Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer

Jingren Zhou, Per-Ake Larson, Ronnie Chaiken

Microsoft, One Microsoft Way, Redmond, WA 98052, USA
{jrzhou, palarson, rchaiken}@microsoft.com

Abstract— **Massive data analysis on large clusters presents new opportunities and challenges for query optimization. Data partitioning is crucial to performance in this environment. However, data repartitioning is a very expensive operation so minimizing the number of such operations can yield very significant performance improvements. A query optimizer for this environment must therefore be able to reason about data partitioning including its interaction with sorting and grouping.**

SCOPE is a SQL-like scripting language used at Microsoft for massive data analysis. A transformation-based optimizer is responsible for converting scripts into efficient execution plans for the Cosmos distributed computing platform. In this paper, we describe how reasoning about data partitioning is incorporated into the SCOPE optimizer. We show how relational operators affect partitioning, sorting and grouping properties and describe how the optimizer reasons about and exploits such properties to avoid unnecessary operations. In most optimizers, consideration of parallel plans is an afterthought done in a postprocessing step. Reasoning about partitioning enables the SCOPE optimizer to fully integrate consideration of parallel, serial and mixed plans into the cost-based optimization. The benefits are illustrated by showing the variety of plans enabled by our approach.

I. Introduction

Internet companies have an increasing need to store and analyze massive data sets, such as search logs, web content, and click streams collected from a variety of web services. Data analysis may involve tens or hundreds of terabytes of data. To be able to perform such massive analysis in a cost-effective manner, several companies have developed distributed data storage and processing platforms on large clusters of shared-nothing commodity servers. Notable examples include Google's File System [6], Bigtable [3], MapReduce [5], Hadoop [1], Microsoft's Cosmos [2], and Dryad [10]. A typical cluster consists of hundreds or thousands of commodity machines connected via a high-bandwidth network.

SCOPE is a SQL-like scripting language used at Microsoft for massive data analysis [2]. SCOPE simplifies data analysis on large clusters by hiding hardware and implementation details, thus allowing users to focus on solving the problem at hand. Hundreds of SCOPE jobs run daily in our data centers. SCOPE uses a transformation-based optimizer, based on the Cascades framework [8], to generate efficient query plans that make use of all cluster resources.

In this environment it is crucial to generate parallel plans. Query optimizers in database systems typically start with an optimal serial plan and then add parallelism in a postprocessing step. This approach may result in sub-optimal plans. The challenge is to seamlessly integrate consideration of parallel plans into the normal optimization process. In this paper we describe how this was accomplished in the SCOPE optimizer.

Parallelism is achieved by partitioning data into subsets that can be processed independently. This may require complete repartitioning which is expensive because it involves transporting all data across the shared network. Reducing the number of partitioning operations is an important optimization goal. However, data partitioning cannot be considered in isolation because it often interacts with other data properties, in particular, with sorting and grouping properties.

We illustrate the importance of property reasoning by an example query. Assume that tables R and S are tens of terabytes and are randomly partitioned and distributed over a cluster of, say, 500 machines.

```
select R.c, S.d, count(*)
from R, S
where R.a = S.a and R.b = S.b and p1(R) and p2(S)
group by R.c, S.d
```

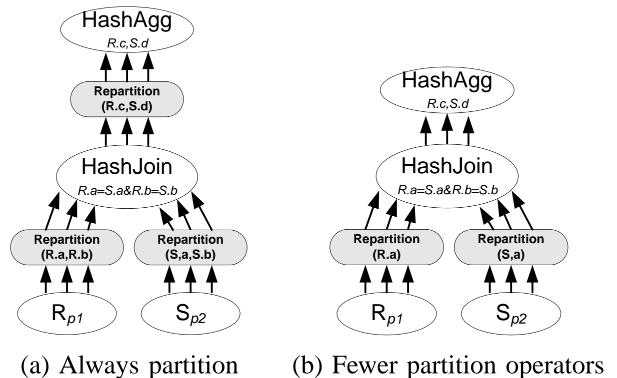


Fig. 1. Two Distributed Execution Plans

A straightforward execution plan is shown in Figure 1(a). A group of three arrows indicates a partitioned data flow with different partitions processed in parallel. Table R is first filtered and then hash partitioned on columns $\{R.a, R.b\}$. This step is done in parallel, with each machine processing its portion of the table. Similarly, table S is filtered and partitioned on $\{S.a, S.b\}$. Next, rows from matching R and S partitions are joined (in parallel), producing a result partitioned on $\{R.a, R.b\}$ (or $\{S.a, S.b\}$). In preparation for the subsequent group-by, the join result is then hash partitioned on $\{R.c, R.d\}$ to ensure that all rows with the same values for $\{R.c, R.d\}$ are contained in the same partition. Finally, each

partition is aggregated independently in parallel, producing the final result.

The default plan (a) shown in Figure 1 is reasonable but it may be expensive because of the two partitioning operations. Both R and S contain tens of terabytes of data so the data reshuffling through the network can pose a serious performance bottleneck. Plan (b) shows another plan where the repartitioning before aggregation has been eliminated. Joins and aggregations do not require the inputs to be partitioned on the *entire* set of join or grouping columns: any subset suffices. It is thus valid to partition R and S on $R.a$ and $S.a$, respectively, and evaluate the join, producing a result that is partitioned on $R.a$ (or $S.a$). If a functional dependency $R.c \rightarrow R.a$ exists (for instance, $R.c$ is the primary key of R), the join result is also partitioned on $R.c$ so that we can safely perform aggregation without repartitioning the data. Depending on data size and join selectivity, plan (b) can be significantly cheaper than plan (a). The optimizer should consider both plans in a cost-based fashion.

Earlier work has considered grouping and sorting but ours is the first paper to also take into account partitioning. The main contributions of our work are as follows.

- We design optimization rules and techniques to seamlessly generate and optimize both serial and parallel plans in a transformation-based optimizer. Our techniques are implemented in the SCOPE optimizer.
- We combine reasoning about partitioning, grouping, sorting properties into a single *uniform* framework.
- We present formal semantics for partitioning, grouping, and sorting properties and introduce a set of inference rules that also exploit functional dependencies and data constraints.

The rest of the paper is organized follows. We first describe different types of data exchange operators in Section II. These operators are key to generating parallel execution plans. In Section III we describe property reasoning procedures and their role inside the SCOPE optimizer. In Section IV, we consider partitioning, grouping, and sorting properties in a uniform framework and present formal semantics. Next, we describe the property reasoning required in three contexts: how to derive the properties of the results of various operators in Section V, how to determine required properties for a physical operator in Section VI, and how to match different properties in Section VII. In Section VIII, we describe a new set of rules to allow the optimizer to consider parallel query execution plans natively. We describe additional system details and illustrate the performance gains on a few example queries in Section IX. We survey related work in Section X and conclude in Section XI.

II. Parallel Plans and Exchange Operators

Distributed query processing is based on partitioning data into smaller subsets and processing partitions in parallel on multiple machines. This requires operators for splitting a single input into smaller partitions, merging multiple partitions into a single output, and repartitioning an already partitioned input into a new set of partitions. This can done by a single logical operator, the *data exchange* operator, that repartitions data from n inputs to m outputs [7]. $n = 1$ corresponds to initial partitioning of a single input and $m = 1$ corresponds to merging n inputs into a single output. After an exchange operator, the data is partitioned into m subsets that are then processed independently and in parallel using standard relational operators, until the data flows into the next exchange operator. Parallelism can be added easily in this way without modification to other relational operators.

At the implementation level, exchange consists of one or two physical operators: a partition operator and/or a merge operator. Suppose we want to repartition n input partitions, each one on a different machine, into m output partitions on a different set of machines. The processing is done by n partition operators, one on each input machine, and m merge operators, one on each output machine. A partition operator reads its input and splits it onto m subpartitions. Each merge operator collects the data for its partition from the n corresponding subpartitions.

To integrate consideration of parallel plans into the optimizer it must be able to reason about physical properties of data streams flowing between physical operators; how the data is partitioned, sorted and/or grouped. This includes understanding what requirements different operators have on their inputs, what effect they have on the physical properties of their outputs, and deciding whether the physical properties of data delivered by an operator satisfy the requirements of a consuming operator. This paper describes how we "taught" the SCOPE optimizer these skills.

To set the stage, we first classify exchange operator according to the topology of their data flows and describe the different partition and merge operators that are considered in this paper.

