Join Optimization Techniques for Partitioned Tables

Herodotos Herodotou
Duke University
hero@cs.duke.edu

Nedyalko Borisov
Duke University
nedyalko@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

ABSTRACT

Table partitioning splits a table into smaller parts that can be accessed, stored, and maintained independent of one another. The main use of partitioning used to be in reducing the time to access large base tables in parallel systems. Partitioning has evolved into a powerful mechanism to improve the overall manageability of both centralized and parallel database systems. Partitioning simplifies administrative tasks like data loading, removal, backup, statistics maintenance, and storage provisioning. More importantly, SQL extensions and MapReduce frameworks now enable applications and user queries to specify how derived tables should be partitioned. However, query optimization techniques have not kept pace with the rapid advances in usage and user control of table partitioning. We address this gap by developing new techniques to generate efficient plans for SQL queries involving multiway joins over partitioned tables. Our techniques are designed for easy incorporation into bottom-up query optimizers in centralized and parallel database systems. We have prototyped these techniques in PostgreSQL and in a parallel database system composed of PostgreSQL nodes managed by Hadoop. An extensive evaluation shows that our partition-aware optimization techniques, with low overhead, generate plans that are often an order of magnitude better than plans produced by current optimizers.

1. INTRODUCTION

Table partitioning is a standard feature in database systems today [12, 13, 19, 20]. For example, a sales records table may be partitioned *horizontally* based on value ranges of a date column. One partition may contain all sales records for the month of January; another partition may contain all sales records for February; and so on. A table can also be partitioned *vertically* with each partition containing a subset of columns in the table. Hierarchical combinations of horizontal and vertical partitioning may also be used.

Growing use of table partitioning: The most popular use of partitioning is in shared-nothing parallel database systems. These systems store each partition in a separate node so that the partitions can be processed in parallel during query processing. This use continues to grow with the fast increasing data sizes in high-end data warehousing systems. At the same time, two trends have turned partitioning into a popular technique used in a wide range of database systems:

- Apart from giving major performance improvements, table partitioning simplifies a number of common administrative tasks in database systems.
- MapReduce frameworks as well as SQL extensions from database vendors now give applications and user queries the power to specify how derived tables are partitioned.

Table partitioning improves ease of administration of large databases in a number of ways. Table 1 in Appendix A summarizes various uses of partitioning in modern database systems ranging from more efficient loading and removal of data on a partition-by-partition basis (reducing contention and improving data availability) to finer control over the choice of physical design, statistics creation, and storage provisioning based on the workload. Interestingly, as shown in Table 1, most high-level uses of table partitioning in modern databases apply irrespective of whether the system is centralized or parallel; only the implementation details differ.

Application and user control of partitioning: The growing use of partitioning has been accompanied by an effort to give applications and user queries the ability to specify partitioning conditions for tables that they derive from base data. MapReduce frameworks enable users to provide partitioning functions that dictate how the data output by the map tasks is partitioned across the reduce tasks. This feature is used extensively in well-tuned MapReduce programs as well as by the Hive and Pig frameworks that translate declarative queries into MapReduce programs for execution [10, 15]. Hive's SQL-like HiveQL query language offers direct support to specify table partitioning.

Application and user control of table partitioning enables both new functionality—e.g., total-order partitioning is used in MapReduce programs to produce final output (in parallel) that is totally ordered—as well as better performance—e.g., by balancing the load across all reduce tasks. There is growing interest among database vendors in providing query interfaces that combine the best features of SQL and MapReduce; and we are starting to see SQL extensions that provide first-class support for partitioning [7].

1.1 Partition-aware Query Optimization

Query optimization technology has not kept pace with the growing usage and user control over table partitioning. Previously, query optimizers only had to account for the restricted partitioning schemes that a database administra-

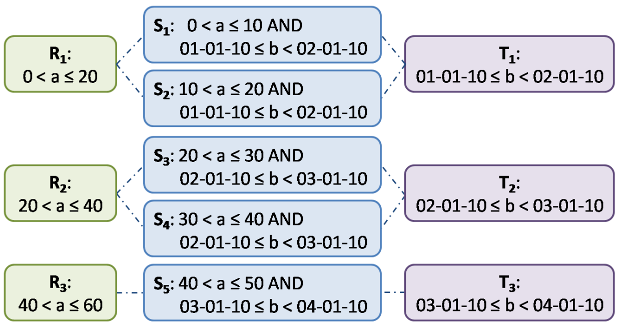
R₁: 0 < a ≤ 20
S₁: 0 < a ≤ 10 AND 01-01-10 ≤ b < 02-01-10
S₂: 10 < a ≤ 20 AND 01-01-10 ≤ b < 02-01-10
T₁: 01-01-10 ≤ b < 02-01-10
R₂: 20 < a ≤ 40
S₃: 20 < a ≤ 30 AND 02-01-10 ≤ b < 03-01-10
S₄: 30 < a ≤ 40 AND 02-01-10 ≤ b < 03-01-10
T₂: 02-01-10 ≤ b < 03-01-10
R₃: 40 < a ≤ 60
S₅: 40 < a ≤ 50 AND 03-01-10 ≤ b < 04-01-10
T₃: 03-01-10 ≤ b < 04-01-10

Figure 1: Horizontal partitions of tables R, S, T

tor (DBA) specified on the base tables in a parallel system. More challenges and opportunities exist today for optimizing queries over partitioned tables. We will illustrate them using a running-example query Q over the three partitioned tables $R(a)$, $S(a,b)$, and $T(b)$ shown in Figure 1.

```
Q: Select *
   From   R, S, T
   Where  R.a=S.a and S.b=T.b and R.a >= 5 and R.a <= 35
```

Attribute a is an integer and b is a date. Table R is equi-partitioned on ranges of a into three partitions R_1-R_3; each of partitions T_1-T_3 of table T contain all records for a respective month; table S has a multidimensional partitioning on attributes a and b into partitions S_1-S_5.

Per-table partition pruning: An optimization that many current optimizers would apply to Q is to *prune* out partitions R_3 and S_5 from consideration because records with $R.a > 40$ cannot be part of Q's result. (S_5 will be pruned only if the optimizer does a transitive closure of the join and filter predicates in Q.) Partition pruning can speed up query performance drastically by eliminating unnecessary table and index scans as well as reducing memory needs, disk spills, and contention-related overheads.

