# Unified Query Optimization in the Fabric Data Warehouse

Nicolas Bruno
nicolas.bruno@microsoft.com
Microsoft
Redmond, WA, USA

César Galindo-Legaria
cesarg@microsoft.com
Microsoft
Redmond, WA, USA

Milind Joshi
milind.joshi@microsoft.com
Microsoft
Redmond, WA, USA

Esteban Calvo Vargas
esteban.calvo@microsoft.com
Microsoft
Redmond, WA, USA

Kabita Mahapatra
kabitam@microsoft.com
Microsoft
Redmond, WA, USA

Sharon Ravindran
shravind@microsoft.com
Microsoft
Redmond, WA, USA

Guoheng Chen
guche@microsoft.com
Microsoft
Redmond, WA, USA

Ernesto Cervantes Juárez
ernesto.cervantes@microsoft.com
Microsoft
Redmond, WA, USA

Beysim Sezgin
beysims@microsoft.com
Microsoft
Redmond, WA, USA

## ABSTRACT

Over a decade ago, Microsoft introduced Parallel Data Warehouse (PDW), a massively parallel processing system to manage and query large amounts of data. Its optimizer was built by reusing SQL Server's infrastructure with minimal changes, which was an effective approach to bring cost-based query optimization quickly to PDW. Over time, learnings from production as well as architectural changes in the product (such as moving from an appliance form factor to the cloud, separation of compute and storage, and serverless components) required evolving the query optimizer in Fabric DW, the latest offering from Microsoft in the cloud data warehouse space. In this paper we describe the changes to the optimization process in Fabric DW, compare them to the earlier architecture used in PDW, and validate our new approach.

## CCS CONCEPTS

• **Information systems → Query optimization**.

## KEYWORDS

Query Optimization; Cascades Framework; Distributed database systems

**ACM Reference Format:**
Nicolas Bruno, César Galindo-Legaria, Milind Joshi, Esteban Calvo Vargas, Kabita Mahapatra, Sharon Ravindran, Guoheng Chen, Ernesto Cervantes Juárez, and Beysim Sezgin. 2024. Unified Query Optimization in the Fabric Data Warehouse. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653369

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3653369

## 1 INTRODUCTION

Over a decade ago there was a major trend in the data warehouse industry towards the wide adoption of Massively Parallel Processing systems (MPP) for managing large amounts of data [3, 6, 12, 20, 23, 25, 26]. MPP systems use multiple independent nodes with their own software stack and memory, connected by a high-speed network. Microsoft SQL Server Parallel Data Warehouse (PDW) was a shared-nothing MPP appliance introduced in 2010. It came in a number of hardware configurations, with the necessary software preinstalled and ready to use. PDW had a control node that managed a number of backend nodes. The distributed engine in the control node provided the external interface to the appliance, and was responsible for parsing input queries, creating distributed execution plans, issuing plan steps to backend nodes, tracking the execution steps of the plan, and assembling the individual pieces of the final results into the single returned result set. Backend nodes provided the data storage and the query processing backbone of the appliance. Both control and backend nodes had an instance of SQL Server RDBMS running on them, and user data was stored in hash-partitioned or replicated tables across the backend nodes.

A few years later, with cloud providers taking the stage, PDW evolved into Synapse Data Warehouse (or Synapse DW). Synapse DW transformed the appliance-based engine into a fully managed PAAS offering. The main building blocks remained the same, with the control node hosting the distributed engine and a local front-end SQL Server instance, and backend nodes hosting SQL Server instances. Some architectural changes included the decoupling of compute and storage, which provided flexible resource scaling, and a multi-layered data caching model with prefetching of column-store data for large workloads.

Over time, the architecture of Synapse DW began evolving from a simple port of an on-premise system to a cloud-native scale-out service. This transition started with Synapse Serverless SQL Pool [14], based on the highly available Polaris framework [1]. Polaris follows a stateless architecture, requiring backend nodes to hold no state information (e.g., data, transactional logs and metadata) other than transient caches for performance. This allows the engine to partially restart execution of queries in the event of

compute node failures or online changes of the cluster topology. Additionally, global resource-aware scheduling and fine-grained scale-out execution based on directed acyclic graphs provide extra flexibility and performance during query execution. This evolution continues now with the introduction of Microsoft Fabric [16], an all-in-one solution that covers data movement, data science, real-time analytics, and business intelligence. Fabric offers various tools and services, including Fabric Data Warehouse (or Fabric DW), which is also based on Polaris and converges the world of data lakes and warehouses. Fabric DW stores data in the Parquet file format [9] and published as Delta Lake Logs [7], enabling ACID transactions and cross-engine interoperability that can be leveraged through other components such as Spark and Power BI [2].

As part of this architectural transformation and based on learnings from a decade of experience with various distributed form factors, the query processor of Fabric DW evolved as well. In this paper we explore this evolution from the point of view of query optimization. In Section 2 we review the compiler architecture of the PDW system. In Section 3 we summarize some learnings from deploying PDW and Synapse DW in production, as well as some limitations we encountered as we evolved the overall architecture of the system. In Section 4 we describe the new architecture of Fabric DW from the point of view of compilation, and the changes in query optimization required to evolve the compiler to this new architecture. We report some initial results in Section 5, review related work in Section 6, and conclude in Section 7.

## 2 PDW COMPILER ARCHITECTURE

We now summarize the architecture of the earlier PDW compiler by describing the various steps involved in evaluating input queries (see Figure 1 and refer to [20] for more details):

(1) The input query is submitted to the PDW distributed engine, where it is parsed, validated, and sent to the frontend SQL Server Engine (which is collocated with the distributed engine in the control node).
(2) The SQL Server Frontend Engine maintains a *shell database*, which defines all metadata and global statistics about tables (whose data is actually partitioned on backend nodes) as well as information about users and privileges. From the compilation point of view, the shell database is indistinguishable from one that contains actual data. The query is then parsed, algebrized, and transformed into a logical relational tree, which is simplified and optimized. Optimization uses transformation rules to explore the space of equivalent alternatives, costs these alternatives, and returns the one that is expected to be the most efficient. Note that this optimization does not take into account the distributed nature of the data, but instead explores all *centralized* plans, as if all the data was actually associated with that server. While the output of the traditional SQL Server optimizer is the best execution plan, the frontend SQL Server optimizer returns the whole search space along with statistical information for each alternative. This is done because simply parallelizing the best centralized plan can result in suboptimal plans [20]. The search space (or `MEMO` as discussed in Section 4.1) is serialized and sent back to the distributed engine.
(3) The distributed engine deserializes the `MEMO` and starts a second stage of optimization using a *distributed query optimizer*, or *DQO*. A bottom-up strategy similar to that of System-R [19] enumerates distributed execution strategies by introducing appropriate data movement operators in the `MEMO`. *DQO* considers distributions as interesting properties analogous to sort orders in System-R, and thus reduces the search space using the classical dynamic programming approach. It then costs alternatives and obtains the distributed execution plan that minimizes data movement. This distributed plan is transformed into an executable format. The result of this procedure is a linearized sequence of steps. Each step corresponds to an operator tree whose root and leaf nodes are connected to other steps by data movement operators.
(4) The distributed plan is sent to the scheduler/executor in the distributed engine. At that point, steps are executed in sequence. The execution subplan of each step is transformed into `SQL` and sent to backend nodes, along with instructions on how to move the resulting data across those nodes (e.g., shuffling the result on a subset of columns).
(5) Each backend node receives a *standard* `SQL` statement over the data slice it owns, plus a data move directive on the result. Parsing, algebrization and a new round of optimization are done on the received query, and the resulting execution plan is passed to the execution engine in the backend node.
(6) The plan is executed in the backend node as if it was obtained from a regular input query.
(7) The result of execution is moved across backend nodes by using temporary landing tables as the backing storage. Metadata information about each result is sent back to the executor in the distributed engine, so that subsequent steps are executed correctly against temporary tables. For the last step, the actual results are sent back to the distributed engine.
(8) The final results from all backend nodes are aggregated and sent back to the client.

In short, the frontend engine maintains a shell database with global statistical information on tables, and backend nodes maintain a slice of the global database with actual data. Query compilation involves three different optimization calls: the centralized optimization in the frontend engine that produces the full logical search space, the distributed optimization in the distributed engine that minimizes data movement, and a third round of fragment optimizations in the backend nodes for each distributed step.