A. Exchange Topology

Figure 2 shows five types of exchange operators considered in this paper.

- *Initial Partitioning*: Shown in Figure 2(a), this operator consumes a single input stream and outputs m output streams with the data partitioned among the m stream.
- *(Full) Repartitioning*: Shown in Figure 2(b), this operator consumes n input partitions and produces m output partitions, partitioned in a different way.
- *Full Merge*: Shown in Figure 2(c), this operator consumes n input streams and merges then into a single output stream.
- *Partial Partitioning*: Shown in Figure 2(d), this is a special case of repartitioning. It takes n input streams and produces kn output streams. The data from each input partition is further partitioned among k output streams.
- *Partial Merge*: Shown in Figure 2(e), this is the inverse of partial partition. A partial merge takes kn input streams, merges groups of k of them together, and produces n output streams.

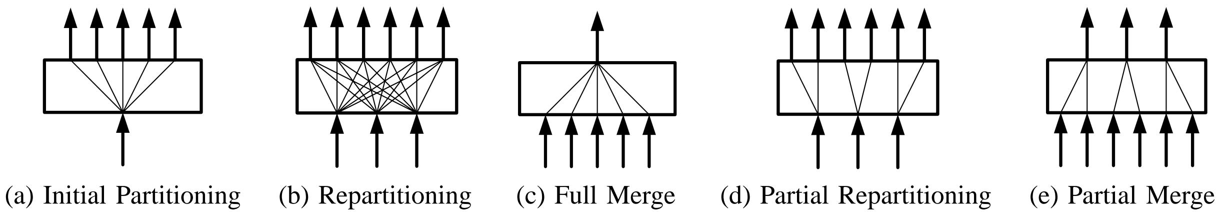


Fig. 2. Different Types of Data Exchange

Partial partitioning and partial merge operators are not discussed further in this paper as they are relatively rare. The same effect can be accomplished by full repartitioning and full merge, though not as efficiently.

B. Partitioning Schemes

Conceptually, an instance of a partition operator takes one input stream and generates multiple output streams. It consumes one row at a time and writes the row to the output stream selected by a partitioning function applied to the row. In this paper, we assume all partition operators are FIFO (first-in, first-out), that is, the order of two rows r_1 and r_2 in the input stream is preserved in the output stream if they are assigned to the same partition. There are several different types of partitioning schemes.

- *Hash Partitioning* applies a hash function to the partitioning columns to generate the partition number to which the row is output. This partitioning scheme is results in non-ordered partitions.
- *Range Partitioning* divides the domain of the partitioning columns into a set of disjoint ranges, as many as the desired number of partitions. A row is assigned to the partition determined by the value of its partitioning columns. This partitioning scheme produces ordered partitions.
- *Non-deterministic Partitioning* is any scheme where the data content of a row does not affect which partition the row is assigned to. Round-robin partitioning and random partitioning are of this type.
- *Broadcasting* takes one row at a time and outputs a copy of the row to every partition, that is, every output stream is a copy of the input stream. Broadcasting is typically used only on small, non-partitioned inputs.

The partition function to use and the number of output partitions are other aspects of exchange operators. It is up to the optimizer to make an optimal choice, based on data cardinalities and appropriate models for CPU, disk, and network costs. The detailed discussions are beyond the scope of this paper – our focus is on how to reason about partitioning, regardless of how many partitions are generated.

C. Merging Schemes

A merge operator combines data from multiple input streams into a single output stream. Depending on whether the input streams are sorted individually and how rows from different input streams are ordered, we have several types of merge operations.

- *Random Merge* randomly pulls rows from different input streams and merges them into a single output stream. While the ordering of rows from the same input stream is preserved, the ordering of rows from different inputs is indeterminate.
- *Sort Merge* takes a list of sort columns as a parameter and a set of input streams sorted on the same columns. The input streams are merged together into a single sorted output stream.
- *Concat Merge* concatenates multiple input streams into a single output stream. It consumes one input stream at a time and outputs its rows *in order* to the output stream. That is, it maintains the row order within an input stream but it does not guarantee the order in which the input streams are consumed.
- *Sort-Concat Merge* is a sorted version of concat merge. It takes a list of sort columns as a parameter. First, it picks one row (usually the first one) from each input stream, sorts them on the values on the sort columns, and uses the row order to decide the order in which to concatenate the input streams. This is useful for merging range-partitioned inputs into a totally ordered output.

III. Property Reasoning Inside the Optimizer

Conceptually, an optimizer generates all possible rewritings of a query expression and chooses the one with the lowest estimated cost. Query expressions are represented as operator trees. Operators are of two types: logical and physical. A logical operator specifies what operation to perform but not the algorithm while a physical operator also specifies the algorithm. For example, join is a logical operator while hash join, merge join and nested-loop join are physical operators.

Transformation-based optimization can be viewed as divided into two phases, namely, logical exploration and physical optimization. Logical exploration applies transformation rules that generate new logical expressions. During physical optimization, implementation rules are applied that convert logical operators to physical operators.

Algorithm 1 shows a (simplified) recursive optimization routine that takes as input a query expression and a set of requirements. We highlight three different contexts where reasoning about data properties occur during query optimization.

- *Determining child required properties.* The parent (physical) operator imposes requirements that the output from the current physical operator must satisfy, for example, the data must be sorted on $R.b$. To function correctly, the operator may itself impose certain requirements on its inputs, for example, the two inputs to a join must be partitioned on $R.a$ and $S.a$, respectively. Based on these

Algorithm 1: OptimizeExpr($expr$, $reqd$)

```
Input: Expression expr,    ReqdProperties reqd
Output: QueryPlan plan
/*Enumerate all the possible logical rewrites  */
LogicalTranform(expr);
foreach logical expression lexpr do
    /*Try out implementations for its root
      operator                                   */
    PhysicalTranform(lexpr);
    foreach expression pexpr that has physical
    implementation for its root operator do
        ReqdProperties reqdChild =
        DetermineChildReqdProperties(pexpr, reqd);
        /*Optimize child expressions             */
        QueryPlan planChild =
        OptimizeExpr(pexpr.Child, reqdChild);
        DlvdProperties dlvd =
        DeriveDlvdProperties(planChild);
        if PropertyMatch(dlvd, reqd) then
            EnqueueToValidPlans();
        end
    end
end
plan = CheapestQueryPlan();
return plan;
```

two requirements, we must then determine what requirements to impose on the result of the input expressions. The function DetermineChild ReqdProperties is used for this purpose. If the requirements are incompatible, a compensating operator such as a sort or partition may need to be added. This part is covered in Section VI.

- *Deriving delivered properties.* Once physical plans for the child expressions have been determined, we compute the data properties of the result of the current physical operator, by calling the function DeriveDlvd Properties. A child expression may not deliver exactly the requested properties. For example, we may have requested a result grouped on $R.a$ but the chosen plan delivers a result that is, in addition, sorted on $R.a$. The delivered properties are a function of the delivered properties of the inputs and the behavior of the current operator, for example, whether it is hash or merge join. We explain this process in Section V.
- *Property matching.* Once the delivered properties have been determined, we test whether they satisfy the required properties, by calling the function PropertyMatch. If they do not match, the plan with the current operator is discarded. The match does not have to be exact – a result with properties that exceed the requirements is acceptable. We cover the details in Section VII.

IV. Property Formalism

We begin in Section IV-A by briefly reviewing functional dependencies and a few other constraints. Partitioning, grouping, and sorting properties are formally defined in Section IV-B. We summarize a set of inference rules in Section IV-C.

A. FDs, Constraints and Equivalences

A set of columns $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ functionally determines a set of columns $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, if for any two rows that agree on the values of columns in $\mathcal{R}$, they also agree on the values of columns in $\mathcal{S}$. We denote ***functional dependency*** by $\mathcal{R} \rightarrow \mathcal{S}$. $\mathcal{R} \rightarrow \mathcal{S}$ is simply a shorthand notation for $\mathcal{R} \rightarrow S_1, \mathcal{R} \rightarrow S_2, \cdots \mathcal{R} \rightarrow S_m$.

Functional dependencies can arise in several ways.

Trivial FDs: $\mathcal{R} \rightarrow \mathcal{R}'$ whenever $\mathcal{R} \supseteq \mathcal{R}'$.

