Query Optimization in Microsoft SQL Server PDW*

Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung,
Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian,
David DeWitt, César Galindo-Legaria
Microsoft Corporation
srinaths@microsoft.com, rimman@microsoft.com, jaguilar@microsoft.com,
kchung@microsoft.com, mostafae@microsoft.com, alanhal@microsoft.com,
errobins@microsoft.com, msankar@microsoft.com, dewitt@microsoft.com,
cesarg@microsoft.com

ABSTRACT

In recent years, Massively Parallel Processors have increasingly been used to manage and query vast amounts of data. Dramatic performance improvements are achieved through distributed execution of queries across many nodes. Query optimization for such system is a challenging and important problem.

In this paper we describe the Query Optimizer inside the SQL Server Parallel Data Warehouse product (PDW QO). We leverage existing QO technology in Microsoft SQL Server to implement a cost-based optimizer for distributed query execution. By properly abstracting metadata we can readily reuse existing logic for query simplification, space exploration and cardinality estimation. Unlike earlier approaches that simply parallelize the best serial plan, our optimizer considers a rich space of execution alternatives, and picks one based on a cost-model for the distributed execution environment. The result is a high-quality, effective query optimizer for distributed query processing in an MPP.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Parallel databases; H.2.4 [**Database Management**]: Query processing

Keywords

Query Optimization

1. INTRODUCTION

1.1 Overview of SQL Server PDW

One of the major trends in recent years has been the wide adoption of Massively Parallel Processing (MPP) systems, i.e., distributed systems consisting of multiple independent nodes connected by a network. MPP systems are typically used as *data warehouses*, that is they are used to manage and query vast amounts of data. Microsoft SQL Server Parallel Data Warehouse [9, 10] is a shared-nothing parallel database *appliance* and is one example of an MPP system.

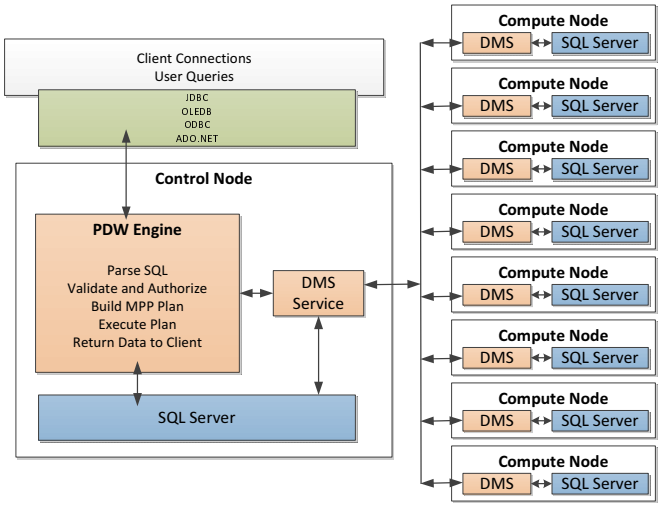

Figure 1: Microsoft SQL Server PDW.

SQL Server PDW comes in a number of hardware configurations, with the necessary software pre-installed and ready to use. It has a *control node* that manages a number of *compute nodes* (see Figure 1). The control node provides the external interface to the appliance, and query requests flow through it. The control node is responsible for query parsing, creating a distributed execution plan, issuing plan steps to the compute nodes, tracking the execution steps of the plan, and assembling the individual pieces of the final results into the single result set that is returned to the user. Compute nodes provide the data storage and the query processing backbone of the appliance. The control and compute nodes each have a single instance of SQL Server RDBMS running on them. User data is stored in tables that are hash-partitioned or replicated tables across the SQL Server instances on the compute nodes.

To execute a query, the control node transforms the user query into a distributed execution plan (called *DSQL plan*) that consists of a sequence of operations (called *DSQL Operations*). At a high-level, every DSQL plan is composed of two types of operations: (1) *SQL operations*, which are SQL statements to be executed against the underlying compute nodes' DBMS instances, and (2) *DMS operations*[1] which are operations to transfer data between DBMS instances on different nodes.

*PDW is short for Parallel Data Warehouse.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

[1] *DMS* is short for *Data Movement Service*.

1.2 Query Optimization in PDW

Queries executed in MPP environments tend to be complex – involving many joins, nested sub-queries and aggregations – and are usually long-running and resource-intensive. The goal of the query optimizer is to find the best execution plan for a given query, which is usually accomplished by examining a large space of possible execution plans and comparing these plans according to their estimated execution costs.

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. For a commercial product, time to market is also a critical dimension to consider. Rather than starting from scratch, the PDW query optimizer reuses the technology developed for SQL Server, which has been tuned and tested over a number of releases. This allowed us to craft in a short time a query optimizer for parallel queries that is rich in functionality and produces high-quality plans.

PDW invokes the SQL Server QO against a "shell database" to obtain a compact representation of the *optimization search space* called a `MEMO` [5, 6]. This search space is then augmented with statistical information on the distribution of data in the appliance to find a parallel execution plan for the query. , taking into consideration the available statistics and the actual data distribution in the appliance.