*Example 2.1.* Consider two tables $T(Ta, Tb)$ and $S(Sa, Sb)$, both distributed using a round-robin scheme, and the following query:

```
SELECT DISTINCT T.tb
FROM T INNER JOIN S ON T.Ta = S.Sa
WHERE S.Sb > 3
```

The query eventually reaches the frontend SQL Server engine and is optimized (e.g., considering different join orders, optionally pushing a partial distinct operator below the join) and the search space is sent back to the PDW distributed engine, where various distribution strategies are compared and the best distributed plan is generated (see Figure 2(a)). In turn, Figure 2(b) shows the graph associated with the best distributed plan, where each node (or step)

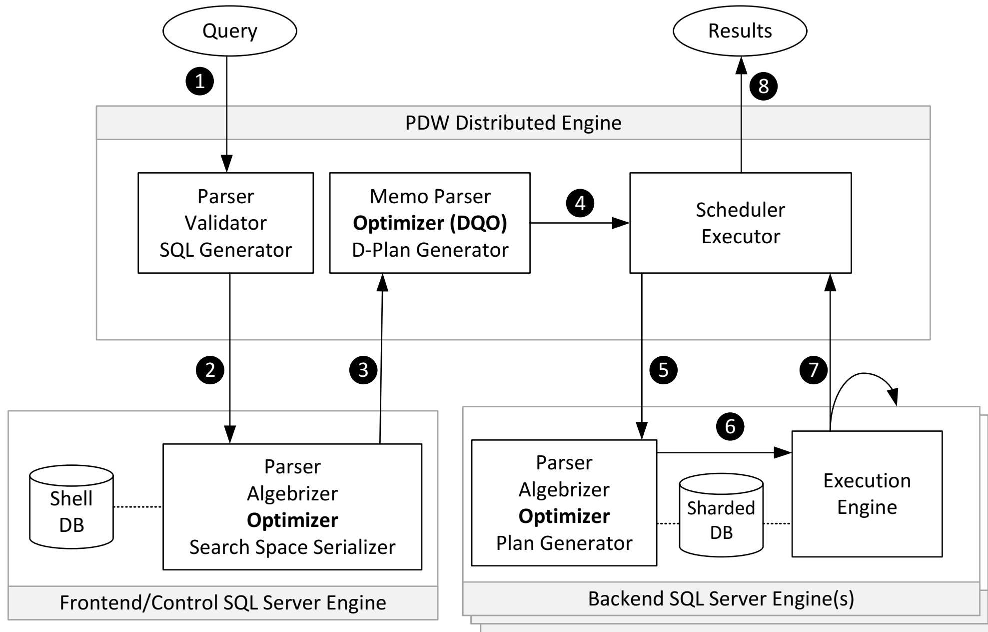



**Figure 1: Life of a query in the original Parallel Data Warehouse system.**

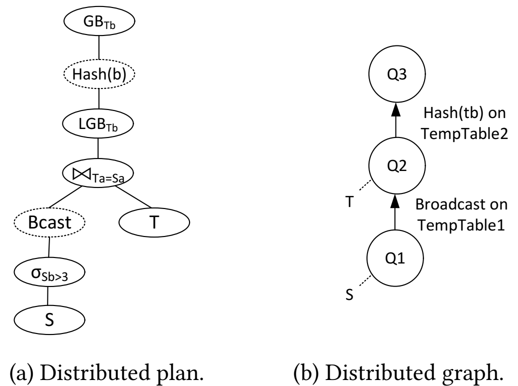
corresponds to a SQL template and edges represent data movement. The SQL templates in the figure are defined as follows:

```
Q1: SELECT sa FROM S WHERE sb > 3

Q2: SELECT T.tb
    FROM TempTable1 AS TT1 INNER JOIN T
    ON TT1.sa = T.ta
    GROUP BY T.tb

Q3: SELECT tb FROM TempTable2
    GROUP BY tb
```

The distributed plan is evaluated one step at a time, by sending the SQL fragments to the backend nodes, where they are optimized and executed. First $Q1$ is sent and evaluated in multiple backend nodes, reading fragments of table $S$ and filtering tuples that satisfy $sb > 3$. The result is estimated to be small enough that a replicated join strategy is chosen, so the intermediate results are broadcast to all nodes using $TempTable1$ as the backing store. Fragment $Q2$ is then sent to the backend nodes, where fragments of $T$ are joined with the replicated $TempTable1$ and locally aggregated. The results are then shuffled by the distinct column $tb$ and stored in $TempTable2$. Finally, fragment $Q3$ is evaluated in all nodes, the global distinct operator is applied to all fragments (since data is partitioned by column $T.tb$) and results are sent back to the client.




(a) Distributed plan. (b) Distributed graph.

**Figure 2: Distributed execution plan for a simple query.**

## 3 LEARNINGS FROM PRODUCTION

Developing an industrial-strength query optimizer from scratch is a major undertaking. Enumerating execution alternatives adequately requires an understanding of relational algebra and its properties, as well as deriving and identifying desirable execution plans. Effective plan selection requires careful modeling of data distributions and cost estimation and should go beyond simply parallelizing the best serial plan. The PDW optimizer went beyond simple predicate pushing and join reordering, and incorporated a number of advanced query optimization techniques, including contradiction detection, redundant join elimination, subquery unnesting, and outerjoin reordering [20].

For a commercial product, time to market is a critical dimension to consider. Rather than starting from scratch, the PDW query optimizer reused technology developed for SQL Server, which has been tuned and tested over a number of releases. This choice shortened the time to build a cost-based optimizer for PDW and was an effective approach to quickly bring cost-based query optimization to

the appliance, leveraging SQL Server infrastructure with minimal changes to existing components. Ultimately, it effectively incorporated state-of-the-art query optimization techniques in PDW.

Over time, however, we identified a number of shortcomings in the original approach. This was coupled with new architectural changes that made us revisit some initial design choices. We next mention some issues we encountered, mainly centered around the separation of query optimization into three interdependent sub-optimizers (see Figure 1), and the internal representation of information passed across the various components.

For large queries, it is infeasible to do an exhaustive search over the plan space. The optimizer in SQL Server relies on heuristics to restrict the exploration of the search space, and progressively expands this space via stages and timeouts. This infrastructure, however, is not integrated with the generation of distributed plans, because that logic is in a separate component. For that reason, the frontend optimizer must evaluate a larger portion of the search space, since the ultimate cost of plans is not known during the initial centralized optimization stage. The full search space of alternatives must be copied between the frontend and distributed engines. This can take a large amount of memory, since the required space in the worst case is exponential in the number of joins in the query.

During distributed optimization, the cost-based decisions for the distributed plan only consider the data movement cost. In some cases, however, it is important to also consider the execution per node. This shortcoming has come up a few times, such as during implementation of materialized views. It is possible that evaluating a query over base tables does not require any data movement due to data collocation, while the use of a materialized view introduces some data movement but it processes much less data. A unified cost model also enables the use of optimizer-driven physical schema recommendations, such as indexes, and materialized views.

Over the years, SQL Server developed a set of tools that go all the way from graphical plans and progress reporting to query stores and plan forcing [15, 17]. In PDW, physical execution plans are known only incrementally as distributed plans are optimized and executed in backend nodes. This difference in compilation prevents reusing SQL Server supportability mechanisms.

Additionally, execution plans in different distributions might be different, as they are optimized from SQL based on local statistics. While this mechanism provides additional flexibility via adaptive query execution and could improve performance in the backend nodes, it greatly increases the complexity of debugging performance issues or, in general, understanding execution plans. Conversely, backend optimization might lack some global information on intermediate results (e.g., unique constraints) that is known during frontend optimization but not carried over in the temp tables created as result of intermediate data movement. This lack of information further affects the optimality of backend plans.