Key constraints: Keys are a special case of functional dependencies. If $\mathcal{X}$ is a key of relation T, then $\mathcal{X}$ functionally determines every column of T.

Column equality constraints: A selection or join with a predicate $R_i = S_k$ implies that the functional dependencies $\{R_i\} \rightarrow \{S_k\}$ and $\{S_k\} \rightarrow \{R_i\}$ hold in the result.

Constant constraints: After a selection with a predicate $R_i = constant$ all rows in the result have the same value for column R_i. This can be viewed as a functional dependency which we denote by $\emptyset \rightarrow R_i$.

Grouping columns: After a group-by with grouping columns $\mathcal{R}$, $\mathcal{R}$ is a key of the result and, thus, functionally determines all other columns in the result.

A set of columns that are known to have the same value in all tuples of a relation belong to a ***column equivalence class***. An equivalence class may also contain a constant c, which implies that all column in the class have the value c. Equivalence classes are generated by equality predicates, typically equijoin conditions and equality comparisons with a constant.

Functional dependencies and column equivalence classes can be computed bottom up in an expression tree. As this issue is well covered elsewhere [4], [16], [13], [12], we omit the details but assume that functional dependencies and equivalence classes have been computed.

B. Structural Properties

We now formally define three properties describing the structure or layout of a relation: partitioning, grouping and sorting, collectively referred to as *structural (data) properties*. A relation can be a source table or a result produced by a query expression. Table I summarizes some of the notation used in this paper.

A partition operation divides a relation into disjoint subsets, called partitions. A partition function defines which rows belong to which partitions. Partitioning applies to the whole relation; it is a *global structural property*. Grouping and sorting properties define how the data within each partition is organized and are thus partition-local properties, here refereed to as *local structural properties*.

We first specify what it means for a sequence of rows to be grouped or sorted and then formally define *local structural properties*.

Definition IV.1 (Grouping) A sequence of rows $r_1, r_2, \ldots, r_m$ is grouped on a set of columns $\mathcal{X} = \{C_1, C_2, \ldots, C_n\}$, if $\forall r_i, r_j, i < j, r_i[\mathcal{X}] = r_j[\mathcal{X}] \Rightarrow \forall k, i < k < j, r_k[\mathcal{X}] = r_i[\mathcal{X}]$. We denote grouping by $\mathcal{X}^g$.

Definition IV.2 (Sorting) A sequence of rows $r_1, r_2, \ldots, r_m$ is sorted on a column C in an ascending (or descending) order,

TABLE I

NOTATION USED IN THIS PAPER

$\mathbb{R}, \mathbb{S}, \ldots$	Relations
$C_1, C_2, \ldots$	Columns
$\mathcal{X}, \mathcal{Y}, \ldots$	Sets of columns
$\mathcal{X} \equiv \mathcal{Y}$	Column sets are equal taking into account column equivalence
$r_1, r_2, \ldots$	Tuples
$\mathrm{P}_1, \mathrm{P}_2, \ldots$	Partitions
$r[C], r[\mathcal{X}]$	Projection of r onto column C and columns $\mathcal{X}$, respectively
$*$	Any properties (including empty)

if $\forall r_i, r_j, i < j \Rightarrow r_i[C] \leq r_j[C]$ (or $r_i[C] \leq r_j[C]$). We denote this ordering by C^o where $o \in \{o_\uparrow, o_\downarrow\}$.

Note that grouping is performed on a *set* of columns and the column order within the set does not matter while sorting is performed on a *list* of columns and the column order matters.

Local structural properties can be represented by an ordered sequence of actions $\{\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_m\}$. Each action is either grouping on a set of columns $\mathcal{X}^g$, or sorting on a single column C^o. The definitions of grouping and sorting actions follow.

Definition IV.3 (Grouping Action) A sequence of rows $r_1, r_2, \ldots, r_n$ satisfies the local structural properties

$$\{\hat{A}_1, \ldots, \hat{A}_{m-1}, \mathcal{X}^g\}$$

if it satisfies the properties $\{\hat{A}_1, \ldots, \hat{A}_{m-1}\}$ and, in addition,

$$\forall r_i, r_j, i < j,\ r_i[\hat{A}_1, \ldots, \hat{A}_{m-1}, \mathcal{X}] = r_j[\hat{A}_1, \ldots, \hat{A}_{m-1}, \mathcal{X}] \Rightarrow \forall k, i < k < j,\ r_k[\hat{A}_1, \ldots, \hat{A}_{m-1}, \mathcal{X}] = r_i[\hat{A}_1, \ldots, \hat{A}_{m-1}, \mathcal{X}].$$

Definition IV.4 (Sorting Action) A sequence of rows $r_1, r_2, \ldots, r_n$ satisfies the local structural properties

$$\hat{A}_1, \ldots, \hat{A}_{m-1}, C^o$$

where $o \in \{o_\uparrow, o_\downarrow\}$, if it satisfies the properties $\{\hat{A}_1, \ldots, \hat{A}_{m-1}\}$ and, in addition,

$$\forall r_i, r_j, i < j,\ r_i[\hat{A}_1, \ldots, \hat{A}_{m-1}] = r_j[\hat{A}_1, \ldots, \hat{A}_{m-1}] \Rightarrow r_i[C]\ op\ r_j[C].$$

where $op = '\leq'$ when $o = o_\uparrow$ and $op = '\geq'$ when $o = o_\downarrow$.

Wang et, al. [17] also used an ordered sequence of annotated columns to represent secondary ordering or grouping. But their approach is more limited because each step in their sequence can only be either grouping or sorting on a single column. Note that $\{C_1^g, C_2^g\} \Rightarrow \{\{C_1, C_2\}^g\}$ but $\{\{C_1, C_2\}^g\} \not\Rightarrow \{C_1^g, C_2^g\}$.

There are two major classes of partitioning schemes, *ordered* and *non-ordered*. A non-ordered partitioning scheme ensures only that all rows with the same values of the partitioning columns are contained in the same partition. This is analogous to *grouping* as local property. An ordered partitioning scheme provides the additional guarantee that the partitions cover disjoint ranges of the partitioning columns. In other words, rows assigned to a partition P_i are either all less than or greater than rows in another partition P_j. This is analogous to *ordering* as a local property.

The following definitions formally state what properties are guaranteed by different partitioning schemes.

Definition IV.5 (Non-ordered Partitioning) A relation $\mathcal{R}$ is *non-ordered partitioned* on columns $\mathcal{X}$, if it satisfies the condition

$$\forall r_1, r_2 \in \mathcal{R}:\ r_1[\mathcal{X}] = r_2[\mathcal{X}] \Rightarrow \mathrm{P}(r_1) = \mathrm{P}(r_2)$$

where P denotes the partitioning function used.

Definition IV.6 (Ordered Partitioning) A relation $\mathcal{R}$ is *ordered-partitioned* into partitions $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_m$ on columns $\{C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n}\}$ where $o_i \in \{o_\uparrow, o_\downarrow\}$, if it satisfies the condition in the previous definition and the additional condition

$$\forall \mathrm{P}_i, \mathrm{P}_j, i \neq j : (\forall r_1 \in \mathrm{P}_i, r_2 \in \mathrm{P}_j : r_1 <_{C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n}} r_2)\ or\ (\forall r_1 \in \mathrm{P}_i, r_2 \in \mathrm{P}_j : r_1 >_{C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n}} r_2).$$

An ordered partitioning can be achieved by range partitioning while several methods produce a non-ordered partitioning. We discuss different partitioning methods further in Section II.

Definition IV.7 (Structural Properties) The structural properties of a relation $\mathcal{R}$ can be represented by partitioning information and an ordered sequences of actions,

$$\{\mathcal{P}^\theta; \{\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n\}\}$$

where $\theta \in \{o, g\}$, meaning ordered and non-ordered partitioning. The first part defines its global structural property while the second sequence defines its local structural property.

Empty structural properties are allowed. $\{\perp; *\}$ indicates that data is not partitioned while $\{\emptyset; *\}$ indicates that data is randomly partitioned. Another special case is $\{\top; *\}$, which indicates that data is completely duplicated and each partition contains a complete copy of the data. For our purpose, the exact partitioning function or the number of partitions are not needed so they are not specified in Definition IV.7.