The main idea of PDW QO can be summarized as follows:

1. We store the metadata of the distributed tables in a "shell database" on a single SQL Server instance. The "shell database" provides the "single system image" of the data in the appliance. Importantly, it also stores aggregated statistical information on the user data.
2. Using the shell database, we use the existing compilation stack of SQL Server to parse a given query, and generate and export the space of execution alternatives (`MEMO`).
3. We traverse the space of execution alternatives to introduce data movement operations, and make a cost-based decision on the best execution plan for the distributed environment.

The rest of the paper is organized as follows. We first give an overview of the Microsoft SQL Server PDW architecture in Section 2. We provide detailed discussion of PDW QO implementation in Section 3. Section 4 contains detailed query example illustrating the flow of query optimization. Finally, we conclude in Section 5.

2. MICROSOFT SQL SERVER PDW ARCHITECTURE OVERVIEW

2.1 Appliance

The PDW appliance is composed of hardware and software architected to function together as one "box." Multiple servers are used to implement scale-out query processing in a shared-nothing fashion.

Users can only access the box through the user interface of the appliance. Industry standard hardware is used for servers, networking components and storage arrays. This approach allows cost effective and incremental growth of the appliance by adding extra servers or storage. Additionally, as the appliance grows over time the server components can be upgraded or individually replaced with newer generations of more powerful CPUs, memory, storage, etc.

There are two distinct types of nodes that implement the query processing functionality (see Figure 1).

1. **Control Node**. The control node manages the distribution of query execution across the compute nodes, accepts client connections to the PDW appliance and manages client authentication. In addition to containing a SQL Server instance, the control node contains additional software to support the distributed architecture of the PDW. This includes the engine that coordinates the data warehousing functions that are specific to processing parallel queries, stores appliance-wide metadata and configuration data, and manages appliance and database authentication and authorization. The control node also manages the Data Movement Service (DMS) (described in Section 2.3) that runs on the appliance nodes and is the communication layer for transfering data between the nodes in the appliance. A user can connect to the PDW control node using a variety of client access tools using the drivers with connection types, such as: ODBC, OLE DB, ADO.NET.
2. **Compute Nodes**. Each compute node is the host for a single SQL Server instance. It also runs a DMS process for communication and data transfer with the other nodes in the appliance. Each compute node stores a portion of the user data.

Tables in a PDW appliance can either be

1. replicated on each compute node in the appliance, or
2. hash-partitioned on a specified column(s) across the compute nodes.

2.2 Shell Database

A "shell database" is a SQL Server database that defines all metadata and statistics about tables, but does not contain any user data. From the point of view of compilation, authentication, authorization and query optimization, a shell database is undistinguishable from one that contains actual data. Shell databases are used by SQL Server for testing and debugging of compilation issues without having to copy large databases.

For PDW, a shell database residing on the SQL Server instance at the control node is used to store the metadata for the user tables partitioned across the compute nodes. It provides all the information needed to compile and generate a space of execution alternatives for queries. In addition to table metadata, we also store in this database all information regarding users and privileges, so that compilation can check for security and access rights. This enables PDW to provide the same security model as SQL Server, at no extra cost.

The shell database also contains global statistics for all the tables in the appliance. To compute global statistics, local statistics are first computed on each node via the standard SQL Server mechanisms, and are then merged together to derive global statistics..

2.3 Data Movement

The Data Movement Service (DMS) is responsible for moving data between all the nodes on the appliance. Once instance of DMS runs on each of the control and compute nodes. Certain steps of a user query may require intermediate result sets to be moved from one compute node to another. In addition, sometimes intermediate result sets from one or more compute nodes must be moved to the control node for final aggregations and sorting prior to returning the result set to the client. PDW utilizes temporary (*temp*) tables on the compute and control nodes as necessary to move data or store intermediate result sets. In some cases, queries can be written that generate no temp tables and results can be streamed from the compute nodes directly back to the client that issued the query – such queries will not involve DMS.

2.4 The DSQL Plan and its Execution

Given a user-specified SQL query, the PDW engine is responsible for creating a parallel execution plan (known as a DSQL plan). A DSQL plan may include the following types of operations:

- **SQL Operations** that are executed directly on the SQL Server DBMS instances on one or more compute nodes.
- **DMS Operations** which move data among the nodes in PDW for further processing, e.g. moving intermediate result sets from one compute node to another.
- **Temp table operations** that set up staging tables for further processing.
- **Return operations** which push data back to the client.

Query plans are executed serially, one step at a time. However, a single step typically involves parallel operations across multiple compute nodes.

DSQL Plan Example: Using the TPC-H schema as an example, let's assume that the *Customer* table is hash-partitioned on *c_custkey*, and the *Orders* table is hash-partitioned on *o_orderkey* and we want to perform the following join between these two tables.

```
SELECT c_custkey,
       o_orderdate
FROM Orders, Customer
WHERE o_custkey = c_custkey AND o_totalprice > 100
```

The table partitioning is not compatible with the join since *Orders* is not partitioned on *o_custkey*). Thus, a data movement operation is required in order to evaluate the query. The optimizer on the control node may produce a DSQL plan consisting of the following two steps:

1. *DMS Operation* that repartitions data in the *Orders* table on *o_custkey* in preparation for the join.
2. *Return SQL Operation* that selects tuples for the final result set from each compute node and returns them back to the client.

Once the plan has been generated, the Engine service executes the plan by walking the list of plan steps and distributing the steps one-by-one to the compute nodes.