Finally, the SQL decoding of plans from the distributed engine to the backend nodes was initially important to avoid deeper changes in the SQL Server codebase. However, in general it is problematic to rely on this mechanism for two reasons. First, it is rather difficult to translate internal execution plans back into SQL since the former has more expressive power via constructs that are not available in the SQL language (for instance , $Q2$ in Example 2.1 should be a partial aggregate, but there is no SQL construct to express this modality). Second, it is not possible to guarantee *plan idempotence* across decoding/optimizing steps. To illustrate these points, consider tables $R(ra)$ and $S(sb)$, and the deceptively simple query below:

```
SELECT (SELECT sb FROM S) FROM T
```

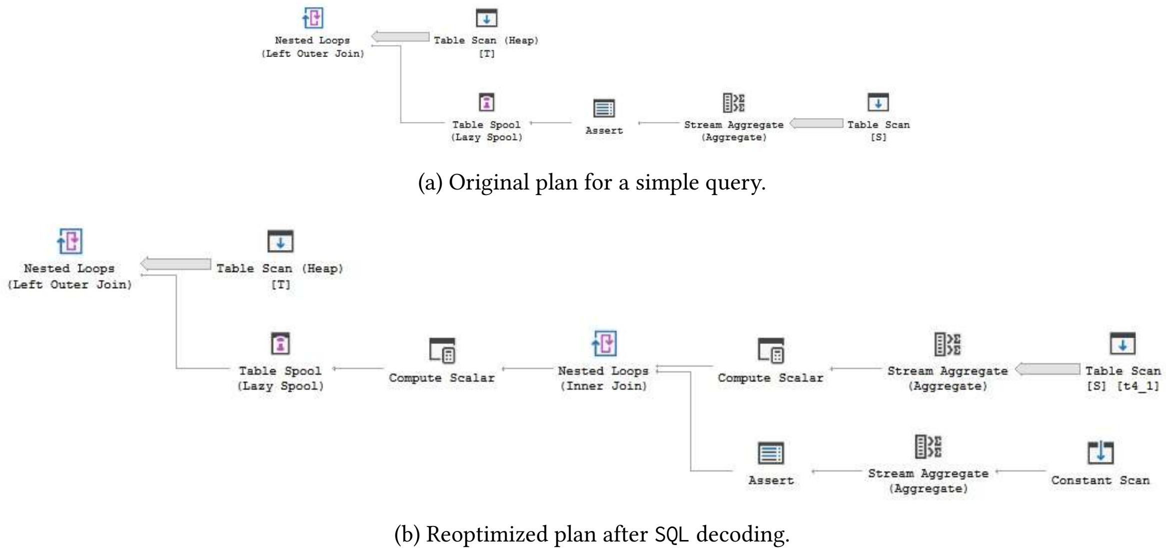
The query processes each row of $T$ and follows one of three cases, depending on the result of evaluating the subquery on table $S$. Specifically, if exactly one row is returned from the subquery, such value is used in the outer query. If no rows are returned from the subquery, then NULL is used. Finally, if more than one row is returned from the subquery, a runtime error is generated [13]. For that reason, execution must somehow validate that $S$ has at most one value and fail otherwise. Figure 3(a) shows a possible plan that SQL Server generates to evaluate this query assuming, for simplicity, that data is not distributed. The plan does a left outer join between $T$ and the subplan that (i) reads $S$, (ii) performs a scalar aggregation with aggregates `e1 = count(*)` and `e2 = any(S.sb)`, and (iii) asserts (via a special operator) that $e1 <= 1$ before spooling the result of $e2$ so that this computation is done only once.

Suppose now that we need to transform this plan back to SQL, to send it to backend nodes. Conceptually, the simplest way is to return the original query, but that requires arbitrarily complex global analysis of plans. Instead, PDW uses the *QRel* programming framework [8], which encapsulates the knowledge of mapping relational trees to query statements. While the produced queries in most cases are similar to the original query fragments, some cases are difficult to handle. In our example, we need to somehow decode the *Assert* operator, which does not have a counterpart in SQL (a smaller secondary problem is that the ANY aggregate function is not currently supported in PDW's SQL dialect). For the tree in Figure 3(a), *QRel* generates the following SQL query:

```
SELECT t2.col AS sb
FROM T LEFT OUTER JOIN
  (SELECT CASE WHEN (t3.col > 1)
           THEN (SELECT t5.asrt
                   FROM (VALUES (0), (0)) AS t5(asrt))
           ELSE t3.col1 END AS col
   FROM (SELECT count(0) AS col, max(t4.sb) AS col1
          FROM S AS t4) AS t3) AS t2
ON 0 = 0
```

Specifically, it decodes the `Assert` operator by creating a subquery that would return the same result (including runtime failures) as the original query. For that purpose, it first gets the number of elements in $S$ (`count(0) AS col`) and the maximum value of $S.sb$ (`max(sb) as col1`), which, for the case of a single value, has the same semantics as the ANY aggregate. Then, it projects from this single-row result a CASE statement that checks whether the `col <= 1`, in which case returns $col1$, or else returns a subquery with two rows (using the VALUES clause in the query). This two-row subquery is part of a scalar context (the THEN clause of a CASE statement) so it would require checking that it has at most one row, failing in the same cases as the original query. In addition to being complex to understand, the resulting SQL, when optimized, results in the plan of Figure 3(b), which is clearly different from the original one.

We note that in most common scenarios we do have idempotence of plans modulo SQL decoding, but as shown in Figure 3 this is not always the case. Moreover, other plan constructs (especially those




(a) Original plan for a simple query.

(b) Reoptimized plan after SQL decoding.

**Figure 3: Plans before and after a decoding/optimizing roundtrip.**

using advanced Update or Merge clauses) are significantly harder to decode than the example above. Over time, the SQL decoder grew in complexity to capture all possible execution plans, carefully selecting SQL constructs in cases that there were multiple ways of expressing the same query, and sometimes using private SQL extensions on backend nodes (e.g., hints) to express subtle operator semantics. A large testing effort complemented this approach, using random data/query generators and carefully crafted test cases.

# 4 QUERY OPTIMIZATION IN FABRIC DW

We next describe the query processing architecture in Fabric DW by summarizing how input queries are evaluated. This architecture is built on top of that of Synapse Serverless SQL pools, powered by the Polaris framework. Next, we dive deep into details on how this is supported from the point of view of query optimization. Figure 4 shows the steps for evaluating input queries (it might be useful to contrast this workflow with that of Figure 1).

(1) The input query is sent directly to the Frontend SQL Server Engine (SQL FE). The initial stages of compilation are similar to those in Figure 1, using the *Shell Database* as the source of metadata, table statistics, and authentication. A unified query optimizer (UQO) combines the work done by multiple optimizations in the previous architecture, and generates a distributed plan (see Section 4.2). From the point of view of SQL Server, the resulting plan consists of a single ExternalComputation operator, which encodes the actual distributed plan. The distributed plan is similar to that in Figure 1 with some differences. Unlike PDW, which linearizes the plan into a fixed sequence of steps, Polaris treats distributed plans as directed acyclic graphs to leverage independent parallelism of tasks. Also, tasks in the execution graph are not decoded back into SQL, but kept as physical plans.
(2) The distributed plan is sent to a thin execution wrapper in the SQL FE, which serves as an intermediate between the client and the distributed computation platform (DCP).
(3) The distributed plan arrives to the Polaris DCP [1, 2], which performs distributed workload management, scheduling, and execution (DQP), sending work to Backend SQL Server instances (SQL BE).
(4) Plans received by SQL BE nodes are directly executed without the additional round of optimization of the previous architecture[1].
(5) Results of backend executions are moved according to the data move directives among SQL BE nodes, and execution metadata is sent back to the DCP. For the last step, results are sent to the SQL FE.
(6) SQL FE aggregates results and sends them back to the client.

In summary, the SQL FE is the entry point to the system, and the three complementary optimizations were consolidated into a single unified component. We next describe the various changes required to achieve this goal.

## 4.1 The Cascades Framework

The Cascades Optimization Framework [10] was developed in the mid-nineties and it is used as the foundation for both industrial (e.g., the SQL Server engines discussed in this work [11] and Tandem's NonStop SQL [5]) and academic (e.g., Columbia [4]) query optimizers. We next provide a high level overview of the Cascades framework followed by a focused description of the components that are relevant to this work.

The Cascades Optimization framework results in top-down transformational optimizers that produce efficient execution plans for

[1]Some decisions are still late-bound, such as memory grants for individual operators. Although not currently used, the architecture allows for mixing logical and physical operators in the plans sent to backend nodes, to enable progressive optimization scenarios that take into account local statistics.