TABLE II

A EXAMPLE RELATION WITH PARTITIONING, GROUPING, AND SORTING

Partition 1	Partition 2	Partition 3
{1,4,2}, {1,4,5}, {7,1,2}	{4,1,5}, {3,7,8}, {3,7,9}	{6,2,1}, {6,2,9}

Table II shows an instance of a relation with three columns C_1, C_2, C_3 and structural properties $\{\{C_1\}^g; \{\{C_1, C_2\}^g, C_3^o\}\}$. In words, the relation is partitioned on column C_1 and, within each partition, data is first grouped on columns C_1, C_2, and, within each such group, sorted by column C_3.

C. Inference Rules

We now briefly discuss inference rules for structural properties. The first rule shows that local properties can be truncated.

$$\{*; \{\hat{A}_1, \ldots, \hat{A}_{m-1}, \hat{A}_m\}\} \Rightarrow \{*; \{\hat{A}_1, \ldots, \hat{A}_{m-1}\}\} \quad (1)$$

Global properties cannot be truncated but they can be expanded. A result that is partitioned on columns C_1, C_2 is not partitioned on C_1 because two rows with the same value for C_1 may be in different partitions. However, a result partitioned on C_1 alone is in fact partitioned on C_1, C_2 because two rows

that agree on C_1, C_2 also agree on C_1 alone and, consequently, they are in the same partition. This observation gives the following two rules.

$$\{\{C_1, C_2, \ldots, C_m\}^g; *\} \Rightarrow \{\{C_1, C_2, \ldots, C_m, C_{m+1}\}^g; *\} \quad (2)$$
$$\{\{C_1^o, C_2^o, \ldots, C_m^o\}; *\} \Rightarrow \{\{C_1^o, C_2^o, \ldots, C_m^o, C_{m+1}^o\}; *\} \quad (3)$$

If a sequence of rows is sorted, it is also grouped. This yields two rules.

$$\{*; \{\hat{A}_1, \ldots, C^o, \ldots, \hat{A}_m\}\} \Rightarrow \{*; \{\hat{A}_1, \ldots, C^g, \ldots, \hat{A}_m\}\} \quad (4)$$
$$\{\{C_1^o, C_2^o, \ldots, C_n^o\}; *\} \Rightarrow \{\{C_1, C_2, \ldots, C_n\}^g; *\} \quad (5)$$

Functional dependencies allow us to eliminate grouping and sorting columns. The following two simplification rules can be applied to global properties and to individual actions in local properties.

$$\exists\ C \in \mathcal{X} : (\mathcal{X} - \{C\}) \rightarrow C, \mathcal{X}^g \Rightarrow (\mathcal{X} - \{C\})^g \quad (6)$$
$$\exists\ \{C_1, \ldots, C_{j-1}\} \rightarrow C_j : \{C_1^o, \ldots, C_{j-1}^o, C_j^o, C_{j+1}^o, \ldots\} \Rightarrow \{C_1^o, \ldots, C_{j-1}^o, C_{j+1}^o, \ldots\} \quad (7)$$

However, the following rule applies only to local structural properties, where $\mathfrak{Columns}$ returns a set of columns defined in actions.

$$\exists\ \mathfrak{Columns}[\hat{A}_1, \ldots, \hat{A}_{i-1}] \rightarrow \mathfrak{Columns}[\hat{A}_i] : \{*; \{\hat{A}_1, \ldots, \hat{A}_{i-1}, \hat{A}_i, \hat{A}_{i+1}, \ldots\}\} \Rightarrow \{*; \{\hat{A}_1, \ldots, \hat{A}_{i-1}, \hat{A}_{i+1}, \ldots\}\} \quad (8)$$

V. Deriving Structural Properties

We now consider how to derive the structural properties of the output of a *physical* operator. Earlier research has shown how to derive ordering and grouping properties for standard relational operators executed on non-partitioned inputs [15], [16], [17], [13], [12]. Ordering and grouping are local properties, that is, properties of each partition, so previous work still applies when the operators are running in partitioned mode. Standard relational operators have no effect on partitioning. What remains is to derive global and local properties after a physical partition, merge or repartition operator.

A. Properties After a Partitioning Operator

Partition operators are assumed to be FIFO, that is, they output rows in the same order that they are read from the input. Thus, they affect the global properties but not local properties. Every output partition inherits the local properties (sorting, grouping) of its input. Table III summarizes the properties of the output after a partition operator when the input has properties $\{\mathcal{X}; \mathcal{Y}\}$.

Hash partitioning on columns $C_1, C_2, \ldots, C_n$ produces a non-ordered collection of partitions. This is similar to grouping so we indicate hash partitioning by $\{C_1, C_2, \ldots, C_n\}^g$. The order of partitioning columns does not matter, that is, $\{\ldots, C_i, \ldots, C_j, \ldots\}^g \Leftrightarrow \{\ldots, C_j, \ldots, C_i, \ldots\}^g$.

Range partitioning on columns $C_1, C_2, \ldots, C_n$ produces an ordered collection of partitions. We denote range partitioning by $\{C_1^{o_1}, C_2^{o_2}, \ldots, C_m^{o_m}\}$, $o_i \in \{o_\uparrow, o_\downarrow\}$. We use the shorthand $\{C_1^o, C_2^o, \ldots, C_m^o\}$ or $\{C_1, C_2, \ldots, C_m\}^o$ if the sort order on individual columns is not important. For range partitioning, the order of partitioning columns does matter.

In a non-deterministic partitioning scheme (round-robin and random partitioning) which partition a row is assigned to is independent on its content. They use no partitioning columns so we indicate this form of partitioning by $\emptyset$.

TABLE III
Structural Properties of the Result After Partitioning an Input with Properties $\{\mathcal{X}; \mathcal{Y}\}$

Scheme	Result
Hash on $C_1, \ldots, C_n$	$\{\{C_1, \ldots, C_n\}^g; \mathcal{Y}\}$
Range on $C_1^o, \ldots, C_n^o$	$\{\{C_1^o, \ldots, C_n^o\}; \mathcal{Y}\}$
Non-Deterministic	$\{\emptyset; \mathcal{Y}\}$
Broadcast	$\{\top; \mathcal{Y}\}$

B. Properties After a Merge Operator

A merge operator produces a single output. Its local properties depend on the local properties of the input and the merge operator type: random merge, sort merge, concat merge and sort-concat merge.

Table IV summarizes the structural properties after a full merge, depending on the type of merge operator and whether the input partitioning is ordered or non-ordered.

A random merge does not guarantee any row order in the result, so no local properties can be derived for the output.

For a sort-merge, there are two cases. If the local properties of the input imply that the input streams are sorted on the columns used in the merge, the output will be sorted, otherwise not.

A concat merge operator maintains the row order within each source partition. If each source partition is *grouped* in a similar way to how it is *non-ordered partitioned*, the result of is also grouped, otherwise not.

Example 1 Given inputs with properties $\{\{C_1, C_2\}^g; \{C_2^o, C_1^o, C_3^o\}\}$, concat merging generates an output with properties $\{\{C_1, C_2\}^g; \{\{C_1, C_2\}^g, C_3^o\}\}$, if the merge is part of repartitioning operation, and $\{\bot; \{\{C_1, C_2\}^g, C_3^o\}\}$, if it implements a full merge.

A sort-concat merge produces a sorted result if inputs are range partitioned and each partition is also sorted on the same columns as it is partitioned on.

Example 2 A sort-concat full merge on $\{C_1^o, C_2^o\}$ of inputs with properties $\{\{C_1^o, C_2^o\}; \{C_1^o, C_2^o, C_3^g\}\}$ generates an output with properties $\{\bot; \{C_1^o, C_2^o, C_3^g\}\}$.

C. Properties After a Repartitioning Operator

The properties of the result after repartitioning depends on the partitioning scheme, the merge scheme and the local properties of the input. Table V summarizes the structural properties after repartitioning an input with properties $\{\mathcal{P}; \mathcal{Y}\}$ where $\mathcal{P}$ denotes any partition property.

