iterations are completed, the result of the last job will be returned to the user.

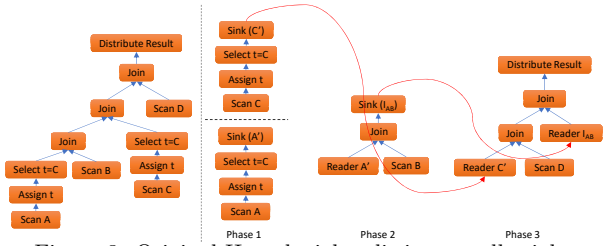



Figure 3: Original Hyracks job split into smaller jobs

Figure 3 shows the job-splitting process. The left side depicts the usual job for the three-way join query ($Q_1$); the final result is returned to the user via the DistributeResult operator. Instead, on the right side of the Figure (Phase 1), two subjobs are created that push down the UDF predicates to datasources A and C. Their results are the post-predicate versions of A and C (Sink(A') and Sink(C') accordingly). The Sink operator is responsible for materializing intermediate data while also gathering statistics on them.

In Phase 2, the subjob formed wraps the join between datasets A' and B, as this is the plan output by the Planner. Note that the new operator introduced in this phase (Reader A') indicates that a datasource is not a base dataset. Instead, it is intermediate data created by a previous subjob.

In our example, Reader A' represents the materialized data created in the previous phase by Sink(A'). Since the original query has not finished yet (remaining joins), the Sink operator will be triggered once again and it will store in a temporary file the joined results ($I_{AB}$), while at the same time it will collect the corresponding statistics.

Finally, the goal of Phase 3 is to wrap the output of the Planner which is a two-way join. The existence of two joins indicates that we are at the final iteration of the dynamic approach - thereby this job is the final one and its result should be returned to the user. Consequently, the DistributeResult operator re-appears in the job, as depicted in Figure 3.

**Discussion:** To integrate our approach into AsterixDB, we had to create an iterative workflow that would let us trigger multiple re-optimization points. We concentrate here on multi-join queries that may contain multiple and/or complex selection predicates. While other operators may exist in the query, for now they are evaluated after the joins and selections have been completed and traditional optimization has been applied. More costly UDF predicates and more complex queries (e.g., nested queries) are future work.

# 7. EXPERIMENTAL EVALUATION

We ran experiments on a cluster of 10 AWS nodes with an Intel(R) Xeon(R) E5-2686 v4 @ 2.30GHz CPU (4cores), 16GB of RAM and 2TB SSD running 64-bit Red-Hat 8.2.0. Every experiment was run five times and we calculated the average of the results. We explored performance for multiple TPC-DS [2] and TPC-H [3] join queries. Here we present four representative queries, TPC-DS Queries 17 and 50 and TPC-H Queries 8 and 9, chosen for their complexity (number of joins) and join condition variety (primary/foreign key vs. fact-to-fact). To explore multiple fixed value predicates, we added two (correlated [39]) predicates on the *orders* table in Query 8. To see the effect of UDFs, we added various UDFs on top of the *part* and *orders* tables in Query 9. In Query 50, we added two selections with parameterized values on one of the dimension tables. The queries' SQL versions and the plans produced for all experiments are available in [31].

We generated 3 TPC-DS and 3 TPC-H datasets with scale factors 10, 100, 1000. For scale factor 1000 the cumulative size for the datasets is 1TB. Scale factor 10 was used to see the overhead of our approach with very small datasets. We gathered the initial statistics used to form the initial plans while loading the data into AsterixDB. This was done once and was not a part of the query execution process or times. Loading times varied from 10 minutes to 8 hours depending on the dataset sizes. As shown in [5, 6], the statistics collection overhead is minimal relative to the loading time.

## 7.1 Overhead Considerations

To see the impact of re-optimization points and online statistics gathering on AsterixDB's execution times, we did three runs of each query. The first acquired all statistics needed to form the optimal execution plan using our runtime dynamic optimization technique. The second re-executed the query using the updated statistics so that the optimal plan is found from the outset. For the third run we enabled re-optimization points but disabled online statistics collection. This helped pinpoint the overhead due to materialized data handling. Then, to evaluate the cost of online statistics gathering, we deducted the third execution time (re-optimization) from the first (the full dynamic optimization technique). We examined query execution times at scale factors 100 and 1000. We observed a 7-13% overhead for scale factor 100 and up to 20% for scale factor 1000 (see [31] for details). This seems tolerable given our approach's benefits.