Step 1: DMS Operation: In the example above, the first step in the DSQL plan is a DMS operation that repartitions *Orders* data on *o_custkey*. The DMS operation specifies the (1) SQL statement required to extract the source data, (2) the tuple routing policy (e.g., replicate or hash-partition on a particular column), and (3) the name of a (temporary) destination table. The Engine service then begins broadcasting the DMS operation from the control node to the DMS instance on each node. Upon receiving the DMS message, the DMS instance on each compute node begins execution of the data movement operation by issuing the SQL statement below:

```
SELECT o_custkey,
       o_orderdate
FROM Orders
WHERE o_totalprice > 100
```

against the local SQL Server instance. Each DMS instance reads the result tuples out of the local SQL Server instance, routes the tuples to the appropriate DMS process by hashing on *o_custkey*, and also inserts the tuples it receives from other DMS instances into the specified local destination table (*Temp_Table* in this example). Once all of the tuples from the source SQL statement have been inserted into their respective destinations the DMS operation is complete.

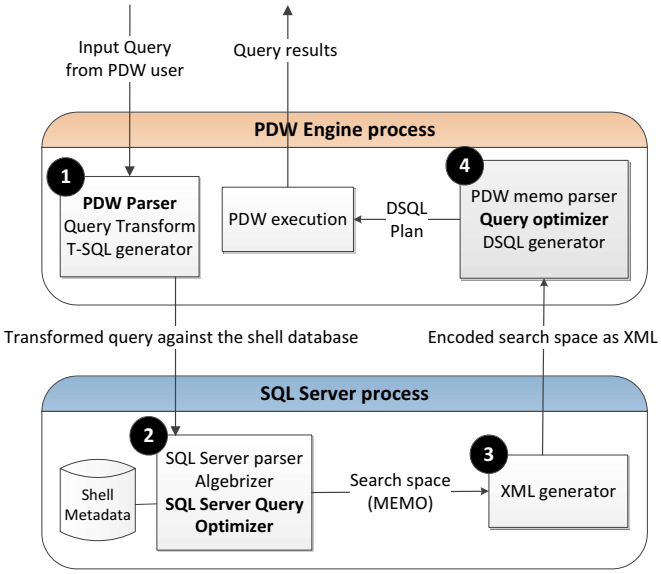

Figure 2: Overview of query optimization in PDW.

Step 2: SQL Operation: After the DMS operation has completed, the Engine service moves on to the second step in the plan, which is the SQL operation that is used to pull the result tuples from each compute node. To perform this operation, the Engine service obtains a connection to the SQL Server instance on each compute node and issues a specified SQL statement. In this case, the SQL statement that will be executed is:

```
SELECT c.c_custkey,
       tmp.o_orderdate
FROM Customer c,
     Temp_Table tmp
WHERE c.c_custkey = tmp.o_custkey
```

To complete the request, the Engine service reads the result tuples from each compute node, packages them into the final result, and sends them back to the client. The query execution is now complete. From the client's perspective, it appears as if all of the data was stored and all of the computation took place in a single SQL Server instance on the control node.

2.5 Cost-Based Query Optimization in PDW

Figure 2 provides the high-level data flow for PDW query optimization. The key observation that forms the basis of PDW QO is that the problem of a) algebrizing input queries into operator trees, and b) the *logical* (as opposed to physical, or partition-dependent) exploration done on operator trees to find plan alternatives is *the same* for PDW as it is for a single SQL server instance. We describe each of the QO components and their functionality below:

1. **PDW Parser**: This component is responsible for parsing the input query string and creating an abstract syntax tree (AST) structure that can be validated against PDW syntax rules. Some PDW queries may also need a few basic transformations before they are ready to be sent to SQL Server against the shell database.
2. **SQL Server Compilation**: After validation by the PDW parser, the query is passed to SQL Server for compilation against the shell database. The SQL Server optimizer performs the following functions:
   (a) Simplification of the input operator tree into a normalized form. This is inserted as the initial plan into the `MEMO` [5, 6] data structure, which will hold the space of alternative plans for the query.

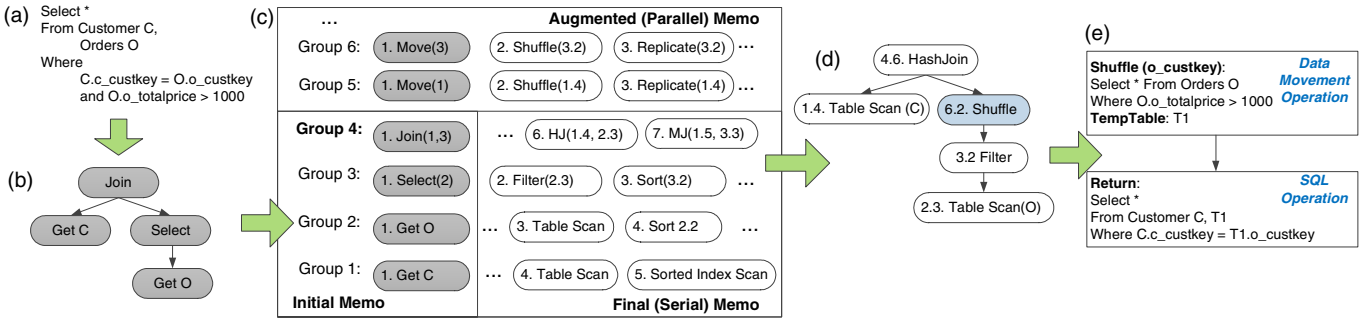

Figure 3: Parallel query optimization flow: (a) input query, (b) logical query tree, (c) augmented MEMO, (d) best query plan, (e) final DSQL plan.