TABLE IV

STRUCTURAL PROPERTIES OF THE RESULT AFTER A FULL MERGE

	Input Properties $\{\mathcal{X}^g;\mathcal{Y}\}$	Input Properties $\{\mathcal{X}^o;\mathcal{Y}\}$
Random merge	$\{\perp;\emptyset\}$	$\{\perp;\emptyset\}$
Sort merge on $\mathcal{S}^o$	1). $\{\perp;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\perp;\emptyset\}$ otherwise	1). $\{\perp;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\perp;\emptyset\}$ otherwise
Concat merge	1). $\{\perp;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\perp;\emptyset\}$ otherwise	1). $\{\perp;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\perp;\emptyset\}$ otherwise
Sort-concat merge on $\mathcal{S}^o$	1). $\{\perp;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\perp;\emptyset\}$ otherwise	1). $\{\perp;\mathcal{Y}\}$ if $\mathcal{S}^o\Leftrightarrow\mathcal{X}^o$ and $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\perp;\emptyset\}$ otherwise

TABLE V

STRUCTURAL PROPERTIES OF THE RESULT AFTER REPARTITIONING ON $\mathcal{X}$ AN INPUT WITH PROPERTIES $\{\mathcal{P};\mathcal{Y}\}$

	Hash partitioning	Range partitioning	Non-determ. partitioning
Random merge	$\{\mathcal{X}^g;\emptyset\}$	$\{\mathcal{X}^o;\emptyset\}$	$\{\emptyset;\emptyset\}$
Sort merge on $\mathcal{S}^o$	1). $\{\mathcal{X}^g;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\mathcal{X}^g;\emptyset\}$ otherwise	1). $\{\mathcal{X}^o;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\mathcal{X}^o;\emptyset\}$ otherwise	1). $\{\emptyset;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\emptyset;\emptyset\}$ otherwise
Concat merge	1). $\{\mathcal{X}^g;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\mathcal{X}^g;\emptyset\}$ otherwise	1). $\{\mathcal{X}^o;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\mathcal{X}^o;\emptyset\}$ otherwise	1). $\{\emptyset;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\emptyset;\emptyset\}$ otherwise
Sort-concat merge on $\mathcal{S}^o$	1). $\{\mathcal{X}^g;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\mathcal{X}^g;\emptyset\}$ otherwise	1). $\{\mathcal{X}^o;\mathcal{Y}\}$ if $\mathcal{S}^o\Leftrightarrow\mathcal{X}^o$ and $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\mathcal{X}^o;\emptyset\}$ otherwise	1). $\{\emptyset;\mathcal{Y}\}$ if $\mathcal{S}^o\Leftrightarrow\mathcal{X}^o$ and $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\emptyset;\emptyset\}$ otherwise

VI. DERIVING REQUIRED PROPERTIES

We now consider how to determine required properties of the inputs for different physical operators. Table VI lists required input properties for the most common physical operators. Depending on whether the operator is executed in either partitioned or non-partitioned mode, it imposes different requirements on its inputs.

Table scan, select and project process individual rows and impose no requirements on their inputs, that is, it doesn't matter how the input data is partitioned, sorted, or grouped. Thus their input requirements are shown as $\{\mathcal{X};*\}$ where $\mathcal{X}$ can be any set of columns.

In the partitioned mode, a sort of the complete input requires that the input be range partitioned on a prefix of the sort columns and each partition be sorted on the sort columns. Sorted output partitions are then obtained by a sort-merge.

For a hash aggregation to work correctly, all rows with the same value of the grouping columns must be in a single partition. This is guaranteed as long as the input is partitioned on a *subset* of the grouping columns. A stream aggregation also requires that the input be partitioned on a *subset* of the grouping columns. In addition, the rows within each partition must, as a minimum, be grouped on the grouping columns.

We consider two types of partitioned joins: pair-wise join and broadcast join. A pair-wise join takes two partitioned inputs. The inputs must be partitioned on a subset of the join columns and in the same way, that is, on the same set of equivalent columns into the same number of partitions. Broadcast join takes one partitioned input and one non-partitioned input that is sent (broadcast) to each partition of the other input. It doesn't matter how the partitioned input is partitioned. These are the only requirements for a nested-loop or hash join. A merge join has the additional requirement that each partition be sorted on the join columns.

Example 3 Suppose we are considering using a partitioned merge join to join tables $\mathbb{R}$ and $\mathbb{S}$ on $R.C_1 = S.C_1$ and $R.C_2 = S.C_2$. Based on the rules in Table VI, both inputs must be partitioned and sorted in the same way. The partitioning columns must be a subset of or equal to the join columns ($\{R.C_1, R.C_2\}$ and $\{S.C_1, S.C_2\}$, respectively). A prefix of the sort columns must also be equal to the join columns on each input.

Each of the following requirements satisfies the restrictions and is thus valid input requirements. We do not list all the possibilities here and also leave the exact sort order, $o_\uparrow$ and $o_\downarrow$, unspecified.

- $\{R.C_1^g;\{R.C_2^o, R.C_1^o\}\}$ and $\{S.C_1^g;\{S.C_2^o, S.C_1^o\}\}$
- $\{R.C_2^g;\{R.C_1^o, R.C_2^o\}\}$ and $\{S.C_2^g;\{S.C_1^o, S.C_2^o\}\}$
- $\{\{R.C_1, R.C_2\}^g;\{R.C_2^o, R.C_1^o\}\}$ and $\{\{S.C_1, S.C_2\}^g;\{S.C_2^o, S.C_1^o\}\}$

As shown by the example, the requirements in Table VI for the child expressions are not always unique and can be satisfied in several ways. For instance, aggregation on $\{C_1, C_2\}$ requires the input to be partitioned on $\{C_1\}$, $\{C_2\}$, or $\{C_1, C_2\}$. Conceptually, each requirement corresponds to one specific implementation. This situation could be handled by generating multiple alternatives, one for each requirement. However, this approach would generate a large number of alternatives, making optimization more expensive. Instead, we allow required properties to cover a range of possibilities and rely on enforcer rules, described in Section VIII, to generate valid rewrites.

To this end, our optimizer encodes required structural properties as follows.

- Partitioning requirement:
  - Non-partitioned ($\perp$), broadcast ($\top$) or partitioned mode
  - If in partitioned mode, *minimum* partitioning column set $\mathcal{P}_{min}$ and *maximum* partitioning column set $\mathcal{P}_{max}(\emptyset \subseteq \mathcal{P}_{min} \subseteq \mathcal{P}_{max})$
  - If $\mathcal{P}_{min} = \emptyset$, whether an explicit partitioning by some column is required so that $\mathcal{P} \neq \emptyset$.

TABLE VI

REQUIRED STRUCTURAL PROPERTIES OF INPUTS TO PHYSICAL OPERATORS

	Non-Partitioned Version	Partitioned Version
Table Scan	$\{\bot;*\}$	$\{\mathcal{X};*\},\mathcal{X}\neq\emptyset$
Select	$\{\bot;*\}$	$\{\mathcal{X};*\},\mathcal{X}\neq\emptyset$
Project	$\{\bot;*\}$	$\{\mathcal{X};*\},\mathcal{X}\neq\emptyset$
Sort on $\mathcal{S}^o(\mathcal{S}\neq\emptyset)$	$\{\bot;\{\mathcal{S}^o,*\}\}$	$\{\mathcal{X}^o;\mathcal{S}^o\},\mathcal{X}\neq\emptyset,\mathcal{S}^o\Rightarrow\mathcal{X}^o$
Hash Aggregate on $\mathcal{G}$	$\{\bot;*\}$	$\{\mathcal{X};*\},\emptyset\subset\mathcal{X}\subseteq\mathcal{G},\mathcal{G}\neq\emptyset$
Stream Aggregate on $\mathcal{G}$	$\{\bot;\{\mathcal{G}^g,*\}\}$	$\{\mathcal{X};\{\mathcal{G}^g,*\}\},\emptyset\subset\mathcal{X}\subseteq\mathcal{G},\mathcal{G}\neq\emptyset$
Nested-loop or Hash Join (equijoin on columns $\mathcal{J}_1\equiv\mathcal{J}_2$)	Both inputs $\{\bot;*\}$	*Pair-wise Join:* Input 1: $\{\mathcal{X};*\},\emptyset\subset\mathcal{X}\subseteq\mathcal{J}_1$; Input 2: $\{\mathcal{Y};*\},\emptyset\subset\mathcal{Y}\subseteq\mathcal{J}_2$; $\mathcal{X}\equiv\mathcal{Y}$ *Broadcast Join:* Input 1: $\{\top;*\}$; Input 2: $\{\mathcal{X};*\},\mathcal{X}\neq\emptyset$
Merge Join (equijoin on columns $\mathcal{J}_1\equiv\mathcal{J}_2$)	Input 1: $\{\bot;\mathcal{S}_1^o\}$ Input 2: $\{\bot;\mathcal{S}_2^o\}$	*Pair-wise Join:* Input 1: $\{\mathcal{X};\mathcal{S}_1^o\},\emptyset\subset\mathcal{X}\subseteq\mathcal{J}_1$; Input 2: $\{\mathcal{Y};\mathcal{S}_2^o\},\emptyset\subset\mathcal{Y}\subseteq\mathcal{J}_2$; $\mathcal{X}\equiv\mathcal{Y}$ *Broadcast Join:* Input 1: $\{\top;\mathcal{S}_1^o\}$; Input 2: $\{\mathcal{Y};\mathcal{S}_2^o\},\mathcal{Y}\neq\emptyset$
	$\mathcal{J}_1=prefix(\mathcal{S}_1^o)$, $\mathcal{J}_2=prefix(\mathcal{S}_2^o)$	