For scale factor 100, the total re-optimization time was around 10% of the execution time; for Q50, with only four joins (and two reoptimization points), the overhead was 2%. For scale factor of 1000, the re-optimization overhead increases up to 15% for most queries, as the intermediate data produced is larger and the I/O cost introduced by reading and writing intermediate data is thus higher.

Online statistics collection added a small overhead of 1% to 3% (scale factor 100) to the total execution time, as it is masked from the time needed to store and scan the intermediate data. Moreover, the extra time for statistics depends on the number of attributes that we need to keep statistics for. For example, for Q50, the statistics collection overhead was only 1% because it has the fewest join conditions. For scale factor 1000, the overhead of gathering statistics is increased, as the data for which we collect statistics is larger in size, but it remains insignificant (up to 5%).

Finally, we assessed the overhead of applying the incremental optimization approach to estimate the influences of multiple/complex predicates. We first deactivated the multiple re-optimization points and executed the plan formed as if the right statistical data is available from the beginning. Then, the experiment was repeated by enabling dynamic optimization only for materializing the intermediate results coming from pushing down and executing multiple predicates. The remaining query was executed based on the refined statistics coming from the latter step. Our results showed that even when there are multiple filters present (like in Q17), the overhead does not exceed 3% of the total execution time, even for scale factor 1000.

## 7.2 Comparison of Execution Times

We compared our dynamic approach with: (i) AsterixDB with the worst-order, (ii) AsterixDB with the best-order (as submitted by the user), (iii) AsterixDB with static cost-based optimization, (iv) the pilot-run [26] approach, and (v) an INGRES-like approach [38]. For the worst-order plan, we enforce a right-deep tree plan that schedules the joins in decreasing order of join result sizes (the size of the join results was computed during our optimization). The best-order plan assumes that the user knows the optimal order generated by our approach and uses that order in the FROM clause when writing the query. We also added some broadcast hints so the default optimizer can choose the broadcast algorithm. These two settings represent the least and the most gain, accordingly, that we can achieve with our approach against the default approach(es) of AsterixDB.

To compare with a traditional cost-based optimization approach, we collected statistics on the base datasets during the ingestion phase and formed the complete execution plan at the beginning based on the collected statistics. When UDFs or parameters are present we use the default selectivity factors as described in [32]. For the pilot-run method, we gathered the initial statistics by running select-project queries (pilot-runs) on a sample of each of the base datasets participating in the submitted query. If there are predicates local to the datasets, they are included in the pilot-runs. In

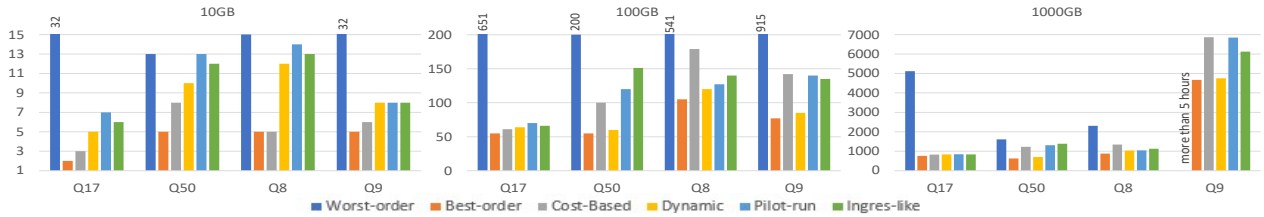
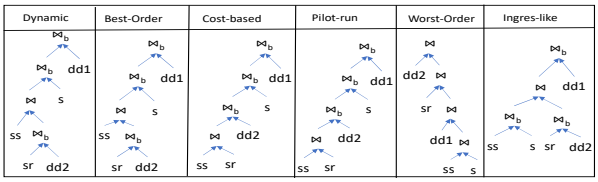
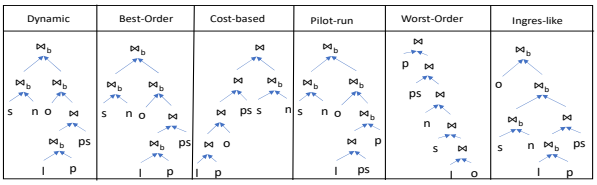