Join-aware partition pruning: Partition T_3 can also be pruned using a deeper optimization technique that most current optimizers do not use. To apply this optimization, the optimizer has to reason that records in T_3 can only join with records in S_5 and R_3 because of the partitioning conditions (illustrated using dotted lines in Figure 1). Pruning S_5 and R_3 leaves T_3 with no joining records, so T_3 can be pruned.

Partition-aware join path selection: Consider a centralized database system that stores all the partitions from Figure 1 on a single node. Plan P_1 from Figure 2 is an example plan that this system may use to process Q (assuming that the optimizer is smart enough to prune R_3 and S_5). Plan P_1 logically appends the unpruned partitions for each table, and joins the appended partitions using (centralized) join algorithms like hash and merge join. The partitions are accessed using table or index scans.

Depending on the data properties, physical design, and storage characteristics in the database system, a plan like P_2 shown in Figure 2 can significantly outperform plan P_1. P_2 exploits a number of properties of the given setting:

- Records in partition R_1 can join only with $S_1 \cup S_2$ and T_1. Similarly, records in partition R_2 can join only with $S_3 \cup S_4$ and T_2. Thus, the full $R \bowtie S \bowtie T$ join can be broken up into joins that are smaller and more efficient.
- The best join order for $R_1 \bowtie (S_1 \cup S_2) \bowtie T_1$ can be different from that for $R_2 \bowtie (S_3 \cup S_4) \bowtie T_2$. One likely reason is change in the data properties of tables S and T over time, causing variations in statistics across partitions.[1]
- The best choice of join operators for $R_1 \bowtie (S_1 \cup S_2) \bowtie T_1$ may differ from that for $R_2 \bowtie (S_3 \cup S_4) \bowtie T_2$, e.g., due to storage or physical design differences across partitions (e.g., index created on one partition but not on another).

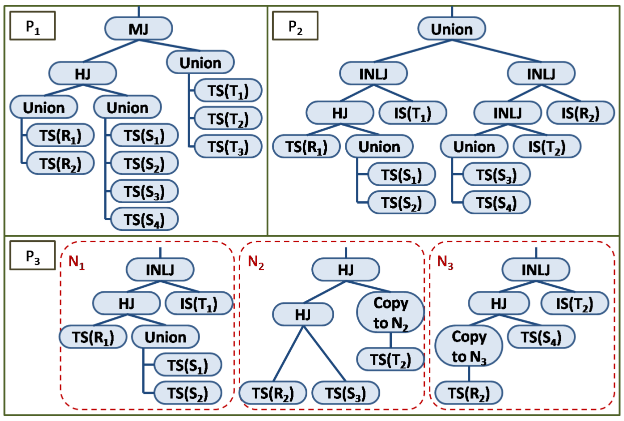

Figure 2: Plans for query Q. P1: from current optimizer; P2, P3: from our advanced optimizer

Next, let us consider the same query in a parallel database system where the relevant partitions are stored as follows: (i) Node N_1 stores partitions R_1, S_1, S_2, and T_1; (ii) Node N_2 stores partitions R_2 and S_3; and (iii) Node N_3 stores partitions S_4 and T_2. Now an additional complexity arises: partitions that need to be joined are not all on the same node, so some partitions have to be copied to other nodes.

Plan P_3 in Figure 2 is an example plan for Q in the parallel system. P_3 consists of three subplans that execute on nodes N_1, N_2, and N_3, respectively producing the result fragments $R_1 \bowtie (S_1 \cup S_2) \bowtie T_1$, $R_2 \bowtie S_3 \bowtie T_2$, and $R_2 \bowtie S_4 \bowtie T_2$. (If needed, the full join result can be copied finally to a single node.) The *communication cost* of P_3 includes copying partition R_2 from N_2 to N_3, and T_2 from N_3 to N_2. Depending on the computation Vs. communication cost tradeoff in the parallel system, there may be other plans that are superior to P_3.

1.2 Contributions

The above example illustrates the space of optimization possibilities for queries over partitioned tables. To our knowledge, no current optimizer (commercial or research prototype) takes this full space into account to find efficient plans with low optimization overhead. We address this limitation by developing an *advanced partition-aware query optimizer*:

- Our new techniques are designed for easy incorporation into bottom-up query optimizers (like the seminal System R optimizer [18]) that are in wide use today. With this design, we can leverage past investment as well as future enhancements to these optimizers (e.g., new rewrite rules, new join operators, and cost model improvements).
- In addition to conventional DBA-specified conditions on base tables, our optimizer can optimize for a wide range of user-specified partitioning conditions including multi-dimensional conditions or those involving varying ranges.
- We have developed versions of our optimizer for both centralized and parallel database systems, prototyped respectively in PostgreSQL and a shared-nothing system of PostgreSQL nodes managed by Hadoop (like [1]).
- An extensive evaluation shows that our optimizer, with low optimization-time overhead, generates plans that outperform plans generated by current optimizers.

[1]Most enterprises keep 6-24 months of historical data online.

2. RELATED WORK

Various table partitioning strategies and techniques to find good partitioning strategies automatically have been proposed (e.g., [17, 3]). Because of space constraints, this section focuses on work related to partition-aware optimization.

Partition pruning: Most commercial optimizers have excellent support for per-table partition pruning. In addition to optimization-time pruning, systems like IBM DB2 support execution-time pruning of partitions, e.g., to account for join predicates in index-nested-loop joins [12].

Partition-aware join processing: Some optimizers generate plans containing n *one-to-one partition-wise joins* for any pair of tables R and S that are partitioned into the same number n of partitions with one-to-one correspondence between the partitions [13, 20]. For joins where only table R is partitioned, Oracle supports dynamic partitioning of S based on R's partitioning; effectively creating a one-to-one join between the partitions. Partition-aware join algorithms used in parallel database systems are described in Section 6.

Selectivity-based partitioning [16], *content-based routing* [4], and *conditional plans* [6] are techniques that enable different plans to be used for different subsets of the input data. Unlike our work, these techniques focus on dynamic partitioning of (unpartitioned) tables and data streams rather than exploiting the properties of existing partitions. Furthermore, seamless incorporation into widely-used optimizers in centralized and parallel database systems is not a focus of [4, 6, 16]. A recent work considers multiway join processing in MapReduce frameworks, but does not consider the problem of using existing partitions efficiently [2].