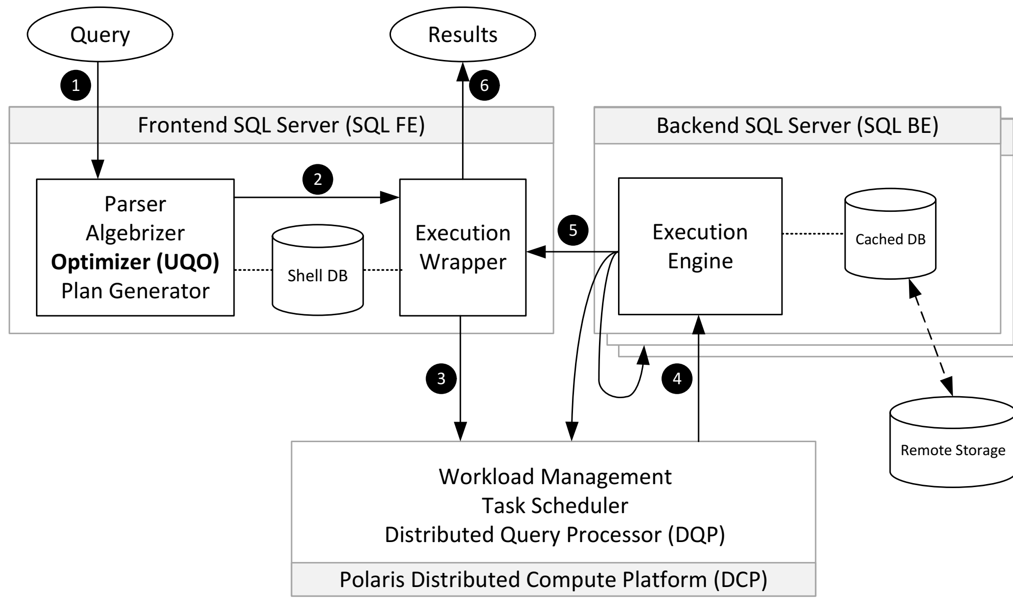



**Figure 4: Life of a query in the Fabric Data Warehouse.**

input declarative queries. These optimizers work by manipulating *operators*, which are the building blocks of *operator trees* and used to describe both the input declarative queries and the output execution plans. Consider the simple SQL query:

```
SELECT *
FROM R, S, T
WHERE R.x = S.x AND S.y = T.y
```

Figure 5(a) shows a tree of logical operators that specify, in an almost one-to-one correspondence, the relational algebra representation of the query above. In turn, Figure 5(c) shows a tree of physical operators that corresponds to an efficient execution plan for the above query. The goal of a query optimizer is to transform the original logical tree into an efficient physical tree. For that purpose, Cascades-based optimizers rely on two components: the MEMO data structure (which keeps track of the explored search space) and optimization tasks (which guide the search strategy).

*4.1.1 The* MEMO *Data Structure.* The MEMO data structure in Cascades provides a compact representation of the search space of plans. In addition to enabling memoization (a variant of dynamic programming), a MEMO provides duplicate detection of operator trees, cost management, and other supporting infrastructure needed during optimization. A MEMO consists of two mutually recursive data structures, which we call *groups* and *groupExpressions*. A *group* represents all equivalent operator trees producing the same output. Note that a *group* does not explicitly enumerate all its operator trees. Instead, it implicitly represents them by using *groupExpressions*. A *groupExpression* is an operator that has *groups* (rather than operators) as children. As an example, Figure 5(b) shows a MEMO for the simple query above, in which logical operators are shaded and physical operators have white background. In the figure, *group* 3 contains all equivalent expressions for $R \bowtie S$. Note that *groupExpression* 3.1, $Join(1,2)$, represents all operator trees whose root is $Join$ and two children that belong to groups 1 and 2, respectively. A MEMO compactly represents a very large number of operator trees. Also note that the children of physical *groupExpressions* point to the most efficient *groupExpression* in the corresponding groups. For instance, *groupExpression* 3.8 represents a hash join operator whose left-hand-child is the second *groupExpression* in group 1 and whose right-hand-child is the second *groupExpression* in group 2.

The MEMO also manages *groupExpression* properties, which themselves can be *logical* or *physical*. Logical properties are, by definition, shared across all *groupExpressions* in a *group* and associated with the *group* itself (e.g., the cardinality or key columns in the query fragment encoded in a group). Physical properties are specific to physical *groupExpressions* and vary within a group (e.g., the order of tuples and cost of a physical *groupExpression*).

*4.1.2 Optimization Tasks.* Optimization in Cascades is broken into several tasks, which mutually depend on each other [10]. Intuitively, the optimization of a query starts by *copying* the logical operator tree describing the input query into the initial MEMO (see Figure 5(b)). Then, the optimizer schedules the optimization of the group corresponding to the root of the original query tree (group 5 in the figure). This task triggers the optimization of smaller operator sub-trees and eventually returns the most efficient execution plan for the input query. This execution plan is *copied out* from the final MEMO and passed to the execution engine where it is evaluated. Figure 6 shows a simplified version of *OptimizeGroup* that incorporates relevant portions of the remaining optimization tasks. It takes as inputs a group $G$, required physical properties *RP*, and a cost upper-bound UB, and returns the most efficient physical *groupExpression* that satisfies *RP* and costs under $UB$ (or NULL if it cannot find one).

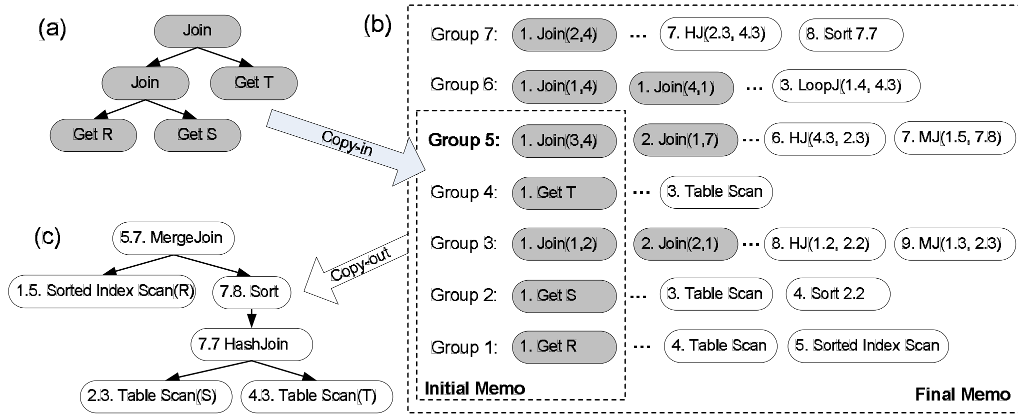



**Figure 5: The MEMO data structure in a Cascades-based optimizer.**

```
OptimizeGroup (G: group, RP: required property, B: cost)
returns groupExpression
 1 (p, isOpt) = winnerCircle[G, RP]
 2 if (isOpt) return (p ≠ NULL and p.cost < B) ? p : NULL
 3 bestP = p; B = p.cost
 4 for each enumerated physical groupExpression candP
 5    candP.cost = candP.localCost
 6    for each input p_i of candP
 7       if (candP.cost ≥ B) go back to 4 // out of bound
 8       G_i = group of p_i; RP_i = required property for p_i
 9       bestP_i = OptimizeGroup(G_i, RP_i, B - candP.cost)
10       if (bestP_i = NULL) break // no solution
11       candP.bestChild_i = bestP_i
12       candP.cost += bestP_i.cost
      // Have valid solution, update state
13    if (candP.cost < B and candP satisfies RP)
14       bestP = candP; B = candP.cost
15 winnerCircle[G, RP] = bestP
16 return bestP
```

**Figure 6: Simplified pseudocode for the *OptimizeGroup* task.**

Initially, line 1 checks the *winner circle* for a previous call *compatible* with *RP*. Specifically, it looks at the result of previous calls with parameters $(G, RP')$. Depending on the relationships between $RP$, $RP'$ and the delivered properties $DP$ of the resulting plan, it determines whether a previous solution $p$ can be reused for the current call. For instance, if there is a previous solution for which $DP \subseteq RP \subseteq RP'$, we know that the best solution for $RP'$ covers all plans in $RP$ and satisfies $RP$, so it can be returned immediately (we convey this information by setting `isOpt` to `true`). If $RP' \subseteq RP$ the resulting plan $p$ is valid for $RP$ but might not be the best one. In that case `isOpt` is `false` and we set $p$ as a starting point in line 3. So, if we find that an optimal plan $p$ for $RP$ has already been found, we return either $p$ (if it costs below `UB`) or `NULL` otherwise. If, instead, we do not find an optimal plan for $RP$ in the winner's circle, lines 4-14 recursively enumerate all physical *groupExpressions* in $G$ in search for the optimal plan (note that this step encapsulates the tasks in Cascades that apply transformation rules).