(b) Logical transformations on the plans in the MEMO data structure to augment the set of choices. These are based on relational algebra rules. For instance, all equivalent join orders are generated in this stage.

(c) Estimation of the size of intermediate results for each of the execution alternatives. These estimations are based on the size of base tables and statistics on the column values.

(d) The implementation phase which adds physical operator (algorithms) choices into the search space. The optimizer costs them and prunes the plans that do not meet established lower bounds.

(e) Extraction of the optimal execution plan.

While the output of the SQL server optimizer is an "optimal" execution plan, the best serial plan (i.e., single-node execution plan) will not produce the best distributed execution plan. We elaborate on this later in this section.

PDW does not take the best plan obtained by the SQL Server optimizer; rather the PDW QO will consume the entire space of alternatives generated by the SQL QO.

3. **XML Generator**: This component takes the search space generated by SQL Server optimizer represented in the MEMO data structure as its input and encodes the information as XML.

4. **PDW Query Optimizer**: The PDW query optimizer is the consumer of the search space output from the XML generator. There is a memo parser on the PDW side which is responsible for constructing the memo data structure for the PDW query optimizer. Once the memo data structure for the PDW side is constructed, the PDW optimizer performs bottom-up optimization (see Section 3.2) with the help of the PDW cost model. The responsibilities of the PDW query optimizer include:

   (a) Enumeration of distributed execution plans by systematically adding appropriate data movement strategies into the search space.

   (b) Costing the alternative plans generated using the PDW cost model.

   (c) Choosing the optimal (minimal cost) distributed execution plan.

Once optimization process is completed, the DSQL generator component constructs the query plan for distributed execution of the query.

Example:

Consider the following SQL query:

```
SELECT *
FROM CUSTOMER C, ORDERS O
WHERE C.C_CUSTKEY = O.O_CUSTKEY
  AND O.O_TOTALPRICE > 1000
```

Figure 3 visually depicts the flow of parallel query optimization for the above query. We first parse the input query (Figure 3(a)) and transform it into a tree of logical operators (Figure 3(b)). Traditional query optimization is performed producing a MEMO. The MEMO consists of two mutually recursive data structures, called *groups* and *groupExpressions*. A group represents all equivalent operator trees producing the same output. To reduce memory requirements, a group does not explicitly enumerate all its operator trees. Instead, it implicitly represents all the operator trees by using *groupExpressions*. A *groupExpression* is an operator having other groups (rather than other operators) as children. As an example, consider Figure 3(c), which shows a MEMO for the query example above (logical operators are shaded and physical operators have white background). In the figure, group 1 represents all equivalent expressions that return the contents of table *Customer* (or *C*, for short). Some operators in group 1 are logical (e.g., *Get C*), and some are physical (e.g., *Table Scan*, which reads the contents of *C* from the primary index or heap, and *Sorted Index Scan*, which does it from an existing secondary index). In turn, group 4 contains all the equivalent expressions for $C \bowtie O$. Note that *groupExpression* 4.1 (i.e., *Join*(1,3)), represents all operator trees whose root is *Join*, first child belongs to group 1, and second child belongs to group 3. In this way, a MEMO compactly represents a potentially very large number of operator trees. Children of physical *groupExpressions* point to the most efficient *groupExpression* in the corresponding groups. For instance, *groupExpression* 4.6 represents a hash join operator whose left-hand-child is the fourth *groupExpression* in group 1 and whose right-hand-child is the third *groupExpression* in group 2. 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 query optimization. Additional details on the organization of the MEMO structure can be found in the literature [5, 6].

Once the memo from SQL Server (i.e. the serial MEMO) has been generated, it is augmented to additionally consider parallelism (see Figure 3(c)). The PDW optimizer then introduces new data movement groups and operations into the MEMO based on the underlying data distributions. For example, see groups 5 and 6 in the Figure 3(c) . Group 5, represents the data movement of the output of group 1 (i.e., the tuples from *C*). Assuming, *C* and *O* are distribution-*incompatible*, this operation would be considered by

the parallel optimizer as one of the options in order to make both C and O partition-compatible to perform the join $C \bowtie O$. Group 6, on the other hand, represents the data movement of the other input to the join, namely O. Like relational operations, logical data movement operations may have a number of physical implementations, such as *Shuffle* (re-partition of data on a column(s)), *Replication*, and so on. The final execution plan, which consists of a tree of physical operators, is extracted from the `MEMO` (shown in Figure 3(d)). THis plan is then transformed into an executable DSQL plan that will be run in the appliance (shown in Figure 3(e)).

Why Parallelizing The Best Serial Plan Is Not Enough

The optimization objectives for PDW and SQL Server are different [8]. The SQL Server optimizer is unaware of the partitioning of data, while PDW has the additional task of inserting and costing data movement operations to obtain a correct and efficient parallel plan. However, the problem of arriving at a logical search space is common to both, for the most part. Hence we export the logical search space from the SQL server process into PDW for optimization based on PDW objectives.