Figure 4: Comparison between Dynamic Optimization, traditional cost-based optimization, regular AsterixDB (join best-order vs worst-order), Pilot-run and Ingres-like




(a) Query 50




(b) Query 9

Figure 5: Plans Generated for: (a) Query 50 and (b) Query 9, of Figure 4 with scale factor 100.

the sampling technique used in [26] during pilot runs, after $k$ tuples have been output the job stops. To simulate that technique we enhanced our "pilot runs" with a LIMIT clause. Based on those statistics, an initial (complete) plan is formed and the execution of the original query begins until the next re-optimization point where the plan will be adjusted according to feedback acquired by online statistics.

For comparison with the INGRES-like approach we used our method to decompose the initial query into single variable queries, but with the choice of the next best subquery to execute based solely on dataset cardinalities. In the INGRES the intermediate data was inserted into a new relation; here we stored it in a temporary file. Our results are given in Figure 4. We discuss in detail TPC-DS Query 50 and TPC-H Query 9; their query plans appear in Figure 5.

*Query 50:* This query has two dimension (*date_dim*) tables, one that is filtered (with parameterized expressions), two large tables, and *Store* which helps to prune the final result. The optimal plan found by our dynamic approach first prunes one of the fact tables by joining it with the filtered dimension table and then joins it with the other large table. Our approach also pick the broadcast algorithm when appropriate (Figure 5a). With broadcast hints, best-order chooses the same plan, leading to slightly better performance than ours (1.05 and 1.1x for scale factors 100 and 1000).

Cost-based optimization yields a different plan due to inaccurate cardinality estimates for the filtered dimension table and the fact table join. It picks most of the broadcast joins, but is 1.5x worse than our approach for scale factors 100 and 1000. The INGRES-like approach forms a bushy tree due to its naive cost-model (using only dataset cardinalities), yielding worse performance. The AsterixDB worst-order case uses hash joins and schedules the fact table join too early, performing the worst. Lastly, pilot-run makes the wrong join ordering decision for the large tables due to inaccurate statistics and is 1.8x slower than our approach.

*Query 9:* Here *lineitem* is joined on foreign/primary key with four smaller tables and foreign key with *part_sup*. Our approach finds the optimal join order, a bushy tree. It chooses broadcast join for the *part* table for scale factors 10 and 100 and for the the *nation* and *supplier* join result (Figure 5b). Cost-based optimization finds a similar bushy tree, but due to poor cardinality estimation, fails to broadcast the part table; intermediate data from joining nation and supplier is only broadcast for scale factor 10. As a result, our approach performs a bit better than the cost-based one. AsterixDB best-order uses the optimal execution plan.

AsterixDB worst-order does the largest result-producing joins early using hash joins, yielding an execution time of over 5 hours. Most other techniques were 7x better than worst-order. For pilot-run, a suboptimal plan was chosen due to inaccurate unique cardinalities estimated by initial sampling. The INGRES-like approach forms a less efficient bushy tree because it focuses only on dataset cardinalities.

Note that there are cases where the traditional cost-based optimizer can pick a better plan (as in Q17 which has multiple uncorrelated predicates [31]). We also examined the behavior of our approach when the Indexed Nested loop Join (INLJ) is added as another possible join algorithm choice. Again its performance was superior to the competitors [31].

**Discussion:** Our experiments have shown the benefits of our approach, especially for key/foreign-key joins. The size of their results can vary greatly from predictions based on base statistics, so intermediate statistics enable better costing and join ordering. The 100GB dataset saw the most benefit. With a large enough base dataset, bad plans from a static optimizer are noticeable and broadcast joins have a better chance of being chosen by our approach due to more accurate selectivities. The 1000GB dataset benefited less, as broadcast joins are less applicable and intermediate results are larger, leading to more overhead. Nevertheless, we still outperformed the other approaches. The 10GB case improved the least since the base datasets' sizes are tiny and the overhead imposed by intermediate materialization is noticeable. An interesting note is that many of the optimal plans were bushy, so even if both inputs must be materialized before a join, having smaller intermediate join results brings more overall benefit. As for the overheads of our techniques, although in the worst case (scale factor 1000) their cost can be high, in most cases the resulting plans are still faster than those produced by traditional optimization.