3. PROBLEM AND SOLUTION OVERVIEW

Our goal is to generate an efficient plan for a SQL query that contains joins of partitioned tables. In this paper, we focus on tables that are partitioned horizontally based on conditions specified on one or more *partitioning attributes* (columns). The condition that defines a partition of a table is an expression involving any number of binary subexpressions of the form *Attr Op Val*, connected by *and* or *or* logical operators. *Attr* is an attribute in the table, *Val* is a constant, and *Op* is one of $\{=, \neq, <, \leq, >, \geq\}$.

Joins in the SQL query can be equi or nonequi joins. The joined tables could have different numbers of partitions (like in Figure 1). Furthermore, the partitions between joined tables need not have one-on-one correspondence with each other. For example, one table may have one partition per month while the other has one partition per day.

We consider both centralized and shared-nothing parallel databases with three key differences between them:

- Physical join operators in a parallel system differ from their centralized counterparts. Details are in Section 6.
- Each partition in a shared-nothing parallel system is associated with the location of the node where the partition resides. This information is usually treated by bottom-up optimizers (including ours) as a *physical property* [8].
- A *two-phase approach*, rewrite and join ordering followed by parallelization [11, 9], is used by most parallel optimizers (including ours) to reduce optimization complexity.

Our approach for partition-aware query optimization is based on extending bottom-up query optimizers. We will give an overview of the well-known System R bottom-up query optimizer [18] on which a number of current optimizers are based, followed by an overview of the extensions we make.

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 plan for the full query is found. First, the best *access path* (e.g., table or index scan) is found for each table in the query. The best *join path* is then found for each pair of joining tables R and S in the query. The join path consists of a physical join operator (e.g., hash or merge join) and the access paths found earlier for the tables. Next, the best join path is found for all three-way joins in the query; and so on.

Bottom-up optimizers pay special attention to physical properties like sort order and partition location that affect the ability to generate the optimal plan for an expression e by combining optimal plans for subexpressions of e. For example, for $R \bowtie S$, the (centralized) System R optimizer stores the optimal join path for each *interesting sort order* [18] of $R \bowtie S$ that can potentially reduce the plan cost of any larger expression that contains $R \bowtie S$ (e.g., $R \bowtie T \bowtie S$).

Our extensions: Consider the join path selection in a bottom-up optimizer for two partitioned tables R and S. R and S can be base tables or the result of intermediate subexpressions. Let the respective partitions be R_1-R_r and S_1-S_s ($r, s \geq 1$). For ease of exposition, we call R and S the *parent tables* in the join, and each R_i (S_j) a *child table*. By default, the optimizer will consider a join path corresponding to $(R_1 \cup R_2 \cdots \cup R_r) \bowtie (S_1 \cup S_2 \cdots \cup S_s)$, i.e., a physical join operator that takes the bag union of the child tables as input. This approach leads to plans like P_1 in Figure 2.

Partition-aware optimization must consider joins among the child tables in order to get efficient plans like P_2 in Figure 2; effectively, pushing the join below the union(s). Joins of the child tables are called *child joins*. When the bottom-up optimizer considers the join of partitioned tables R and S, we extend its search space to include plans consisting of the union of child joins. This process works in four phases: *applicability testing*, *matching*, *clustering*, and *path creation*.

Applicability testing: We first check whether the specified join conditions between R and S match the partitioning conditions on R and S appropriately. Intuitively, efficient child joins can be utilized only when the partitioning columns are part of the join attributes. For example, the $R.a = S.a$ join condition makes it possible to utilize the $R_1 \bowtie (S_1 \cup S_2)$ child join in plan P_2 in Figure 2.

Matching: This phase uses the partitioning conditions to determine efficiently which joins between individual child tables of R and S can potentially generate output records, and prune the empty child joins. For $R \bowtie S$ in our running example query, this phase outputs $\{(R_1, S_1), (R_1, S_2), (R_2, S_3), (R_2, S_4)\}$; the remaining child joins are pruned.

Clustering: Production deployments can contain tables with hundreds of partitions that lead to a large number of joins between individual child tables.[2] To reduce the join path creation overhead, we carefully cluster the child tables; details are in Section 5. For $R \bowtie S$ in our running example and a centralized database, the matching phase's output is clustered such that only the two child joins $R_1 \bowtie (S_1 \cup S_2)$ and $R_2 \bowtie (S_3 \cup S_4)$ are considered during path creation.

[2]We are aware of such deployments in a leading social networking company and for a commercial parallel DBMS.

Path Creation: This phase creates and costs join paths for all child joins output by the clustering phase, as well as the path that represents the union of the best child-join paths. This path will be chosen for $R \bowtie S$ if it costs lower than the one produced by the optimizer without our extensions.

The next three sections give the details of these phases. Section 6 will also show how plans with different join orders among the child tables (like plan P_2) will get generated as the optimizer considers larger join expressions progressively.

4. MATCHING PHASE

Suppose the bottom-up optimizer is in the process of selecting the join path for parent tables R and S with respective child tables $R_1, \ldots, R_r$ and $S_1, \ldots, S_s$. The goal of the matching phase here is to generate all join pairs R_i, S_j such that $R_i \bowtie S_j$ can produce output tuples as per the given partitioning and join conditions. An obvious matching algorithm would enumerate and check all the $r \times s$ child table pairs. The real inefficiency from this quadratic algorithm is that it gets invoked from scratch for each distinct join of parent tables considered throughout the bottom-up optimization process. Recall that R and S can be base tables or the result of intermediate subexpressions.

Partition Index Trees (PITs): We developed a more efficient matching algorithm that builds, probes, and reuses *Partition Index Trees (PITs)*. The core idea is to associate each child table with one or more *intervals* generated from the table's partitioning condition. An interval is specified as a numeric range (e.g., $(0, 10]$) or a single numeric or categorical value (e.g., $[5, 5]$, $[Jan, Jan]$). A PIT indexes all intervals of all child tables for one of the partitioning columns of a parent table. The PIT then enables efficient lookup of the intervals that overlap with a given probe interval from the other table. Use of PITs provides two main advantages:

- For most practical partitioning and join conditions, building and probing PITs has $O(r \log r)$ complexity. The memory needs are $\theta(r)$.
- Most PITs are built once and then reused many times over the course of bottom-up optimization. For example, consider the three-way join condition $R.a{=}S.a$ AND $R.a{=}T.a$. The same PIT on $R.a$ will be reused for four joins: $R \bowtie S$, $R \bowtie T$, $(R \bowtie S) \bowtie T$, and $(R \bowtie T) \bowtie S$.