For each *groupExpression* $p$, line 5 estimates the local cost of `candP` (i.e., without accounting for its inputs). Then, line 8 calculates the input group and required properties for each of `candP`'s inputs and line 9 recursively calls *OptimizeGroup* to optimize them (the upper bound in the recursive call is decreased to `UB - candP.cost`). After each input is successfully optimized, lines 11 and 12 store the best implementation for each of `candP`'s children and update its partial cost. If, at any moment the current cost of `candP` becomes larger than `UB`, the candidate is discarded and the next one is considered (line 7). After `candP` is fully optimized, line 13 checks whether `candP` is the best plan found so far. After all candidates are processed, lines 14-15 add the best plan to the winner circle for $G$ and $RP$, and line 16 returns such plan.

## 4.2 Unified Query Optimization (UQO)

We now describe the extensions we made to SQL Server's optimizer to unify the three different optimization steps in the original PDW architecture (see Figure 1 for details). Specifically, we introduced new properties related to distribution, extended implementation rules and their corresponding required property propagation, and modified the information transferred across components.

*4.2.1 Property management.* To deal with distributed plans in a unified optimizer, we add a new physical property to *groupExpressions*, denoted *Distribution*, which determines how the data is distributed during query processing (similar to the approach in [21, 26]). Distributions can take one of the values described below [2].

**Serial/Control:** Data is not distributed. [3].

**Replicated:** Data is replicated on all nodes.

**Hash(cols):** Data is hash-distributed based on columns *cols*. We use a commutative hash function, so $Hash(\{a, b\})$ is equivalent to $Hash(\{b, a\})$.

**Any(cols):** Data is distributed on columns *cols* with an unknown distribution function ($Any(*)$ is a special value that

[2] Distribution specifications also require the degree of parallelism (i.e., how many partitions the data is divided on). For ease of exposition, we omit those details.

[3] Serial distributions are executed in any SQL BE node, while Control distributions are executed in the SQL FE for specific constructs that require metadata stored in the shell database (e.g., security predicates).

contains all columns in the schema, including a –possibly artificial– key (used for round-robin distributed tables).

**Wildcard** : Data can be distributed in any way *but* Replicated.

Table 1 shows implication rules for distribution properties. Recall that if data is distributed by columns $C$, then it must be that any two rows that agree on $C$ belong to the same partition. For that reason, if data is distributed by $C$, it is also distributed by $C' \supset C$ (but not the other way around). In the figure, $D_1 \implies D_2$ means that plans that satisfy $D_1$ are included in plans that satisfy $D_2$. For instance, $Hash(\{a\}) \implies Any(\{a,b\})$ and therefore a plan that delivers $Hash(\{a\})$ satisfies a requirement $Any(\{a,b\})$.

**Table 1: Distribution implication rules.**

| Distribution D1 | Distribution D2 such that $D1 \implies D2$ |
|---|---|
| Serial | Serial, Wildcard |
| Control | Control, Wildcard |
| Replicated | Replicated |
| Any(C) | Any(C') : C' ⊇ C, Wildcard |
| Hash(C) | Hash(C), Any(C') : C' ⊇ C, Wildcard |
| Wildcard | Wildcard |

Distributions are used as a *derived property* to determine how data is distributed during the execution of a *groupExpression*, and as a *required property* to request during the *OptimizeGroup* task of Figure 6. Derived properties are precise, so they cannot be $Wilcard$.

*4.2.2 Intermediate Interesting Properties.* Consider table $T(a,b,c)$ hash-partitioned on columns $\{a,b\}$ and the following query:

```
SELECT * FROM T T1 INNER JOIN T T2
ON T1.a = T2.a AND T1.b = T2.b AND T1.c = T2.c
```

For correctness, if we are evaluating the join in a distributed way, we need to ensure that any two rows that would join (i.e., that have the same values for columns $a$, $b$, and $c$) belong to the same partition. This can be achieved by requiring the inputs to the join to be distributed by $Hash(\{a,b,c\})$. However, this is not the only option. In fact, requiring the join inputs to be distributed on any subset of $\{a,b,c\}$, the result would be correct. For instance, a hash distribution on column $a$ satisfies that any pair of rows that agree on column $a$ are together, so obviously two columns that agree on columns $\{a,b,c\}$ are together as well. Since in our example table $T$ is already distributed on $Hash(\{a,b\})$, a better choice is to require that partitioning scheme, so that the whole query is evaluated without actual data movement.

It is tempting to require the inputs of the join to be distributed by $Any(a,b,c)$ since then, any distribution on subsets of $\{a,b,c\}$ would satisfy the request (and this is a good choice for a unary operator like GroupBy). However, join inputs must be distributed in the same way to prevent wrong results. Suppose that the join above is between tables $T$ and $U$, and table $U$ is distributed by $Hash(\{c\})$. In this case, a request of $Any(\{a,b,c\})$ to both inputs to the join would be satisfied separately by T with $Hash(\{a,b\})$ and $S$ with $Hash(\{c\})$. However, the results of such distributed join would be incorrect! This example illustrates why join requests must be fully specified (i.e., we cannot use *Any* distribution requests. However, there is an exponentially large number of possibilities to consider (i.e., each subset of the join columns). For our query, the choice of $\{a,b\}$ is interesting because the tables below are already hash distributed on $\{a,b\}$. Also, the choice of $\{a,b,c\}$ is interesting because it results in the lowest risk of data skew (if the number of distinct values of column subsets is too low). Any other choice will be less efficient than the best of the two options identified above.

In general, we need to identify a set of *interesting distributions* to consider as requirements for any given operator. Interesting distributions are those that might be *free* of data movement since another operator below (or the base tables themselves) are already distributed in the right way. We implement interesting distributions as another logical property of *groupExpressions*. The interesting distributions of an operator $Op$ are defined as the union of the interesting distributions of $Op$'s children[4], plus the specific interesting distributions that $Op$ itself might deliver. A few examples of interesting distributions are summarized in Table 2.

**Table 2: Interesting distributions for common operators.**

| Operator | Interesting distribution |
|---|---|
| Scan(T) | $T$'s distribution (e.g., $Hash(cols)$ if $T$ is hash distributed on *cols*). Round-robin tables do not return any distribution. |
| $\text{GroupBy}_{cols,aggs}$(P) | Any(cols). |
| $\text{Join}_{pred}$(L, R) | Any(join keys in *pred*). |
| Union(L, R) | Any(cols in the output schema). |

*4.2.3 Rules and property propagation.* We now describe how we extend implementation rules to consider required distributions. These rules leverage a primitive `Intersect`, shown in Figure 7, which returns the *most general* distribution $D$ that implies both input distributions $D_1$ and $D_2$ (or `NULL` if there is none). If either $D_1$ or $D_2$ is *Wildcard*, we return the other one in lines 1-2. Otherwise, it depends on the type of distribution of $D_1$. Lines 3-4 handle the serial, control, and replicated cases, and lines 8-15 handle the distributed cases (i.e., *Any* and *Hash*). For that purpose, we find the set of columns that would be visible in the output (given contextual properties $P$ of an operator) and belong to both $D_1$ and $D_2$ modulo column equivalences. If either $D_1$ or $D_2$ is $Hash$ distributed, then we do not allow column subsets (lines 13-14) and return `NULL`.

Using `Intersect` as a primitive, we show how to implement operators and propagate required distributions to their inputs. To understand the various algorithms that follow, we assume that the statement `Generate Op<D`$_i$`>` represents the calling of the original implementation rule (e.g., generating a hash-based or stream-based group-by alternative for `Op = GroupBy`) for which we pass down $D_i$ as the required distribution property to $Op$'s $i\text{-}th$ input.