This is best illustrated with an example. Consider the TPCH database, with the *Customer*, *Orders* and *Lineitem* tables partitioned on *custkey*, *orderkey*, and *orderkey*, respectively. Consider a query that performs a join between these three tables on *custkey* and *orderkey*. The best serial plan may produce the following join order: *Customer*, *Orders*, *Lineitem*, in increasing order of table size. However, a better parallel plan may be obtained by choosing the following join order: *Orders*, *Lineitem*, *Customer*, i.e. join *Orders* and *Lineitem* first and then shuffle (i.e., redistribute) the result on *custkey*. This plan may be better due to the co-location of *Orders* and *Lineitem*.

3. IMPLEMENTATION OF PDW QO

Although the conceptual idea behind PDW QO is relatively simple, there were a number of technical challenges to address. We discuss them in detail in this section.

3.1 Changes to SQL Server

A few changes are needed in SQL Server to reuse its logic for PDW optimization.

The first change is to support exporting the optimizer search space. We defined a new compilation entry point to request the optimizer `MEMO`, in a way similar to the "`showplan XML`" functionality already available in SQL Server. This new entry point also triggers the use of any PDW-specific logic in the SQL Server compilation stack. When the PDW-compilation is requested, the output from SQL Server is an XML representation of the `MEMO` data structure.

A second change is to extend the query surface to support all constructs of PDW. The goal for PDW is to be fully compatible with SQL Server. So this extension is limited to a handful of query hints for specific distributed execution strategies.

The third change is to expand the optimizer search space to include some alternatives that are relevant for distributed query execution, especially around collocation of joins and unions. The transformation-based architecture of the query optimizer in SQL Server allows implementing these extensions without major changes to the framework.

For very large search spaces, the SQL Server optimizer uses a timeout mechanism and does not generate all possible plans. In those cases the initial execution alternatives placed in the `MEMO` have a big influence on the space considered. For PDW optimization, we "seed" the `MEMO` with execution plans that consider distribution information of tables, for collocated operations.

3.2 Plan Enumeration

The search space of possible plans is huge and by introducing data movement (DMS) operations we increase it even further. Naïve enumeration is not likely to be successful for any but the simplest of queries. In this section, we describe the bottom-up enumeration strategy in the PDW optimizer. While our current implementation employs a bottom-up search strategy, a top-down enumeration technique is equally applicable to the PDW QO design.

Steps 5-7 in Figure 4 depict the pseudo-code for the bottom-up search strategy. A bottom-up optimizer starts by optimizing the smallest expressions in the query, and then uses this information to progressively optimize larger expressions until the optimal physical plan for the full query is found. Bottom-up optimizers pay special attention to physical properties that affect the ability to generate the optimal plan. *Interesting properties* in the PDW query optimizer represent an extension of the notion of interesting orders introduced in System R [3]. For example, a subplan that returns results distributed on a certain column may be preferable to a cheaper alternative, because later in the plan this distribution can be leveraged to obtain a globally optimum solution. Specifically, the PDW query optimizer considers the following columns to be interesting with respect to data movements: (a) columns referenced in equality join predicates, and (b) group-by columns. Join columns are interesting because they make local and directed joins possible, and group-by columns are interesting because aggregations can be done locally at each node and the results unioned together, without having to do local / global processing.

3.3 Cost Model

To evaluate the performance of a specific plan, we use a cost model that costs data movement operations. Despite the robustness that costing relational operators would give to the cost model, costing DMS operations is a good start for developing a high quality cost based optimizer:

- Costing data movement operations is a subset of the overall "complete" cost model for plans involving both data movement and SQL relational operations. Thus, the scope is smaller and more manageable. The development, testing and debugging of the cost model is less complex compared to the mixed cost model involving relational operations.
- Data movement processing times tend to dominate queries overall execution times in PDW due to materializing data to temp tables. Thus, optimizing for data movements is expected to produce good quality plans for a broad set of queries.
- There is no equivalent to data movement operations inside SQL Server, thus we cannot rely on SQL Server optimizer to generate the costs for these operations.

3.3.1 Cost Model Assumptions

Building a cost model from scratch is a challenging task. Our current version of the cost model strives for simplicity while trying to ensure high quality. The scope of the current version of the cost model is to only cost DMS operations in terms of response time. With this purpose in mind the following assumptions hold:

- **Absence of independent parallelism**. The cost model assumes a sequential execution of the DSQL steps.
- **Absence of pipelined parallelism between DSQL steps**. Two steps in a consumer-producer relationship are not executed concurrently; intermediate results are always materialized.

```
PDWOptimizer ()
01 Parse the MEMO XML from SQL server into the PDW MEMO object.
02 Apply MEMO pre-processor rules (bottom-up).
   Example:  Fix cardinality estimates of partial aggregates based on PDW topology.
03 Merge equivalent group expressions from the perspective of PDW within the groups (bottom up).
04 Derive interesting properties of groups (top-down).
   /*** BOTTOM-UP ENUMERATION ***/
05 For each group in the MEMO, in bottom-up order, do the following:
06    Enumeration step:
06.i     Enumerate PDW optimization options by considering all possible inputs from child groups.
         If it's a base group, add a Get operation.
         Apply heuristic pruning.
06.ii    Cost based pruning:
         As PDW group expressions are added, keep only the overall best option
         and the best option per each interesting property.
         Do pruning every time a new PDW group expression is added into the MEMO.
         The maximum cardinality of the set of PDW group expressions in
         a group is (# of interesting properties + 1).  //one is added to account for overall best.
07    Enforcer step:
         Add PDW move group expressions based on interesting properties in the current group.
         Apply cost-based pruning similar to step 06.ii above.
   /*** END BOTTOM-UP ENUMERATION ***/
08 Extract the best overall plan by starting at the best plan in the root group and going
   down the memo to obtain the optimal plan tree.
09 Apply post-optimization rules on the optimal plan tree.
10 Perform DSQL-generation by traversing the optimal plan tree bottom-up.
11 Apply post-DSQL-generation rules on the DSQL plan.
12 Return the DSQL plan to the engine for execution.
```

Figure 4: Pseudo-code for PDW Optimizer (component 4 in Figure 2).