## 8. CONCLUSIONS

We re-examined dynamic query optimization in a big data

setting. Our approach decomposes a query into subqueries, gathering statistics on intermediate data to plan the remaining query. We focused on complex joins, but also treated predicates with multiple selections, parameters, and UDFs. While our approach interferes with pipelining and uses materialization, it seems to work surprisingly well. Experiments saw it picking much better plans than static cost-based optimization and various recent approaches. With big data, it pays to get good statistics via re-optimization points since small errors in estimating sizes of large data can have major consequences. Our approach performed best when complex predicates were applied to the base datasets or the join conditions were between fact tables (leading to skew in selectivity and join result estimates). Future work should include additional experiments with a wider range of queries.

## 9. ACKNOWLEDGEMENTS

This research was supported in part by NSF grants IIS-1838222 and IIS-1838248, which also provided AWS credits.

## 10. REFERENCES

[1] *Apache AsterixDB*, 2020.

[2] *TPCDS*, 2020.

[3] *TPCH*, 2020.

[4] R. Abdel Kader, P. Boncz, S. Manegold, and M. Van Keulen. ROX: run-time optimization of XQueries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 615–626, 2009.

[5] I. Absalyamov. *Query Processing and Cardinality Estimation in Modern Database Systems*. PhD thesis, University of California, Riverside, 2018.

[6] I. Absalyamov, M. J. Carey, and V. J. Tsotras. Lightweight cardinality estimation in LSM-based systems. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pages 841–855, 2018.

[7] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 281–294, 2012.

[8] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, et al. AsterixDB: A scalable, open source BDMS. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[9] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[10] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[11] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 107–118. ACM, 2005.

[12] S. Babu and H. Herodotou. Massively parallel databases and mapreduce systems. *Foundations and Trends® in Databases*, 2013.

[13] V. Borkar, Y. Bu, E. P. Carman Jr, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In *Proceedings of 6th ACM Symposium on Cloud Computing*, pages 422–433, 2015.

[14] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *27th International Conference on Data Engineering*, pages 1151–1162, 2011.

[15] N. Bruno, S. Jain, and J. Zhou. Continuous cloud-scale query optimization and processing. *Proceedings of the VLDB Endowment*, 6(11):961–972, 2013.

[16] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems (TODS)*, 24(2):177–228, 1999.

[17] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews. The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment*, 9(13):1401–1412, 2016.

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

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

[20] G. Graefe and D. J. DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 160–172, 1987.

[21] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218. IEEE, 1993.

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

[23] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 647–658, 2004.

[24] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 268–277, 1991.

[25] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 106–117, 1998.

[26] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. In *Proceedings of the 2014 ACM SIGMOD International*

*Conference on Management of Data*, pages 943–954. ACM, 2014.

[27] Q. Ke, M. Isard, and Y. Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 15–28. ACM, 2013.

[28] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173*, 2012.

[29] Y. Li, M. Li, L. Ding, and M. Interlandi. Rios: Runtime integrated optimizer for spark. In *SoCC*, pages 275–287, 2018.

[30] F. Meunier, O. Gandouet, É. Fusy, and P. Flajolet. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, 2007.

[31] C. Pavlopoulou, M. J. Carey, and V. J. Tsotras. Revisiting runtime dynamic optimization for join queries in big data management systems. *arXiv preprint arXiv:2010.00728*, 2020.

[32] 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 *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[33] S. Sikdar and C. Jermaine. Monsoon: Multi-step optimization and execution of queries with partially obscured predicates. In *Proceedings of the 2020 ACM SIGMOD Conference*, pages 225–240, 2020.

[34] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, et al. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2014.

[35] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO-DB2's learning optimizer. In *VLDB Conference*, volume 1, pages 19–28, 2001.

[36] I. Trummer and C. Koch. Parallelizing query optimization on shared-nothing architectures. *Proceedings of the VLDB Endowment*, 9(9):660–671, 2016.

[37] L. Wang, G. Luo, K. Yi, and G. Cormode. Quantiles over data streams: an experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 737–748, 2013.

[38] E. Wong and K. Youssefi. Decomposition—a strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[39] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. Automatic discovery of attributes in relational databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 109–120, 2011.

[40] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *The VLDB Journal*, 21(5):611–636, 2012.