Matching algorithm: Suppose $R.a{=}S.a$ is the join condition, and the partitioning conditions are simple ranges on $R.a$ and $S.a$ respectively. Also, let R have fewer child tables than S. Then, the matching algorithm works as follows:

- *Build phase:* For each child table R_i of R, generate the interval for R_i's partitioning condition. Build a PIT that indexes all intervals for R.
- *Probe phase:* For each child table S_j of S, generate the interval *int* for S_j's partitioning condition. Probe the PIT on $R.a$ to find intervals overlapping with *int*. Only R's child tables corresponding to these intervals can have tuples joining with S_j; output the identified join pairs.

For $R \bowtie S$ in our running example query, the PIT on $R.a$ will contain the intervals $(0, 20]$ for child table R_1 and $(20, 40]$ for R_2. When this PIT is probed with the interval $(0, 10]$ for child table S_1, the result will be the interval $(0, 20]$; indicating that only R_1 will join with S_1.

Implementation: The description so far was simplified for ease of presentation. The full details are given in Appendix B.1. A number of nontrivial enhancements to PITs and the matching algorithm were needed to support complex

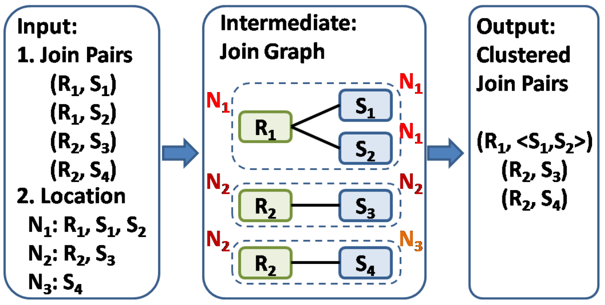

Figure 3: Clustering algorithm applied to the running example query in a parallel system

partitioning and join conditions that can arise in practice. PIT, at a basic level, is an augmented *red-black* tree [5]. However, PITs need support for multiple types of intervals: open, closed, partially closed, one sided (e.g., $(-\infty, 3]$), and single values. In addition, supporting nonequi joins required support from PITs to efficiently find all intervals in the tree that are to the left or to the right of the probe interval.

Both partitioning and join conditions can be complex combinations of AND and OR subexpressions, as well as involve any operator in $\{=, \neq, <, \leq, >, \geq\}$. Our implementation handles all these cases by restricting PITs to unidimensional indexes and handling complex expressions appropriately in the matching algorithm; see Appendix B.1.

5. CLUSTERING PHASE

The number of join pairs output by the matching phase can be large, e.g., when each child table of R joins with multiple child tables of S. In such settings, it becomes important to reduce the number of join pairs that need to be considered during join path creation. (Join path creation has overheads like cardinality estimation, enumerating join operators, as well as accessing catalogs, cost models, and statistics.) The approach we use to reduce the number of join pairs is to cluster together multiple child tables of the same parent table. Figure 3 shows the example from Section 1 for the parallel database system. The four join pairs output by the matching phase are shown. If S_1 is clustered with S_2, and S_3 with S_4, then only the two (clustered) join pairs $R_1 \bowtie (S_1 \cup S_2)$ and $R_2 \bowtie (S_3 \cup S_4)$ need to be considered.

Clustering metric: When can two child tables S_j and S_k be clustered together? First and foremost, there must exist a child table R_i such that the matching phase outputs the join pairs (R_i, S_j) and (R_i, S_k). Otherwise, a union of S_j and S_k will lead to unneeded joins with child tables of R.

Second, S_j and S_k must be indistinguishable from the perspective of joining with R_i. Stated otherwise, S_j and S_k must not have any property that makes it important to consider separate join paths for $R_i \bowtie S_j$ and $R_i \bowtie S_k$. One such property is the location of the child table in a parallel database system. Consider our running example query from Section 1 where partition S_3 is on node N_2, while S_4 is on N_3. In this case, $R_2 \bowtie S_3$ and $R_2 \bowtie S_4$ have to be treated independently because the latter needs an inter-node copy while the former does not.

Other interesting properties that make it important to consider S_j and S_k independently with respect to the join with R_i include: (i) the presence of an index on the joining attribute on S_j, but not on S_k; or (ii) S_j is stored on a faster disk than S_k. Such properties are associated as a *tag* with each child table. For example, the tag per child table in Figure 3 is the node where the table is located.

```
Algorithm for clustering the output of matching phase
Input: Join pairs, with a tag per child table
Output: Clustered join pairs input to join path creation phase
Build the bipartite join graph from the input join pairs;
For each vertex $V$ in the join graph {
    Let $t_1, \ldots, t_k$ be the tags of vertices connected to $V$;
    Create new vertices $V_1, \ldots, V_k$ in the join graph;
    Add edges from $V_k$ to each vertex with tag $t_k$ connected to V;
    Remove vertex V and all its edges from the join graph;
}
Use breadth first search to identify each connected component
        in the modified join graph;
Output a clustered join pair per connected component;
```

Figure 4: Clustering algorithm

Clustering algorithm: Figure 4 shows the clustering algorithm that takes as input the join pairs output by the matching algorithm. Each child table is associated with a tag; with the default being that all tags are the same. The algorithm first constructs the *join graph* from the input join pairs. Each child table is a vertex in this bipartite graph, and each join pair forms an edge between the corresponding vertices. Figure 3 shows the join graph for our example.

As described in Figure 4, each vertex V in the join graph is split further if V is connected to vertices with different tags. For example, in Figure 3, R_2 is connected to S_3 and S_4 that have tags N_2 and N_3 respectively. Hence, R_2 will get split so that edges from S_3 and S_4 go to different copies.

Finally, each connected component in the modified join graph will give a (possibly clustered) join pair. Thus, the output of the clustering phase in Figure 3 consists of the three joins $R_1 \bowtie (S_1 \cup S_2)$, $R_2 \bowtie S_3$, and $R_2 \bowtie S_4$. In contrast, if all the partitions were on the same (centralized) node, then the output of the clustering phase will consist of the two joins $R_1 \bowtie (S_1 \cup S_2)$ and $R_2 \bowtie (S_3 \cup S_4)$.