*Group-By.* Figure 8 shows the relevant portion of implementing a group by operator given a required distribution. Specifically, if $G$ is a local/partial aggregate, we generate the appropriate physical operator in lines 1-2 and propagate down the distribution request (since those aggregates do not require specific input distributions

[4]Technically, some interesting distributions of the children are not propagated through an *incompatible* operator. We do this using an extension of the `Intersect` mechanism discussed in the next section and shown in Figure 7.

```
Intersect (D1, D2: Distribution, P: Properties)
returns distribution D such that
       D ⟹ D1 and D ⟹ D2
       or NULL if there is none.
 1 if (D1 = Wildcard [respectively, D2 = Wildcard])
 2    return D2 [respectively, D1])
 2 switch(D1)
 3    case {Serial, Control, Replicated}:
 4       return (D1 = D2) ? D1 : NULL
 5    case {Any(C1), Hash(C1)}:
 6       if (D2 ∉ {Any(C2), Hash(C2)})
 7          return NULL
 8       hasHash = D1 is Hash(C1) or D2 is Hash(C2)
 9       C = ∅
10       for each c1 ∈ C1:
11          if (∃ c ∈ P.visible, c2 ∈ C2 such that
                P.equivalent(c, c1, c2))
12            C = C ∪ {c}
13          else if (hashHash)
14            return NULL
15       return hasHash ? Any(C) : Hash(C)
```

**Figure 7: Intersecting distribution properties.**

for correctness). We do the same if the required distribution is one of *Serial*, *Control*, or *Replicated* in lines 4-5. For $Any(C)$ or $Hash(C)$ required distributions, we attempt two strategies. First, we intersect the required distribution with a manufactured distribution $Any(keys)$ where *keys* are the grouping columns (lines 10-11). For instance if the grouping columns are $\{a, b, c\}$ and the required distribution if $Any(c, d)$ it will propagate $Any(c)$ to its input. Second, we attempt the same strategy by further intersecting the previous alternative with each of the interesting distributions from the input to the group by. Finally, for a required distribution of $Wildcard$, we generate a serial distribution in line 7 and either return if the grouping keys are empty (e.g., a scalar aggregate) or fall through the treatment of $Any/Hash$ otherwise.

```
GenerateGBs (GB: GroupBy, D: required distribution)
 1 if (GB is local/partial aggregate)
 2    Generate GB<D> and return
 3 switch(D):
 4    case {Serial, Control, Replicated}:
 5       Generate GB<D> and return
 6    case Wildcard:
 7       Generate GB<Serial>
 8       if (GB.keys = ∅) return
         // fallthrough
 9    case {Any(C), Hash(C)}:
         // (a) based on keys
10       DK = Intersect(D, Any(GB.keys), GB.props)
11       if (DK != NULL) Generate GB<DK>
         // (b) based on interesting distributions
12       for each DI ∈ GB.input.IDs
13          DI' = Intersect(DI, DK, GB.props)
14          if (DI' != NULL) Generate GB<DI'>
```

**Figure 8: Generating Group-By operators.**

```
GenerateUnions (U: Union[All], D: reqd distribution)
 1 switch(D):
 2    case {Serial, Control, Replicated}:
 3       Generate U<D> and return
 4    case Wildcard:
 5       Generate U<Serial>
 6       if (U.UnionAll) Generate U<Any(U.columns)>
         // fallthrough
 7    case {Any(C), Hash(C)}:
         // (a) based on keys
 8       DK = Intersect(D, U.columns, U.props)
 9       if (DK != NULL) Generate U<DK>
         // (b) based on interesting distributions
10       for each DI ∈ ∪i U.input_i.IDs
11          DI' = Intersect(DI, DK, U.props)
12          if (DI' != NULL) Generate U<DI'>
```

**Figure 9: Generating *Union* and *UnionAll* operators.**

*Union/UnionAll.* Figure 9 shows the algorithm to generate n-ary *Union* and *UnionAll* operators. It is similar to the algorithm of Figure 8 for group-by operators with the following difference. For a *UnionAll* operator $U$, we generate an alternative that requests $Any(U.columns)$ in line 6. Due to the semantics of *UnionAll*, results are correct even if the inputs are distributed by different columns. In that case, the derived distribution property *UnionAll* could be any of those of its inputs (we pick an arbitrary one). We also slightly abuse the notation and write `Generate U<D>` when we require distribution $D$ from each input of the *Union*[*All*] operator.

*Joins.* Figure 10 shows the algorithm to generate distributed-aware joins. For simplicity, we restrict our attention to inner and outer join variants, but the cases for semi- and anti-joins are derived analogously. As with the previous operators, required distributions *Serial*, *Control* and *Broadcast* simply implement the join and propagate the required distribution to its inputs (lines 2-3). For $Any(C)$ and $Hash(C)$ we attempt two strategies (except for full outer joins, which only implement serial alternatives). First, we generate an alternative based on the full join keys (line 8). Second, we try all interesting distributions coming from both inputs (lines 9-12). For that purpose, we use the `GenerateJoinsHelper` method shown in Figure 10 which, given a required distribution $D$, generates distributed and replicated alternatives. Specifically, lines 1-2 normalize right outer joins into left outer joins to simplify subsequent processing. Line 5-7 generate replicated joins whenever appropriate (e.g., left joins cannot replicate the left side). Lines 8-12 generate distributed joins by (a) obtaining the intersection of $D_L$ and $Any(join.keys)$ in line 8, (b) downgrading the intersection to $Hash$ so that the input requests are fully specified in line 10, (c) mapping the distribution to the join right side using the left- and right-key columns in line 11, and (d) generating the distributed join in line 12.

4.2.4 *Enforcers.* The Cascades framework supports *enforcers*, or *glue operators*, which are special implementation rules that match arbitrary input trees and insert physical operators with the objective of satisfying required properties that otherwise would generate no plans. The most common enforcer is the *order enforcer* which inserts a physical *Sort* operator to satisfy a required order, and itself

```
GenerateJoins (J: Join, D: required distribution)
 1 switch(D):
 2    case {Serial, Control, Replicated}:
 3       Generate Join<D, D> and return
 4    case Wildcard:
 5       Generate Join<Serial, Serial> and fallthrough
 6    case {Any(C), Hash(C)}:
 7       if (J.type ≠ FULL)
            // (a) based on keys
 8          GenerateJoinsHelper(J, D)
            // (b) based on interesting distributions
 9          for each D_I ∈ J.left.IDs ∪ J.right.IDs
10            D_I' = Intersect(D, D_I, J.props)
11            if (D_I' ≠ NULL)
12              GenerateJoinsHelper(J, D_I')
```

```
GenerateJoinsHelper (J: Join, D: required distribution)
requires D ∈ {Wildcard, Any(C), Hash(C)}
 1 if (J.type) = RIGHT
 2    Commute J so that it becomes a LEFT join
 3 D_L = Intersect(D, Any(J.left.columns))
 4 D_R = Intersect(D, Any(J.right.columns))
   // Try replicated alternatives
 5 Generate Join<D_L, Replicated>
 6 if (J.type = INNER)
 7    Generate Join<Replicated, D_R>
 8 D_KL = Intersect(D_L, Any(J.key_left), J.props)
 9 if (D_KL != NULL)
      // Try distributed alternative
10    if (D_KL = Any(C')) D_KL = Hash(C')
11    D_KR = map D_KL using {J.key_left → J.key_right}
12    Generate J<D_KL, D_KR>
```

**Figure 10: Generating join operators.**

requires no order from its input. Analogously, we implement a new enforcer for distribution properties that inserts a physical *Redistribute* operator, which performs data movement and guarantees that the required properties are satisfied. The distribution enforcer kicks in whenever the required distribution is not $Wildcard$, places a specific data move variant from the source to a target distribution, and requires $Wildcard$ distributions from its input.

Different possibilities of redistribute operators are shown in Figure 3. For instance, to go from a serial distribution (in a backend node) to the control node we use an $SN$ (Single Node Move) operation that moves data from one node to another. Other redistribute operations include $M$ (Merge Move) which coalesces all data that resides in multiple distributions into a single node, $B$ (Broadcast Move) which transfers data from each source distribution into all the target nodes, $H$ (Hash Move) which hashes all data in the source distributions and sends them to the appropriate target distributions, $T$ (Trim Move) which locally hashes a replicated distribution and keeps the relevant portion of data in the target distributions, and $A$ (Any Move) which is implemented as $H$ for a target of $Any(C)$ of a round-robin move for $Any(*)$.

*Distribution column sets vs. multisets.* In this work we treat distribution columns as sets to simplify the presentation. In reality, we need to treat them as *multisets* to avoid subtle wrong behavior. Consider tables $T1(a, b, c)$, hash-partitioned by $\{a, b\}$, and $T2(a, b, c)$, hash-partitioned by $\{a\}$. Suppose we have the following query:

```
SELECT a, b, COUNT(*)
FROM (SELECT a FROM T1 WHERE a=b)
INNER JOIN T2
ON T1.a=T2.a
```

Recall that table $T_1$ is distributed by $Hash(\{a, b\})$. After the filter $a = b$ the compiler can canonicalize columns picking $a$ as the representative of the equivalence set $\{a, b\}$. If we consider distribution columns as sets, we would say that the output of the filter is distributed by $Hash(\{a, a\}) = Hash(\{a\})$. This, in turn might cause the join to be evaluated without any data move operation, since both inputs are hash-distributed in the same way modulo column equivalences. This plan, however, is not correct.