- **Isolation**. When costing a query, we assume it runs in isolation.
- **Homogeneity**. Homogeneous hardware across all nodes in the appliance (one topology).
- **Uniformity**. Uniform distribution of data across nodes. When costing an operator which is cloned and executed in multiple independent nodes, we could compute the cost of an operator as $max_{1 \leq i \leq N}(\text{C[i]})$ where N is the number of nodes, and each component of $\overline{C}$ its execution cost on a particular node. With the uniform and homogeneity assumptions, the components of the cost vector $\overline{C}$ are assumed to be identical, thus simplifying the cost model in the sense that only one node needs to be considered.

3.3.2 *Types of DMS Operations*

Before we present the details of our cost model, we first discuss the various physical data movement operations in PDW. There are 7 data movement operations:

1. **Shuffle Move** (many-to-many). Rows are moved from each compute node to target table based on a hash of the value in the specified distribution column.
2. **Partition Move** (many-to-one). Rows are moved from each compute node to the target table on the target node (typically the control node but this is not a requirement).
3. **Control-Node Move** (From the control node to the compute nodes). A table in the control node is replicated to all compute nodes.
4. **Broadcast Move**. Rows are moved from each compute node to the target table on all compute nodes.
5. **Trim Move**. Trim move is initiated against a replicated table on all compute nodes where the destination is to a distributed table on its own nodes. Hashing will take place so that only rows that this node is responsible for will be kept.
6. **Replicated broadcast**. A table which is only in one compute node it is replicated via a broadcast move.

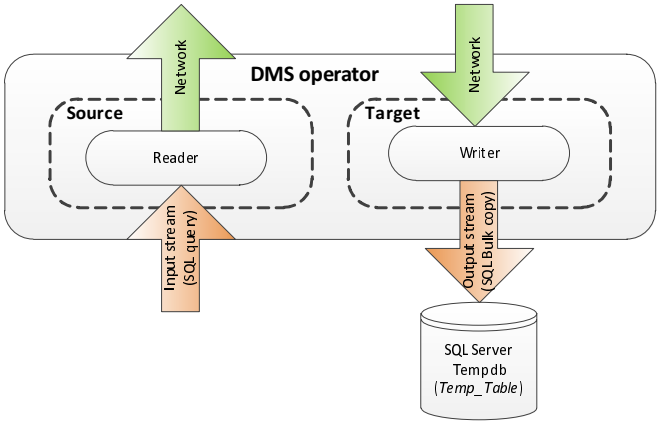

Figure 5: A DMS operator.

7. **Remote copy to single node**. Can be either a remote copy of a replicated table (from control node or from compute node) or, a Remote copy of a distributed table.

Each of the DMS operations defined above is implemented by a common runtime operator, the DMS operator. The cost of the DMS operator will be different depending on the operation being implemented.

3.3.3 *Cost of a DMS operator*

Figure 5, shows the basic structure of a DMS operator. We can envision a DMS operator as two components with separate functionalities that we name the source, and the target. The source of a DMS operator is the "sending side" and can be broken into the following two cost sub-components:

- C_{reader}: Read tuples from the query executed against SQL Server and packing them into a buffer.
- $C_{network}$: Send the data buffers over the network.

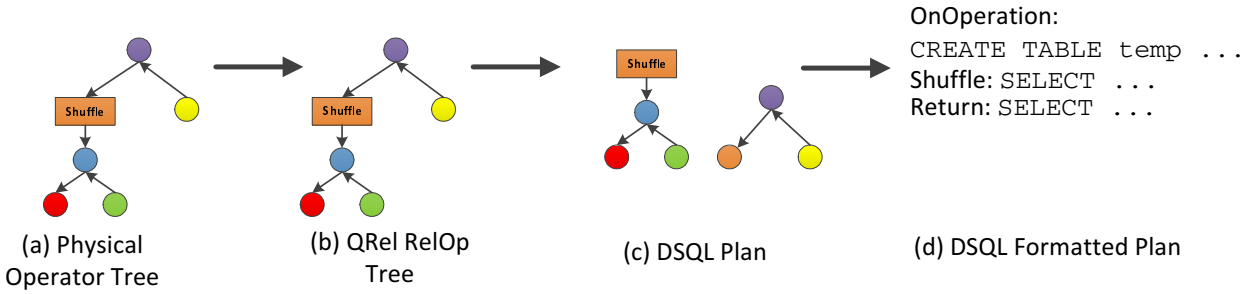

Figure 6: DSQL Generation.