6. PATH CREATION

This phase creates and costs join paths for all (clustered) child joins output by the clustering phase, as well as the union of the best child-join paths. We leverage existing optimizer functionality to create join paths since path creation is coupled tightly with the physical join operators supported by the system. Control is then given back to the optimizer which picks and stores the least-cost path for the join of the parent tables, and continues bottom-up optimization.

The physical join operators in a centralized database system include hash, (sort) merge, and nested-loop joins. Join operators in parallel systems include *collocated*, *directed*, and *repartitioned* joins [17]. A collocated join applies to tables that reside on the same node. A directed join moves one table to the node containing the other table to do the join. A repartitioned join reads both tables, applies a new partitioning condition (e.g., hash) on the join columns, and sends corresponding partitions to nodes that will perform the join.

The combination of how the bottom-up optimizer enumerates joins of all parent tables with how our techniques create paths for child joins makes it possible to produce plans like P_2 in Figure 2 where different child joins can have different join orders or operators. We illustrate this observation using our running example query in a centralized database. During bottom-up optimization, the optimizer will find the best join path for $R \bowtie S$. The matching and clustering phases will create the child joins $R_1 \bowtie (S_1 \cup S_2)$ and $R_2 \bowtie (S_3 \cup S_4)$; and path creation generates the best join paths for them. The best join path for $R_1 \bowtie (S_1 \cup S_2)$ could involve a hash join, while the best path for $R_2 \bowtie (S_3 \cup S_4)$ involves a merge join. A similar process occurs for $S \bowtie T$.

As optimization progresses, the optimizer will consider $(R \bowtie S) \bowtie T$. Our techniques will then treat $R \bowtie S$ as a parent table with two child tables $R_1 \bowtie (S_1 \cup S_2)$ and $R_2 \bowtie (S_3 \cup S_4)$, and proceed to generate join paths for the child joins $(R_1 \bowtie (S_1 \cup S_2)) \bowtie T_1$ and $(R_2 \bowtie (S_3 \cup S_4)) \bowtie T_2$. Later, the optimizer will consider $(S \bowtie T) \bowtie R$, which leads to join paths for $((S_1 \cup S_2) \bowtie T_1) \bowtie R_1$ and $((S_3 \cup S_4) \bowtie T_2) \bowtie R_2$. It is possible that the best join path for $(R_1 \bowtie (S_1 \cup S_2)) \bowtie T_1$ is better than that for $((S_1 \cup S_2) \bowtie T_1) \bowtie R_1$, while the opposite occurs between $(R_2 \bowtie (S_3 \cup S_4)) \bowtie T_2$ and $((S_3 \cup S_4) \bowtie T_2) \bowtie R_2$; which will lead to the selection of plan P_2 (Figure 2) for the query.

Cardinality estimates are needed for costing during path creation. Often, statistics are kept at the level of partitions. When the optimizer considers joining unions of partitions, the estimates are made by aggregating statistics over partitions. This process can give inaccurate estimates and suboptimal plan choices. A benefit of considering child joins over partitions is that it increases the chances of using partition-level statistics directly for costing. More details are provided in Appendix B.3.

7. EXPERIMENTAL EVALUATION

In order to demonstrate the applicability of our techniques, we have prototyped partition-aware query optimizers for two different systems: (i) a centralized database serving requests from a single node, and (ii) a parallel database, where data is partitioned across multiple physical nodes. The centralized database server used is PostgreSQL 8.3.7. All experiments for this setting were run on an Ubuntu Linux 9.04 machine, with an Intel Core Duo 3.16GHz CPU, 4GB of RAM, and an 160GB High Reliability SATA 3.0Gb/s hard drive. For the parallel setting, we developed a system composed of PostgreSQL nodes managed by Hadoop (like [1]). The data is partitioned across the cluster nodes and stored within local PostgreSQL instances. The Hadoop MapReduce execution engine is used to relay commands and (sub)queries to the individual nodes (implementation details are in Appendix B.2). Our parallel system supports the join paths discussed in Section 6. For the experimental evaluation of the parallel setting, we used a Hadoop cluster with 16 nodes. Each node runs a Debian 5.0.2, has an Intel Xeon 2GHz CPU, 1.8GB RAM, and 30GB of local storage.

We used the TPC-H Benchmark with a scale factor of 10 and 50 for the centralized and parallel settings respectively. Following directions from the TPC-H Standard Specifications [21], we partitioned the tables on primary and foreign key columns creating multidimensional partitioning conditions. Unless otherwise noted, the experimental results were obtained using the partitioning schema presented in detail in Appendix C, with each table split into 200 partitions.

7.1 Experimental Setup

For our evaluation, we categorized query optimizers into three categories based on how they use partitioning information to perform optimization:

1. *Basic:* These optimizers use partitioning only for per-table partition pruning, not for join optimizations. The default PostgreSQL optimizer falls in this category.
2. *Intermediate:* In addition to partition pruning, these optimizers generate one-to-one partition-wise joins given the right conditions (see Section 2). This category represents the state-of-the-art techniques used by modern

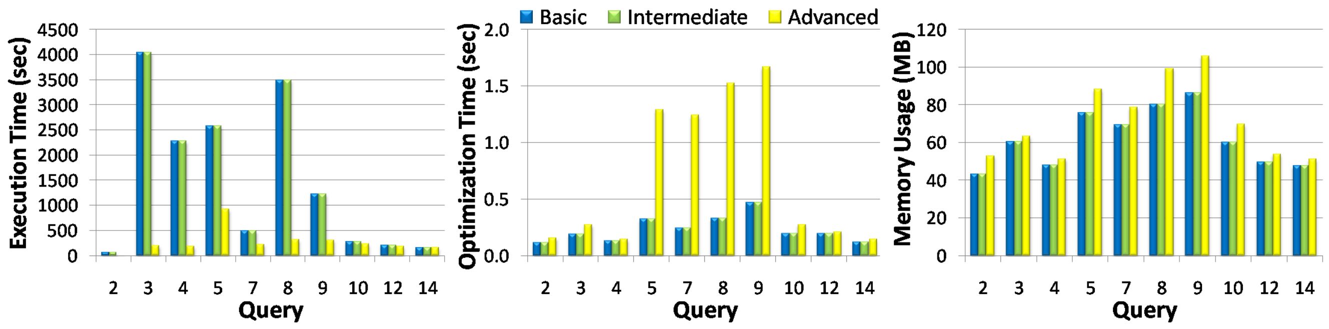

Figure 5: (a) Execution times, (b) Optimization times, (c) Memory usage for TPC-H queries

optimizers (like Oracle [13], SQLServer [20]).