A distribution $Hash(\{a, b\})$ is not equivalent to $Hash(\{a\})$ when $a = b$. The former would hash the same value twice and combine the hashes, which in general produces a different result than hashing the value once. In reality, using multisets for distribution columns, we see that the distribution of $T1$ is $Hash(\{a : 1, b : 1\})$. After the filter $a = b$ this is transformed into $Hash(\{a : 2\})$ which is different from $Hash(\{a : 1\})$, so a redistribute operator on $Hash(\{a : 1\})$ would be inserted before the join.

**Table 3: Different enforcers from source to target distributions, where SNM = Single Node, BM = Broadcast, HM = Hash, MM = Merge, TM = Trim, and AM = Any.**

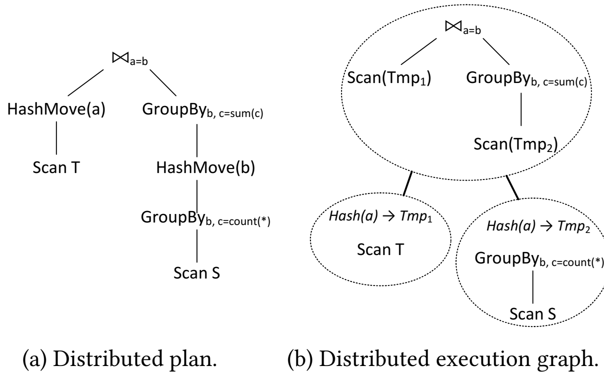
| From\To | Serial | Control | Replicated | Any(Y) | Hash(Y) |
|---|---|---|---|---|---|
| Serial | N/A | SN | B | A | H |
| Control | SN | N/A | B | A | H |
| Replicated | SN | SN | N/A | A | T |
| Any(X) | M | M | B | A | H |
| Hash(X) | M | M | B | A | H |

*4.2.5 Plan transfer.* As shown in Figure 4, SQL FE obtains the distributed execution plan from UQO and sends it over to the DCP. For that purpose, it uses a serialization format similar to Substrait [22]. Consider as an example the query below:

```
SELECT T.a, COUNT(*) AS C
FROM T INNER JOIN S
ON T.a = S.b
GROUP BY T.a
```

Assuming that $T.a$ is a key of $T$, a possible distributed plan for this query, which pushes the aggregation below the join, is shown in Figure 11(a). SQL FE sends this plan to the distributed engine, where is transformed into a graph of steps. Conceptually, this is done by partitioning the distributed plan across data move operations (keeping the move operation in both sides) and (i) replacing the producing side of a data move operation with an annotation on how data should be moved, and which distributed temporary table would hold the result, and (ii) replacing the consuming side of a data move operation with a Scan over such temp table. This transformation is illustrated in Figure 11(b).

Beyond the simple transformation around data move operations, the distributed engine does not need to understand the semantics of the plans, which are sent directly to backend nodes. This mechanism addresses the shortcomings illustrated in Figure 3 by eliminating decoding/optimizing intermediate steps. By sending plans instead of SQL statements, backend nodes are guaranteed to execute the same plan that UQO obtained. For the example in Section 3, the distributed engine sends precisely the plan in Figure 3(a), which includes operators that are complex to represent in SQL (e.g., Assert) or impossible without extensions (e.g., Merge statements).




(a) Distributed plan. (b) Distributed execution graph.

**Figure 11: From distributed plans to a execution graphs.**

*4.2.6 Other improvements.* We next mention capabilities that are possible due to our unified treatment of query optimization.

- **Optimization stages and timeouts:** Since the unified optimizer knows distributed plan costs, it leverages existing SQL Server's infrastructure that progressively increases the search space based on the cost of the best plan found so far. In contrast, the absence of a unified cost model in the original PDW approach required the frontend optimization to cover portions of the search space that could have been pruned with additional cost information.
- **Distribution-aware join reordering:** Queries with tens of joins cannot be fully explored, so optimizers typically resort on heuristics to prune the search space. If exhaustive exploration is not feasible, SQL Server locally explores join orders around some initial seeding plans in the MEMO. Having knowledge about table distributions during UQO allows us to generate new heuristic starting points that would have not been necessarily explored otherwise for large queries.
- **Better placement of local aggregation:** Local aggregation can be pushed below joins. In principle, there can be multiple levels of intermediate aggregation between the local and global operators. The space of alternatives is so large that optimizers typically choose some alternatives in a heuristic manner. These heuristics can make more informed decisions if considering plan generation and distribution placement at the same time, which is harder to do in the earlier PDW compilation process.

# 5 EXPERIMENTAL RESULTS

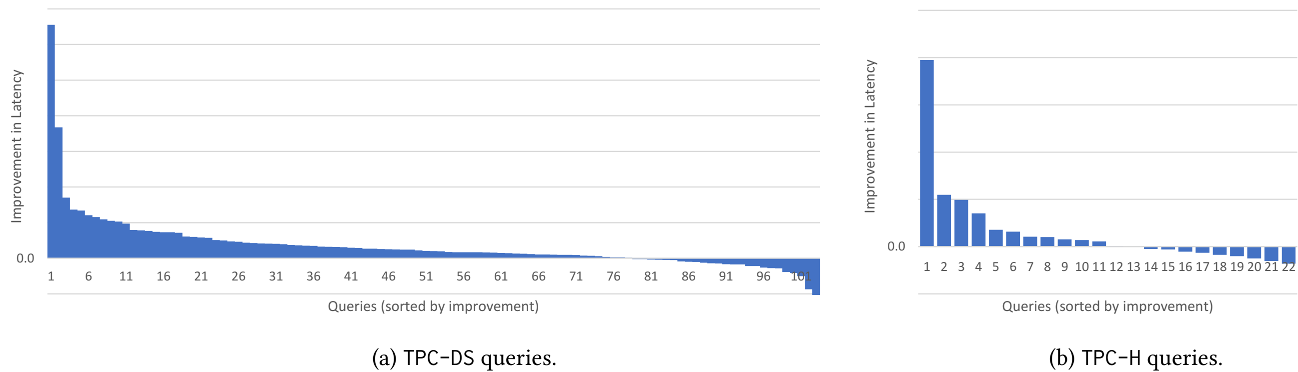
In this section we report initial results for the new unified optimization framework in Fabric DW. We used the TPC-DS and TPC-H benchmarks [24] with a scale factor of 10TB. For TPC-H we partitioned lineitem and orders by column orderkey and left the other tables as round-robin. For TPC-DS we partitioned tables item, inventory, store_sales, catalog_sales, web_sales, catalog_returns, store_returns, and web_returns by column item_sk and left the other tables as round-robin. For the evaluation, we used a DW-15000c cluster with 30 compute nodes. Overall, our approach resulted in an 8.3% improvement in latency and 8.9% improvement in geometric mean for TPC-H, and a 10.7% improvement in latency and 11.1% improvement in geometric mean for TPC-DS.

Figure 12 reports results for individual queries in the benchmarks. Each bar in the figure compares the latency of a single query with and without our new unified optimization framework[5]. The magnitude of each bar represents the absolute latency improvement of the new Fabric optimizer compared to the previous one. Thus, positive values are improvements, while negative values are regressions. Overall, around two thirds of the workload queries have comparable latency in both frameworks. This is a favorable outcome since both TPC-H and TPC-DS have been carefully tuned over the years in the PDW optimizer. The remaining third of the workload improves from 15% to 40% when using our approach.

For large queries, the initial distribution-aware join ordering produces better plans. Additionally, queries with heavy aggregation benefit from a more systematic placement of local grouping operators. Finally, some queries that reuse sets of columns in multiple joins and aggregation benefit from intermediate interesting properties and avoid some amount of data movement altogether. Additionally, the optimization process itself is faster. The most dramatic example is query Q64 in TPC-DS, which joins 18 tables together. The PDW compiler takes a long time traversing the search space with rather limited pruning capabilities (see Section 4.2.6). We see that PDW creates 2,526 *groups*, explores 22,050 logical *groupExpressions*, and implements 10,384 physical *groupExpressions* before timing out. In contrast, the unified optimizer creates 291 groups (11% of those in PDW), explores 546 *groupExpressions* (2.5% of those in PDW), and implements 8,468 physical *groupExpressions* (81% of those in PDW, but in this case optimization finishes without timing out and therefore obtains a better overall plan). The query compiles 14 times faster and produces a better-quality plan.