- Sorting requirement:
  a list of sorting columns $\{S_1^o,S_2^o,\ldots,S_n^o\},o\in\{o_\uparrow,o_\downarrow\}$
- Grouping requirement:
  a set of grouping columns $\{G_1,G_2,\ldots,G_n\}^g$

In the previous example of aggregation on $\{C_1,C_2\}$, the partitioning requirement would be $\mathcal{P}_{min}=\emptyset,\mathcal{P}_{max}=\{C_1,C_2\}$. This requirement is satisfied by hash or range partition with column set $\mathcal{P}$ where $\mathcal{P}_{min}\subset\mathcal{P}\subseteq\mathcal{P}_{max}$.

The rules in Table VI do not consider requirements imposed on the operator by its parent. For instance, if a merge join is required to produce a result sorted on $\{C_1,C_2\}$ but its equality join predicates are on $\{C_3\}$, there is no merge join implementation that could satisfy its sorting requirements, assuming that sorting on $\{C_3\}$ does not imply sorting on $\{C_1,C_2\}$. In this case, this merge join is an invalid alternative - it can never produce an output that satisfies the requirements. The optimizer checks for such invalid alternatives and discards them immediately.

VII. PROPERTY MATCHING

Property matching checks whether one set of properties $\mathfrak{P}_1$ satisfies another $\mathfrak{P}_2$, that is, whether $\mathfrak{P}_1\Rightarrow\mathfrak{P}_2$. The optimizer ensures that a physical (sub)plan is valid by checking that its delivered properties satisfy the required properties.

Matching of structural properties can be done by matching global and local properties separately.

$$\{\mathcal{P}^\theta;\{\hat{A}_1,\hat{A}_2,\ldots,\hat{A}_n\}\}\Rightarrow\{\mathcal{Q}^\beta;\{\hat{B}_1,\hat{B}_2,\ldots,\hat{B}_n\}\}$$
$$\Leftrightarrow$$
$$\mathcal{P}^\theta\Rightarrow\mathcal{Q}^\beta \texttt{and } \{\hat{A}_1,\hat{A}_2,\ldots,\hat{A}_n\}\Rightarrow\{\hat{B}_1,\hat{B}_2,\ldots,\hat{B}_n\}$$

However, two structural properties may be equivalent even if they appear different because of functional dependencies and column equivalences. We cannot simply compare their original forms but must first convert them to *normalized* form. The basic idea of the normalization procedure is as follows: a) in each partitioning, sorting, grouping property, and *functional dependency*, replace each column with the representative column in its equivalence class, then b) in each partitioning, sorting and grouping property, remove columns that are functionally determined by some other columns.

Global and local properties are matched separately. Normalization and matching of local properties (sorting and grouping) have been studied extensively in [13], [12]. Matching of global properties (partitioning) is based on inference rules (2), (3), and (5) in section V. Recall that required properties are not always unique but may encode multiple alternatives, which results in additional matching opportunities.

Example 4 We want to test whether the structural properties $\mathfrak{P}_1=\{\{C_7,C_1,C_3\}^g;\{C_6^{o_\uparrow},C_2^{o_\downarrow},C_5^{o_\uparrow}\}\}$ satisfy the structural properties $\mathfrak{P}_2=\{\{C_1,C_2,C_4\}^g;\{\{C_1,C_2\}^g\}\}$. We know that the data satisfies the FD $\{C_6,C_2\}\rightarrow\{C_3\}$. There are two column equivalence classes $\{\mathbf{C_1},C_6\}$ and $\{\mathbf{C_2},C_7\}$ with C_1 and C_2 as representative columns, respectively.

After replacing columns by representative columns, we have

$$\mathfrak{P}_1=\{\{C_2,C_1,C_3\}^g;\{C_1^{o_\uparrow},C_2^{o_\downarrow},C_5^{o_\uparrow}\}\}$$
$$\mathfrak{P}_2=\{\{C_1,C_2,C_4\}^g;\{\{C_1,C_2\}^g\}\}$$
$$\{C_1,C_2\}\rightarrow\{C_3\}.$$

Next we apply the functional dependency to eliminate C_3, which changes $\mathfrak{P}_1$ to

$$\mathfrak{P}_1=\{\{C_2,C_1\}^g;\{C_1^{o_\uparrow},C_2^{o_\downarrow},C_5^{o_\uparrow}\}\}.$$

while $\mathfrak{P}_2$ is unchanged.

We first consider global properties. We want to prove that $\{C_2,C_1\}^g\Rightarrow\{C_1,C_2,C_4\}^g$. According to the expansion rule for global properties (inference rule (2)), the implication holds and thus the global properties match.

For local properties, we need to show that $\{C_1^{o_\uparrow},C_2^{o_\downarrow},C_5^{o_\uparrow}\}\Rightarrow\{\{C_1,C_2\}^g\}$. Applying the truncation rule for local properties (inference rule (1)), we obtain $\{C_1^{o_\uparrow},C_2^{o_\downarrow}\}\Rightarrow\{\{C_1,C_2\}^g\}$ because sorting implies grouping (inference rule (4)).

Since both global and local properties are matches, we deduce that $\mathfrak{P}_1$ satisfy $\mathfrak{P}_2$.

VIII. ENFORCER RULES

In this section we describe optimization rules that automatically introduce data exchange operators and thus seamlessly generate and optimize distributed query plans. We enhance the optimization framework in two ways.

- For each logical operator, we consider *both* non-partitioned and partitioned implementations, as long as they can ever satisfy their requirements.
- We rely on a series of enforcer rules (explained below) to modify requirements for structural properties, say, from non-partitioned to partitioned, or from sorted to non-sorted, etc.

Together with other optimization rules and property inferences, this enables the optimizer to consider both serial and parallel in a *single integrated* framework. It greatly enhances the power of a traditional query optimizer without dramatic infrastructure changes.

We begin with a simple example of sort optimization. Suppose that, during optimization, there is a request to optimize an expression with a specific sort requirement $\mathcal{S}$. The optimizer then considers different alternative physical operators for the root operator of the expression tree, derives what properties their inputs must satisfy, and requests an optimal plan for each input. There are typically three possible ways of ensuring that the result will be sorted. It is up to the optimizer to decide which plan is the best based on its cost estimates.

- If a physical operator itself can generate a sorted output, try this operator and push requirements imposed by the operator itself to its child expressions.
- If a physical operator retains the input order, try this operator and push the sort requirement plus requirements imposed by the operator itself to its child expressions.
- Otherwise, try the operator but add an explicit sort operator matching the requirement and then optimize the child expressions *without* the sort requirement.

In the last case, the optimizer *enforces* a sort requirement on top of the physical operator. We call such optimization rules *enforcer* rules. Grouping requirements can be handled in a similar way, expect we may not have an explicit group-only operator. A grouped result is usually produced as a side-effect of another operator, for example, a one-to-many nested-loop join.

A data exchange operator is similar to sorting. Its only effect is to change structural properties; it does not add or eliminate rows, nor does it modify individual rows in any way. Therefore, we model data exchange operators as *enforcers* of structural properties.

Algorithm 2 shows simplified pseudo-code for enforcing partitioning. For simplicity, handling of sorting and grouping requirements is not shown. When a sorting requirement exists, we consider both sort-merge exchange and regular exchange operations. We also ignore the details of the partitioning requirement.

Although the pseudo-code is much simplified, it captures the core ideas of enforcing partitioning requirements. For any expression with particular partitioning requirements, the optimizer 1) uses an operator that itself satisfies the requirements; 2) uses a partition-preserving operator and pushes the requirements to its children; 3) adds data exchange operators that allow the requirements for its child expressions to be relaxed or modified. The optimizer tries all the alternatives and selects the best plan based on estimated costs.