3. *Advanced:* Our two partition-aware query optimizers take advantage of partitioning information to perform pruning and join optimizations as described in the paper.

For the centralized setting, we implemented a representative optimizer from each category using PostgreSQL's optimizer as the base. No robust open-source parallel optimizers are available, so we developed an advanced-category parallel optimizer. We did not implement basic and intermediate parallel optimizers. We compare the optimizers on three metrics used to evaluate optimizers: (i) query execution time, (ii) optimization time, and (iii) optimizer's memory usage.

7.2 Overall Performance Results

Figure 5(a) shows the execution times for the plans selected by the three query optimizers for ten TPC-H queries in the centralized setting. (The results for the parallel optimizer are given in Section 7.6.) The Advanced optimizer was able to generate a better plan than the other optimizers for all queries, providing up to an order of magnitude benefit for almost half of them.

We notice that the Basic and Intermediate optimizers produce the same plan in all cases, due to their inability to perform any partition-wise optimizations. First, no TPC-H query contains any filter predicates on the primary or foreign keys, which are used as the partitioning keys. Therefore, partition pruning is not an option for any optimizer (including ours). Second, one-to-one partition-wise joins are not possible because of the primary-foreign key relationships in the TPC-H schema and the partitioning schema used. (A more detailed explanation can be found in Appendix C). However, the Advanced optimizer was still able to generate partition-wise joins that joined clusters of partitions, leading to plans with significant performance improvements.

Figure 5(b) presents the optimization times for the same ten queries. The four queries in the middle of Figure 5(b) have higher optimization times as they are the most complex queries in the TPC-H benchmark. Note that extra optimization times of the Advanced optimizer over the other optimizers is fairly low, and are definitely gained back during execution as we can see by comparing the *y*-axes of Figures 5(a) and 5(b). The memory overhead introduced by our approach is also low as shown in Figure 5(c). The average extra memory overhead is around 14%, and the worse case is 22%.

7.3 Effect of Query Size

Next, we study the effects on optimizer evaluation metrics when the number of tables in the query is varied. The TPC-H queries seen in Figure 5(a) join different numbers of tables. However, most tables within a query are joined on different keys, leading to the creation of a small number of one-to-many or many-to-many partition-wise joins in each plan. For instance, since join keys do not span more that three columns, it is not possible to create four-way partition-wise joins. (Appendix C provides a detailed explanation.)

The above real-life constraints of the TPC-H database schema unfortunately limit our evaluation in two ways: (a) restricting the evaluation of our approach against the Intermediate optimizer which can exploit one-to-one partition-wise joins only; and (i) restricting a stress-test of our approach. To address these issues, we created a synthetic partitioning schema where certain tables were vertically partitioned into multiple tables. All tables were then partitioned on a single attribute (the primary or the foreign key).

Figure 6(a) shows the execution times for 5 queries with increasing number of tables. Once again, we see how the Advanced optimizer was able to generate plans that are up to an order of magnitude better compared to the plans selected by the Basic optimizer. It is interesting to note that as the number of tables in the query increases, the execution times for the plans from the Advanced optimizer barely increase (due to efficient use of child joins); unlike the Basic optimizer's plans whose execution times increase drastically. The Intermediate optimizer was able to take advantage of the partitioning information since all tables were partitioned in the same way. In this case, the Intermediate optimizer's plans were the same as the plans from the Advanced one.

Figures 6(b) and 6(c) show the optimization times and memory consumption, respectively, as the number of tables in the query increases. We note that both metrics increase non-linearly for all three optimizers; but the increase is more profound for the Intermediate and Advanced optimizers. The increasing overhead is due to a non-linear complexity of the path selection process used by the standard PostgreSQL query optimizer. Our approach introduced about the same overhead as the state-of-the-art optimizers did when implementing their (limited) join optimization features. The small additional overhead introduced by our approach is due to the creation of PITs and join graphs.

7.4 Effect of Database Size

In this section, we evaluate the performance of the optimizers as we vary the database size in terms of (i) data size, and (ii) the number of partitions created for each table. We present the results from executing TPC-H query 3 using the default partitioning schema. The results for the other queries are similar.

Figure 7(a) shows how the execution time of query 3 varies when the amount of data stored in the database increases. We observe that the execution times for the plans selected by the Advanced optimizer increase very slowly. In con-

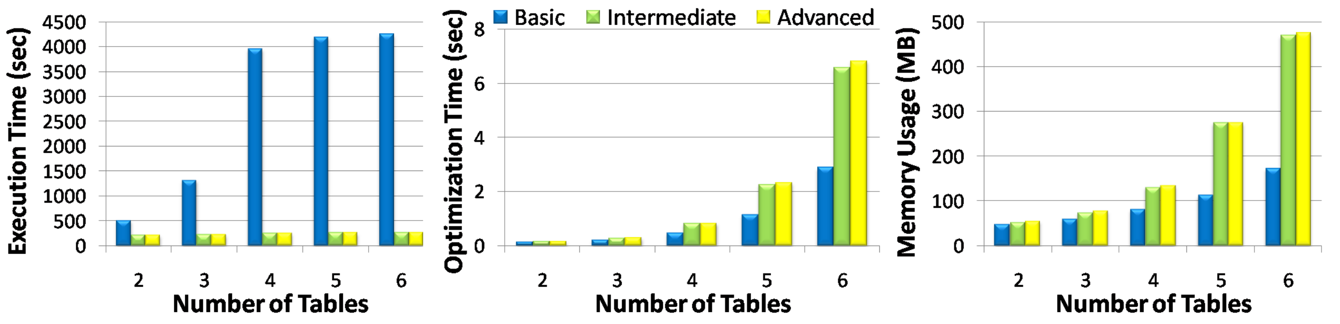

Figure 6: (a) Execution times, (b) Optimization times, (c) Memory usage for the stress-testing workload as we vary the number of tables in the queries