In addition to faster compilations and better-quality execution plans, we found other qualitative benefits of the new approach. First, we were able to leverage SQL Server native tools (e.g., more detailed plan visualizations). We also found improved development agility due to a consolidated framework. In the past, we had to carefully orchestrate new features that spanned over multiple optimization phases, and API changes were slow to ship. The overall code complexity was reduced, mostly by avoiding going back to intermediate SQL fragments and having a unified search strategy and cost models. Finally, the resulting unified optimization framework allowed other incubations and initiatives within Microsoft to leverage the new optimizer with fewer dependencies.

[5]The aim of this figure is to compare both executions but not to show absolute performance numbers. We therefore omit the y-axis legends on the figures.




(a) TPC-DS queries.

(b) TPC-H queries.

**Figure 12: Latency improvements using the unified query optimizer.**

## 6 RELATED WORK

The last decade has witnessed the emergence of new MPP offerings that store and manage large amounts of distributed data [3, 6, 12, 18, 20]. Query optimization and execution strategies have evolved since then to cope with new challenges in such distributed environments.

Some systems fully delay some plan decisions until execution time. In BigQuery [12], query execution plans can dynamically change during runtime based on statistics collected during query execution. For example, BigQuery starts processing hash joins by shuffling data on both sides. If one side finishes fast and is below a broadcast data size threshold, it cancels the second shuffle and executes a broadcast join instead. Snowflake [6] reduces the plan space by postponing choices like the type of data distribution for joins until execution time, to increase robustness at the cost of a loss in peak performance. In SingleStore [18], when join conditions or group-by columns match the tables' shard keys, execution is pushed to individual partitions avoiding data movement. Otherwise, SingleStore redistributes data during query execution, performed as a broadcast or reshuffle operation. Redshift [3] leverages the underlying distribution keys of participating tables to avoid unnecessary data movement. For instance, if a join key matches the underlying distribution keys of both participating tables, then the chosen plan avoids any data movement by processing the join locally for each data partition. While partitioning choices can be leveraged by subsequent operators in an opportunistic manner by Redshift, there is no global view of the query plan which would allow picking shuffle strategies that are useful to multiple operators in the plan. The dynamic schemes of these systems are useful to adapt plans during execution and avoid bad alternatives statically chosen based on bad cardinality estimates. At the same time, adaptive strategies are orthogonal to the choice of the *initial* plan. Our approach attempts to find such best initial alternative based on a global analysis of the query. Given reliable estimates, this approach performs better than pure dynamic alternatives that change plans at runtime. This adaptivity, however, can be used *after* the initial plan is chosen to correct decisions that were found to be suboptimal at runtime.

A different line of work, which is closer to our approach, attempts to model distributions explicitly during compilation, and find the optimal expected plan, looking holistically at the whole query. Examples of earlier work in this area include the Scope system [26] and the Orca modular optimizer [21] [6]. Similar to our approach, these systems model distributions via required and derived properties, and use enforcers to glue together plans that have mismatched distribution properties. The main contribution of our work, compared to these alternatives, is the notion of interesting *intermediate* distribution properties. This approach can identify the smallest set of partition requirements that need to be considered and avoids both brute-force enumeration of alternatives (which can be prohibitively expensive for complex queries) or heuristic approaches (which can miss optimal solutions). Our techniques are especially useful for optimizing complex query plans that combine multiple operations (e.g., joins or aggregates). These plans typically *share* portions of the required distribution keys across operators, which can benefit from a careful placement of shuffle operators.

## 7 CONCLUSIONS

The MPP data warehouse offering at Microsoft underwent several evolutions throughout the last 15 years. It was initially launched with an appliance form-factor (PDW), transformed to a cloud offering (Synapse DW) and later into a stateless all-in-one solution that converges data lakes and warehouses (Fabric DW). The original query optimizer was built on top of the (virtually unmodified) SQL Server optimizer, which was a pragmatic choice that quickly resulted in a sophisticated cost-based optimizer for PDW. Over time, architectural changes in the product and lessons learned from production required that the optimizer itself evolve as well. We showed how we unified separate query optimizers into a unified framework that can reason with distribution properties and allows implementing improvements that span across the original component boundaries of the original system. This new architecture makes the optimizer simpler to reason with, more efficient at optimizing queries, and produces better-quality plans than its predecessor. It also opens the door for exciting new opportunities that were not possible or unnatural to design in the old architecture, such as a better handling of common subexpressions and distributed bitmap filters, or fine-grained control on resource management.

[6] Interestingly, both Scope and Orca are also based on the Cascades framework.

# REFERENCES


- [1] Josep Aguilar-Saborit, Raghu Ramakrishnan, et al. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endowment* 13, 12 (2020), 3204–3216.
- [2] Josep Aguilar-Saborit, Raghu Ramakrishnan, Kevin Bocksrocker, Alan Halverson, Konstantin Kosinsky, Ryan O'Connor, Nadejda Poliakova, Moe Shafiei, Haris Mahmood Ansari, Bogdan Crivat, Conor Cunningham, Taewoo Kim, Phil Kon Kim, Ishan Rajesh Madan, Blazej Matuszyk, Matt Miles, Sumin Mohanan, Cristian Petculescu, Emma Rose Wirshing, and Elias Yousefi Amin Abadi. 2024. Extending Polaris To Support Transactions. In *SIGMOD '24: International Conference on Management of Data*. ACM, Santiago, Chile, June 9 - June 25, 2024.
- [3] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data*. ACM, Philadelphia, PA, USA, June 12 - 17, 2022, 2205–2217.
- [4] Keith Billings. 1997. *A TPC-D Model for Database Query Optimization in Cascades*. Master's thesis. Portland State University.
- [5] Pedro Celis. 1996. The Query Optimizer in Tandem's new ServerWare SQL Product. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*. Morgan Kaufmann, Mumbai (Bombay), India, 592.
- [6] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD '16: International Conference on Management of Data*. ACM, San Francisco, CA, USA, June 26 - July 01, 2016, 215–226.
- [7] delta-io. 2020. Delta Lake. https://github.com/delta-io/delta.
- [8] Mostafa Elhemali and Leo Giakoumakis. 2008. Unit-testing query transformation rules. In *Proceedings of the 1st International Workshop on Testing Database Systems, DBTest 2008*. ACM, Vancouver, BC, Canada, June 13, 2008, 3.
- [9] Apache Software Foundation. 2021. Apache Parquet. https://parquet.apache.org.
- [10] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.
- [11] Goetz Graefe. 1996. The Microsoft Relational Engine. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996*. IEEE, New Orleans, Louisiana, USA, 160–161.
- [12] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.
- [13] Jim Melton. 2000. *Understanding the New SQL: A Complete Guide, Second Edition, Volume I*. Morgan Kaufmann, Ed.
- [14] Microsoft. 2022. Azure Synapse SQL architecture. https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/overview-architecture.
- [15] Microsoft. 2023. https://learn.microsoft.com/en-us/sql/relational-databases/performance/live-query-statistics. https://learn.microsoft.com/en-us/sql/relational-databases/performance/live-query-statistics.
- [16] Microsoft. 2023. Microsoft Fabric. https://www.microsoft.com/en-us/microsoft-fabric.
- [17] Microsoft. 2023. Optimized plan forcing with Query Store. https://learn.microsoft.com/en-us/sql/relational-databases/performance/optimized-plan-forcing-query-store.
- [18] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric N. Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *SIGMOD '22: International Conference on Management of Data*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, Philadelphia, PA, USA, June 12 - 17, 2022, 2340–2352.
- [19] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD '79: International Conference on Management of Data*. ACM, Boston, Massachusetts, USA, 23–34.
- [20] Srinath Shankar, Rimma V. Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David J. DeWitt, and César A. Galindo-Legaria. 2012. Query optimization in microsoft SQL server PDW. In *SIGMOD '12: International Conference on Management of Data*. ACM, Scottsdale, AZ, USA, 767–776.
- [21] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *SIGMOD '14: International Conference on Management of Data*. ACM, Snowbird, UT, USA, June 22-27, 2014, 337–348.
- [22] substrait-io. 2021. Substrait: Cross-Language Serialization for Relational Algebra. https://github.com/substrait-io/substrait.
- [23] Teradata. 2016. Teradata Unified Data Architecture. www.teradata.com/ solutions-and-industries/unified-data-architecture.
- [24] TPC. 2023. TPC Benchmarks. https://www.tpc.org/.
- [25] Florian M. Waas. 2008. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *Business Intelligence for the Real-Time Enterprise BIRTE*. Springer, Auckland, New Zealand, 89–96.
- [26] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB Journal* 21, 5 (2012), 611–636.