Algorithm 2: EnforceDataExchange($expr$, $reqd$)

```
Input: Expression expr,    ReqdProperties reqd
ReqdProperties reqdNew;
if Serial(reqd) then
    /*Require a serial output                          */
    AddExchange(FullMerge);
    reqdNew = GenParallel(reqd);
    Optimize(expr, reqdNew);
else
    /*Require a parallel output                        */
    /*Enumerate all possible partitioning
      properties that P_min ⊆ P ⊆ P_max                */
    foreach valid partition schema P do
        /*Case 1: repartition                          */
        AddExchange(Repartition);
        /*Generate new partitioning requirements
          for its children; remove specific
          partitioning columns                         */
        reqdNew = GenParallel(reqd);
        Optimize(expr, reqdNew);
        /*Case 2: initial partition                    */
        AddExchange(InitialPartition);
        /*Force the child to generate a serial
          output                                       */
        reqdNew = GenSerial(reqd);
        Optimize(expr, reqdNew);
    end
end
return;
```

Enforcer rules can seamlessly enable a single instance of an operator to work on a partitioned input by introducing a full merge operator. It can enable multiple instances of an operator to work on a non-partitioned input by introducing an initial partition operator. It can also enable multiple instances of an operator with specific partitioning requirements to work on any partitioned input sets by introducing a repartition operator.

Example 5 Assume we optimize a filter operator with a requirement that its results must be partitioned on $\{C_1, C_2\}$. Since the filter operator itself cannot generate a partitioned output, the optimizer consider at least the following three alternatives.

1) Keep the filter operator and propagate the partition requirement to its child expression.
2) Add a repartition operator on $\{C_1, C_2\}$ before the filter operator. The child expression of the repartition operator can be partitioned in any way.
3) Add an initial partitioning operator on $\{C_1, C_2\}$ before the filter operator. The child expression of the repartition operator has to produce a non-partitioned output.

If the child expression turns out to produce its results already partitioned on $\{C_1, C_2\}$, possibly because the inputs are partitioned in the same way or there is an explicit partition before, the first choice is likely to be the cheapest. In such a plan, no additional enforcer (partitioning) is needed.

The number of alternatives generated by enforcer rules could be large so usually the optimizer applies cost-based heuristics to prioritize alternatives and prune out less promising ones.

IX. THE OPTIMIZER IN ACTION

The execution platform for SCOPE is called Cosmos. Cosmos is a distributed computing platform for storing and analyzing massive data sets [2]. Cosmos is designed to run on large clusters consisting of thousands of commodity servers. It has two major subsystems.

Cosmos Storage. A distributed storage subsystem designed to reliably and efficiently store extremely large files. A file is divided into extents that are distributed and replicated for fault tolerance and compressed for better storage efficiency and I/O throughput.

Cosmos Execution Environment. A Cosmos computation job is modelled as a dataflow graph: a directed acyclic graph (DAG) with vertices representing processes and edges representing data flows. A job manager handles execution of a job. It schedules a DAG vertex onto the system processing nodes when all the inputs are ready, monitors vertex progress, and re-executes part of the DAG in case of failures [10].

In Cosmos, intermediate results are materialized on disk. This reduces the amount of re-computation required on failure and simplifies scheduling - any task whose inputs are available is runnable and can be scheduled independently of other tasks [5], [10], [2].

A. Sample Query

A typical SCOPE script involves one or more joins and multiple levels of aggregation and scans tens of (or hundreds of) terabytes of data. Here we use a relatively simple script but with a pattern that is very common in user scripts.

Among other things, SCOPE is used to analyze the load and performance of Cosmos production clusters. The system records a variety of system information as runtime events. The system records "ProcessStarted" events for every vertex, which includes the machine name on which the vertex runs, its job guid and process (vertex) guid, user name, job priority, and the time stamp when the vertex starts, etc. When a vertex finishes, the system also records "ProcessEnded" events, which includes its job guid and process guid, exit code, user group, and the time stamp when it ends, etc. Raw event files may contain duplicate events that need to be eliminated during analysis.

The script below calculates how much machine time has been spent on jobs issued by different user groups during the last month. This is a rough estimate of the system resources used by each user group.

```
extractStart =
  EXTRACT CurrentTimeStamp, ProcessGUID
  FROM "[...]/ProcessStartedEvents?Date=(Today-30)..Today"
  USING EventExtractor("ProcessStarted");

startData =
  SELECT DISTINCT CurrentTimeStamp AS StartTime, ProcessGUID
  FROM extractStart;

extractEnd =
  EXTRACT CurrentTimeStamp, UserGroupName, ProcessGUID
  FROM "[...]/ProcessEndedEvents?Date=(Today-30)..Today"
  USING EventExtractor("ProcessEnded");

endData =
  SELECT DISTINCT CurrentTimeStamp AS EndTime, UserGroupName,
    ProcessGUID
  FROM extractEnd;

perUserGroup =
  SELECT SUM((EndTime-StartTime).TotalMilliseconds)/3600000
    AS TotalCPUHours, UserGroupName
  FROM startData JOIN endData
    ON startData.ProcessGUID == endData.ProcessGUID
  GROUP BY UserGroupName
  ORDER BY UserGroupName;

OUTPUT perUserGroup TO "/my/CPUHoursPerUserGroup.txt";
```

The script first selects events from the last month and extracts time stamp information when each vertex (process) starts and ends, respectively, plus process guid and user group information. Next, duplicates in the raw events are removed by applying a DISTINCT aggregate on *all* columns. Finally, the cleaned data are joined on the process guid and the total CPU time per user group is calculated.

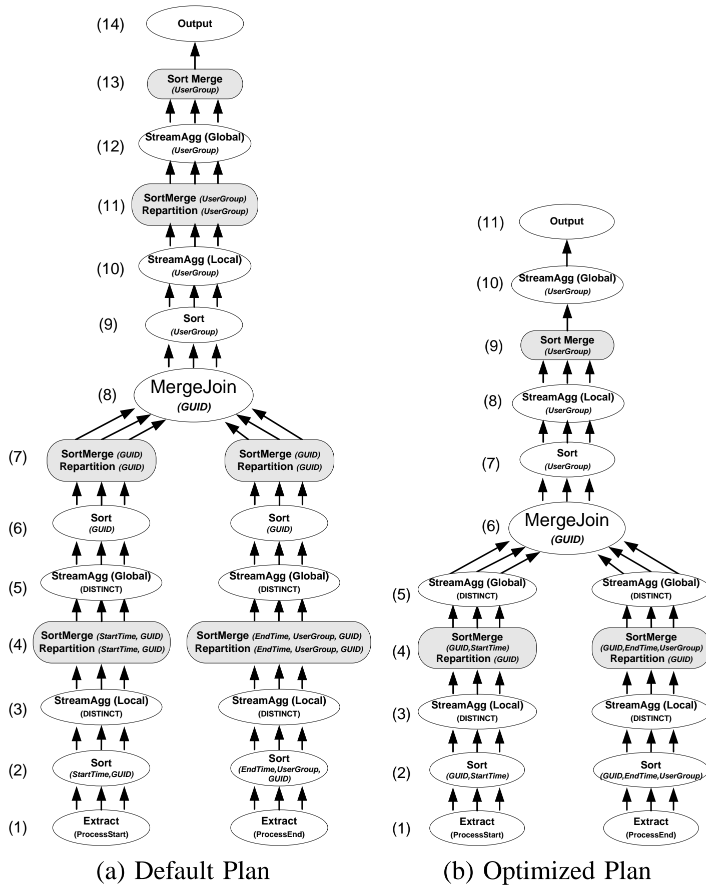
The "ProcessStarted" event in the queried period contains about 173 GB of data while the "ProcessEnded" event contains about 264 GB of data. Both Cosmos files are randomly distributed and replicated across the cluster.

We ran the query against on a small test cluster of 84 machines. Each machine has two dual-core Xeon processors running at 2GHz, 8 GB of DRAM, and four 500GB SATA disks. All machines run Windows Server 2003 Enterprise X64 Edition SP1.