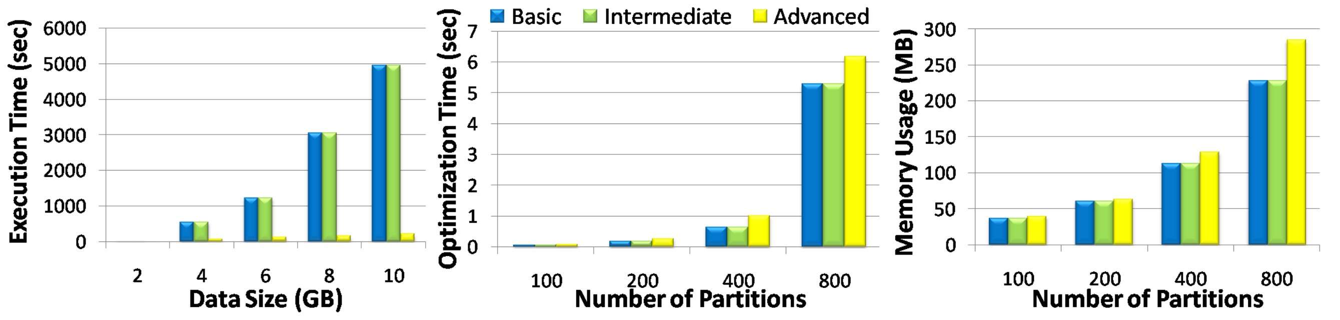

Figure 7: (a) Execution times as we vary data sizes, (b) Optimization times, (c) Memory usage as we vary the number of partitions per table for TPC-H query 3

trast, the execution times for the plans selected by the Basic and Intermediate optimizers increase rapidly as the data size grows. Figure 7(a) shows that the benefits from our approach become more important for larger databases. Note that optimization times and memory consumption are independent of the database size.

Next, we fix the data size to 10GB and vary the number of partitions for each table. Figure 7(b) shows the optimization times taken by the three optimizers. Please note the exponential scale for the number of partitions. As the number of partitions increases, so does the optimization time for all optimizers. The relatively sharp increase in optimization time for a large number of partitions is due to implementation inefficiencies of the PostgreSQL optimizer in handling a large number of partitions. The small additional overhead introduced by the Advanced optimizer is attributed to the creation of the partition-wise joins, which in turn are responsible for the great execution benefits seen in Figure 5(a). This trend is similar for the memory consumption of the optimizers as seen in Figure 7(c). Varying the number of partitions (with fixed data size) did not have a significant effect on plan execution for any optimizer.

7.5 Effect of the Clustering Algorithm

The clustering algorithm is an essential phase in our overall partition-aware optimization approach. Figures 8(b) and 8(c) compare the optimization time and memory consumption of the optimizer when clustering is enabled and disabled. In both figures, we see a high overhead introduced by disabling clustering, since the optimizer must generate join paths for each child join produced by the matching algorithm. When the matching output is one-to-many or many-to-many, the number of such partition-wise joins increases significantly, stressing the importance of clustering to keep the search space to manageable levels.

Figure 8(a) shows the execution times for the plans generated when enabling and disabling clustering. The "No Matching, No Clustering" case is the Basic optimizer and generates plans like P_1 in Figure 2. The "Matching, No Clustering" case generates plans that (intuitively) are a union of all join pairs output by the matching phase (i.e., all join pairs that can produce output tuples). In all cases, the plan generated without clustering is worse than the plan generated when clustering is used. The "Matching, No Clustering" case performs poorly. According to our partitioning schema and the join conditions, most partition-wise joins are one-to-many, i.e., the same partition joins with multiple partitions from the other table. Hence, the "Matching, No Clustering" plan may scan the same partition multiple times (in different joins). We conclude that the use of clustering is very important in order to find good execution plans.

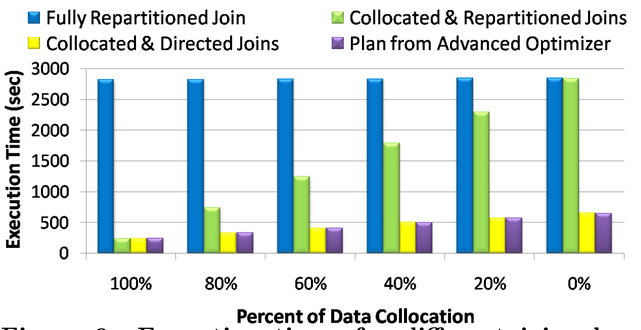

Figure 9: Execution times for different join algorithms as the percent of data collocation is varied

7.6 Effect of Collocation in the Parallel System

We now evaluate the effectiveness of the advanced parallel query optimizer using a simplified version of TPC-H query 4 (no aggregation). For comparison purposes, Figure 9 considers four different combinations of the parallel join operators: (i) fully repartitioned - repartitioned join is used without considering the partition locations; (ii) combination of

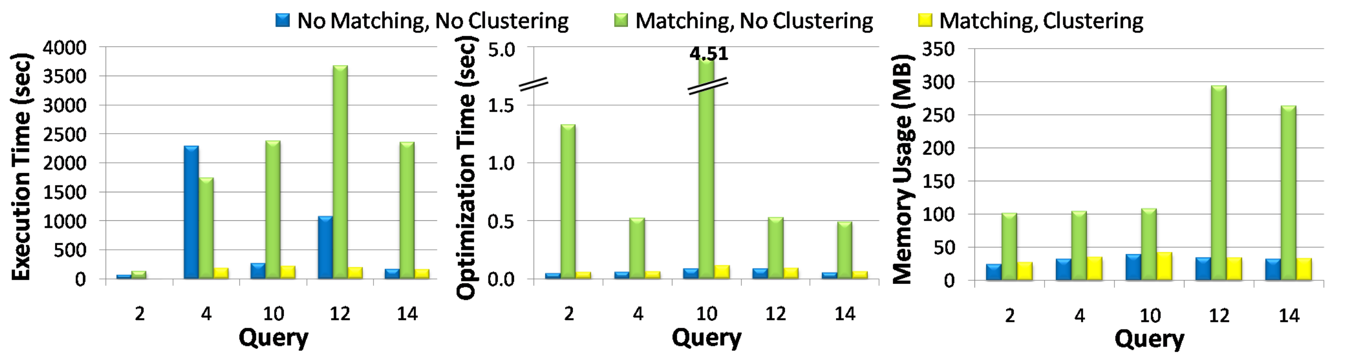

Figure 8: (a) Execution times, (b) Optimization times, (c) Memory usage for enabling and disabling clustering

collocated and repartitioned - all collocated partitions are joined with collocated joins and the rest with repartitioned joins; (iii) combination of collocated and directed - all collocated partitions are joined with collocated joins and the rest with directed joins; and (iv) our parallel optimizer's choice.