The data is sent asynchronously over the network; therefore we can define the cost of the source component as the most expensive of its components: C_{source} = max(C_{reader},$C_{network}$).

The *target* in a DMS operator is the "receiving side" and, is also broken into two cost sub-components:

- C_{writer}: Unpack the tuples from the buffers sent by the sending process, and prepare buffers for insertion into a temporary table.
- $C_{SQLBlkCpy}$: Bulk copy operation for insertion of the data buffers into a SQL Server temporary table.

The bulk insert into the temporary table is done asynchronously, so similarly to the source component, the cost of the target can be defined as: C_{target} = *max*(C_{writer}, $C_{SQLBulkCpy}$).

The *source* and the *target* of a DMS operator run in parallel on each node. Therefore, we define the cost of a DMS operation as follows: C_{DMS} = max(C_{source}, C_{target}).

Costing of an Individual Component

Theoretically, the more complex the model the more accurate the estimates should be. However, it has also been proven that, the more sophisticated the cost model is, the more sensitive it is to slight changes in data and statistics [7]. Furthermore, it becomes more difficult to debug and maintain the cost model over time. Our current version of the cost model costs each individual component based on the number of raw bytes processed: $C_X = B * \lambda$, where B is the number of raw bytes and λ is the cost per byte for the specific component (C_X).

The constant λ is calculated via targeted performance tests after a meticulous instrumentation of the source code. We call the process of defining the value of λ for each cost component *cost calibration*. The results of our *cost calibration* showed that there are differences in the value of λ depending on the number of rows, number of columns or column type. However, the differences observed were not significant enough to justify stepping up the complexity of the cost model. Therefore, the value of λ is considered to be constant regardless of these parameters. Each cost component has its own constant value of λ. C_{reader}, however, was one exception and required two constants, denoted as λ_{hash} and λ_{direct}, to account for the extra overhead that hashing has for some data move operations e.g. *Shuffle* or *Trim*.

The number of bytes B to be processed by each individual cost component depends on the distribution properties of the input and output streams. Let Y denote global cardinality, and w – the width of the row (both values are provided by the statistics exported in the `MEMO`). Let N denote the number of nodes in the appliance. Then, under the uniformity assumption, we can compute B as:

- $(\frac{Y*w}{N})$ for distributed data streams, and
- $(Y * w)$ for replicated data streams.

3.4 DSQL Generation

Finally, once the query execution plan is selected by the PDW query optimizer, it must be translated to DSQL format to be able to run it on the actual compute nodes. Unlike other MPP systems, e.g., GreenPlum [2], instead of sending an operator tree to each compute node, SQL Server PDW sends a SQL statement to the compute nodes. The SQL statements are executed against the underlying compute nodes' standard DBMS instances, and data movement operations are used to transfer data between DBMS instances on different nodes. This is a similar approach to AsterData [1].

Performing DSQL generation requires taking an operator tree and translating it back to SQL. We employ the *QRel* programming framework [4], which encapsulates the knowledge of mapping relational trees to query statements. The process is depicted in Figure 6. A physical operator tree produced by the PDW query optimizer is first converted into a *RelOp* tree [4], which has structure similar to SQL server algebrizer output tree. *RelOp* tree is then converted into a *PIMOD* AST by the *QRel* library. Finally, the *PIMOD* script generator generates T-SQL string from the resulting AST. For more details on this process, we refer the reader to [4].

4. QUERY OPTIMIZATION EXAMPLE

Figure 7 is an example of the parallel plan generated by PDW for Q20 in TPC-H. The figure also shows the DSQL plan generation output for each section of the parallel query plan, separated by a red line in the Figure. For readability we don't show the temporary tables in the query plan, however, each data move operation is materializing the output of each DSQL step; in other words, DSQL steps are not pipelined.

Q20 is interesting from the point of view that exercises key features in query optimization like: sub-query removal, sub-query into join transformation, join transitivity closure detection etc.

- **DSQL steps 0 and 1**. The semi-join between *lineitem* and *part* – transformed into a join and a group by on *p_parkey* – is the result of de-correlating the two most inner sub-queries (SQ2 and SQ3), transforming them into joins, and detecting the transitivity closure between *ps_partsupp*, *part* and *lineitem*; this strategy allows the early filtering of lineitem, by joining it with part. Due to the high selectivity predicate on *part*, a broadcast join is the preferred option in this case (DSQL step 0); a *shuffle* of *lineitem* is also considered but results into a more expensive strategy. After the broadcast of part, we shuffle the output of the join between *part* and *lineitem* (DSQL step 1) on *l_partkey* to compute the group by aggregation (sum(*l_quantity*)) in a distributed manner. Note

that the cost model opts for a local-global transformation of the group by aggregation because is cheaper from a data move point of view due to the reduction of the data set performed by the local group by.

- **DSQL step 2**. The global aggregation sum(*l_quantity*) takes place and we perform the semi-join between *partsupp* and *part*. Finally, we perform a shuffle of the result set on *ps_suppkey* to perform a distributed group by to execute the final semi-join between *supplier* and *partsupp*; that is the result of transforming SQ1 into a join. Again, a local group by strategy is chosen to reduce the amount of data being communicated.
- **DSQL step 3**. Is the last step of the parallel plan in which results are communicated to the user.