B. Query Plans

Figure 3 compares query plans with and without optimization. We begin by explaining the default SCOPE plan, shown in Figure 3(a). A partitioned stream is shown as three arrows. Different partitions of an input are processed in parallel with one operator instance for each partition. Partitioning operators are shown with a grey background. A sequence of operators between two partitioning operators are grouped together into a pipeline and may run on the same machine. Note that a partitioning operator may consist of two suboperators, the first generating partitions on source machines and the second merging corresponding source partitions on destination machines.

- (1) - (3). The input event files are randomly partitioned and distributed across all machines in the cluster. After extracting the "ProcessStarted" event, the input is sorted on each machine by the grouping columns, $\{starttime, guid\}$ and duplicates removed locally (local aggregation) to reduce data before hitting the network in the next stage. The data is sorted by the grouping columns so a streaming aggregation operator is used. The intermediate result after (3) is randomly partitioned with each partition sorted. Similar operations are performed on the "ProcessEnded" event except that the sorting and aggregation are on $\{endtime, usergroup, guid\}$.
- (4). To do a full aggregate, the intermediate result must be repartitioned by the grouping columns so all rows with the same value of the grouping columns end up on the same destination machine. Each destination machine



(a) Default Plan (b) Optimized Plan

Fig. 3. Query Plan Comparison

sort-merges its inputs from the source machines so the sort order is maintained. Similar operations apply to both inputs, except that one side is partitioned and sort-merged by $\{startTime, guid\}$ and the other side is by $\{endtime, usergroup, guid\}$.

- (5) - (6). The global aggregates are then calculated in parallel. On each machine, the result is next sorted by the $guid$ column in preparation for the subsequent merge join.
- (7) - (8). The intermediate result is repartitioned by the join column $guid$ of the merge join. Each destination machine sort-merges its partitions to maintain the sort order on $guid$ and the result flows directly into the merge join instance on that machine.
- (9) - (10). The join results are resorted locally on the next grouping column $usergroup$ and a local aggregate is applied immediately to reduce the data.
- (11). The data is repartitioned on $usergroup$ and each destination machine sort-merges its inputs to retain the sort order on $usergroup$.
- (12). All rows from the $usergroup$ now reside on the same machine and the global aggregate on $usergroup$ is calculated.
- (13) - (14). Finally, all result partitions are routed to the same machine that sort-merges them, producing a single sorted result that is output to a Cosmos file.

The default plan uses 150 partitions. This slight over-partitioning improves load balancing and mitigates effects of data skew.

Figure 3(b) shows the optimized plan generated by the optimizer. The plan also chooses a merge join and stream aggregates. As described in Algorithm 1 in Section III, we first determine input properties required by individual operators. Based on the rules in Table VI, the partitioned merge join requires both inputs to have structural properties $\{\{guid\}; \{guid^o, *\}\}$. The partitioned stream aggregate on "ProcessStarted" events requires an input with properties

$$\{\mathcal{X}; \{\{starttime, guid\}^g, *\}\}, \emptyset \subset \mathcal{X} \subseteq \{starttime, guid\}$$

and the partitioned stream aggregate on "ProcessEnded" events requires its inputs to have properties

$$\{\mathcal{Y}; \{\{endtime, usergroup, guid\}^g, *\}\},$$
$$\emptyset \subset \mathcal{Y} \subseteq \{endtime, usergroup, guid\}$$

- (1) - (4). In the optimized plan the inputs are sorted by $\{guid^o, starttime^o\}$ and $\{guid^o, endtime^o, usergroup^o\}$, respectively, and then partitioned on $guid$. By the inference rules in Section V and the derivation rules in Table IV, the results have properties $\{\{guid\}^g; \{guid^o, starttime^o\}\}$ and $\{\{guid\}^g; \{guid^o, endtime^o, usergroup^o\}\}$, respectively. These properties satisfy the requirements of its DISTINCT aggregate.
- (5) - (6). After the DISTINCT aggregates, the outputs have the properties $\{\{guid\}^g; \{guid^o, starttime^o\}\}$ and $\{\{guid\}^g; \{guid^o, endtime^o, usergroup^o\}\}$, respectively. The properties satisfy the requirements of the partitioned merge join so there is no need to resort or repartition the inputs. This avoids the unnecessary sorting and repartitioning in the default query plan.
- (7) - (11). The optimizer also takes advantage of the fact that join results in this particular query are relatively small and decides not to repartition on $usergroup$. Instead, it sort-merges the inputs into a single serial output in (9) and performs the global aggregate on a single machine (10). The final result is then output directly into a Cosmos file.

Compared with the default query plan, the optimized plan saves unnecessary sorting and repartitioning. The default query plan takes 21 minutes to finish while the optimized query plan takes around 11 minutes, a speedup of close to 2X. The optimization process takes much less than a second and the cost is negligible.

X. Related Work

Reasoning about plan properties has received considerable attention in the research community for the last two decades. However, all previous work focused on inferring sorting and/or grouping properties. To the best of our knowledge, this is the first paper to integrate reasoning about partitioning, grouping, and sorting in a uniform framework.

System R [15] pioneered keeping track of ordering information for intermediate query results in order to influence the choice of sort-merge joins. Simmen, et al. [16] showed how to exploit functional dependencies to infer sorting properties. Wang, et al. [17] combined reasoning about grouping and sorting together and, specifically, exploited both primary and secondary ordering information. They used a postprocessing step that derives local properties for intermediate results and eliminates unnecessary sorting and grouping operations. Our definition of local structural properties is more general and integrated directly into the optimization process. Neumann and Moerkotte described a combined platform to handle sorting and grouping optimization [13], [12]. They invented new data structures to keep track of sorting and grouping properties efficiently and simplified the normalization process needed for property comparisons. Their techniques are orthogonal to ours and can be applied in our framework as well.

Recently there has been a flurry of research on large-scale distributed computation. Google's MapReduce programming model [5] provides a simple abstraction of common group-by-aggregation operations where Map functions correspond to groupings and Reduce functions correspond to aggregations. Hadoop is an open-source version of a MapReduce execution engine. Microsoft's Dryad [10] provides a more flexible model where a distributed computation job is represented as a dataflow graph. High-level declarative languages have also been introduced that allow users to easily program distributed computation jobs. Pig Latin [14] is a data flow language using a nested data model. DryadLINQ [18] integrate Dryad with the .NET Language Integrated Query (LINQ). SCOPE [2] is a SQL-like declarative scripting language with rich classes of runtime implementations. Regardless of the language differences, their declarative nature hides system complexities from the users and need an optimizer to generate efficient execution plans.

Query processing techniques in parallel and distributed database systems have been studied extensively [11], [9]. Many traditional optimization techniques are of course applicable in the new context of cloud-scale computations. Conversely, the techniques described in this paper are also applicable in parallel database systems.

XI. Conclusion

Massive data analysis in cloud-scale data centers plays a crucial role in improving quality of service. High-level scripting languages free users from understanding various system trade-offs and complexities, and provide a transparent abstraction of the underlying system. Such languages pose great challenges for query optimization, requiring the optimizer to generate efficient parallel execution plans.

In the optimizer for SCOPE, Microsoft's scripting language for massive data analysis, consideration of parallel plans is fully integrated into the optimization process. We described the optimizer extensions required for parallel plans. A key extension was reasoning about structural properties (partitioning, grouping, and sorting properties) of data. We presented formal semantics for structural properties, introduced rules to infer properties of intermediate and final results, and described how property reasoning is integrated into the optimizer. The optimizer is in production use at Microsoft and has proven to be effective, greatly improving query performance.

References

[1] Apache. Hadoop. http://hadoop.apache.org/.

[2] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB Conference*, 2008.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of OSDI Conference*, 2006.

[4] H. Darwen and C. Date. The role of functional dependencies in query decomposition. *Relational Database Writings 1989 - 1991*, 1992.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI Conference*, 2004.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of SOSP Conference*, 2003.

[7] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceeding of SIGMOD Conference*, 1990.

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

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

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys Conference*, 2007.

[11] H. Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1994.

[12] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *Proceedings of VLDB Conference*, 2004.

[13] T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *Proceedings of ICDE Conference*, 2004.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD Conference*, 2008.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of SIGMOD Conference*, 1979.

[16] D. Simmen, E. Shekita, and T. Malkenus. Fundamental techniques for order optimization. In *Proceedings of SIGMOD Conference*, 1996.

[17] X. Wang and M. Cherniack. Avoiding sorting and grouping in processing queries. In *Proceedings of VLDB Conference*, 2003.

[18] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI Conference*, 2008.