5. CONCLUSION

Through the use of technology developed for SQL Server, the PDW QO goes beyond simple predicate pushing and join reordering, and incorporates a number of advanced query optimization techniques. These techniques include things such as contradiction detection, redundant join elimination, subquery unnesting, and outerjoin reordering. The cost model of PDW QO is specially crafted to reflect the distributed environment, and its use on a rich space of alternatives produces much higher-quality plans than simply parallelizing the best serial plan.

Another important aspect of technology reuse was that it shortened the time to build a cost-based optimizer for PDW. Successful delivery required designing the right abstractions and interfaces. The quality and effectiveness of the result validate the approach, which quickly incorporated state of the art commercial query optimization in the PDW product.

6. REFERENCES

[1] Aster Data. http://www.asterdata.com/.

[2] Greenplum. http://www.greenplum.com/.

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[4] M. Elhemali and L. Giakoumakis. Unit-testing query transformation rules. In *DBTest*, pages 1–6, 2008.

[5] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[6] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[7] J. R. Haritsa. The picasso database query optimizer visualizer. *PVLDB*, 3(2):1517–1520, 2010.

[8] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. Technical Report UCB/ERL M91/50, EECS Department, University of California, Berkeley, 1991.

[9] Microsoft Corporation. Microsoft SQL Server PDW. `http://www.microsoft.com/sqlserver/en/us/solutions-technologies/data-warehousing/pdw.aspx`.

[10] R. V. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, 2011.

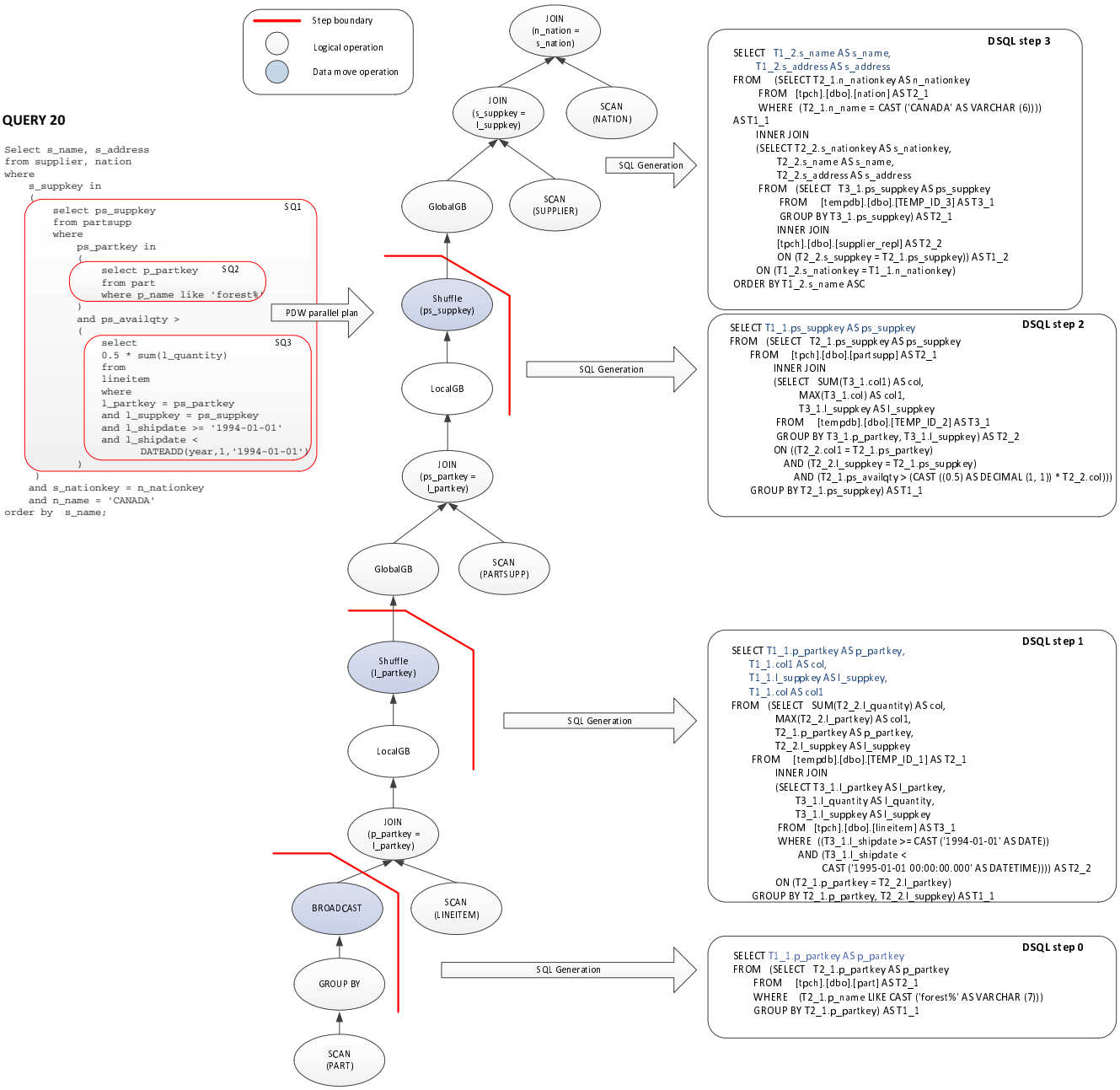

Figure 7: Parallel query plan for TPCH query 20.