We investigated the effect of the fraction of collocated partitions for these four cases, and present the execution times for each case in Figure 9. When the partitions are fully collocated (100%), cases (ii), (iii) and (iv) take full advantage of the collocations and outperform the fully repartitioned method. As the reader might expect, when the partition collocation decreases, case (ii)'s execution time becomes closer to the fully repartitioned method. The good performance obtained by case (iii) (which is what our optimizer also selects) is in part due to the minimal movement of data through the network to perform the joins.

The fully repartitioned join is suitable for any joins irrespective of whether the condition is part of the partitioning columns or not. However, this ability comes with the price of moving all the data over the network. Over the years, commercial parallel RDBMS have used methods such as results pipelining to speed up repartitioned joins. Currently, none of these methods are supported by the Hadoop MapReduce framework, which is also shown in a recent benchmarking paper [14]. Despite the differences between the Hadoop-based and commercial parallel DBMSs, both could benefit from our partition-aware optimization techniques.

8. CONCLUSION

Query optimization technology has not kept pace with the growing usage and user control over table partitioning. We addressed this gap by developing new partition-aware optimization techniques to generate efficient plans for SQL queries. We made the following contributions:

- Our new techniques are designed for easy incorporation into bottom-up query optimizers for both centralized and parallel systems.
- We have prototyped these techniques in PostgreSQL and in a parallel shared-nothing database system composed of PostgreSQL nodes managed by Hadoop.
- An extensive evaluation showed that our optimizer, with low optimization-time overhead, generates plans that are often an order of magnitude better than plans produced by current optimizers.

9. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proc. of VLDB*, 2009.

[2] F. Afrati and J. D. Ullman. Optimizing Joins in a MapReduce Environment. In *EDBT*, 2010.

[3] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD Conference*, 2004.

[4] P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *VLDB*, 2005.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Osborne Media, 2nd edition, 2003.

[6] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, 2005.

[7] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In *Proc. of VLDB*, 2009.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), 1993.

[9] W. Hasan and R. Motwani. Coloring Away Communication in Parallel Query Optimization. In *Proc. of VLDB*, 1995.

[10] *Hive.* http://hadoop.apache.org/hive/.

[11] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, pages 9–32, 1993.

[12] IBM DB2. *Partitioned tables*, 2007. http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/ topic/com.ibm.db2.luw.admin.partition.doc/doc/ c0021560.html.

[13] T. Morales. *Oracle(R) Database VLDB and Partitioning Guide 11g Release 1 (11.1).* Oracle Corporation, 2007. http://download-uk.oracle.com/docs/cd/B28359_01/server.111/b32024/toc.htm.

[14] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, 2009.

[15] *Pig.* http://hadoop.apache.org/pig/.

[16] N. Polyzotis. Selectivity-based partitioning: a divide-and-union paradigm for effective query optimization. In *CIKM*, 2005.

[17] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *SIGMOD Conference*, 2002.

[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*, 1979.

[19] Sybase, Inc. *Performance and Tuning: Optimizer and Abstract Plans*, 2003. http://infocenter.sybase.com/help/topic/com.sybase.dc20023_1251/pdf/optimizer.pdf.

[20] R. Talmage. *Partitioned Table and Index Strategies Using SQL Server 2008.* Microsoft, 2009. http://msdn.microsoft.com/en-us/library/dd578580.aspx.

[21] TPC. *TPC Benchmark H Standard Specification*, 2009. http://www.tpc.org/tpch/spec/tpch2.9.0.pdf.

```
Algorithm for performing the matching phase
Input: Table $R$, Table $S$, Join Condition
Output: Join Pairs ($R_i$,$S_j$)
For each (binary join predicate in Join Condition) {
    Convert all partition conditions to intervals
    Build PIT with intervals from partitions of $R$
    Probe the PIT with intervals from partitions of $S$
    Adjust matching result based on logical and or or
      semantics of the Join Condition
}
```

Figure 10: Matching algorithm

APPENDIX

A. USES OF TABLE PARTITIONING

Table 1 summarizes various uses of partitioning in modern database systems ranging from more efficient loading and removal of data on a partition-by-partition basis (reducing contention and improving data availability) to finer control over the choice of physical design, statistics creation, and storage provisioning based on the workload. The table also lists whether the use benefits centralized database systems in addition to parallel database systems. Most high-level uses of table partitioning in modern database systems apply irrespective of whether the system is centralized or parallel.

B. IMPLEMENTATION DETAILS

This section provides some insightful implementation details regarding the matching phase, as well as the implementation of our parallel system.

B.1 Details of the Matching Phase

Given the joining partitioned relations R and S, the goal of the matching phase is to identify all *partition-wise join pairs* (R_i, S_j) such that $R_i \bowtie S_j$ can produce output tuples, according to the join and partition conditions. Equivalently, this algorithm can be used to prune out partition-wise joins that cannot produce any results from all possible join pairs.

Figure 10 provides all the steps for the matching algorithm. The input is the two tables to be joined and the join condition. We will describe the algorithm using our running example from Section 1. The join condition for $R \bowtie S$ is a simple equality expression, $R.a = S.a$. Later, we will discuss how the algorithm handles more complex conditions involving logical and and or operators, as well an nonequi join predicates. Since the matching phase is executed only if the Applicability Test passes (see Section 3), the column attributes $R.a$ and $S.a$ must appear in the partition conditions for the partitions of R and S respectively.

The table with the smallest number of partitions is identified as the *build* relation and the other as the *probe* relation. The conditions from the partitions of the build relation are used to create a PIT. In our example, the PIT will contain the intervals (0, 20] and (20, 40), which are associated with partitions R_1 and R_2 respectively. The conditions from the partitions of the probe relation S will be used to probe the PIT and find the overlapping intervals. For example, when the interval (0, 10] representing S_1 is used to probe the PIT, the result will be the partition R_1, indicating that it is possible for $R_1 \bowtie S_1$ to produce output tuples.

Support for complex conditions: Before building and probing the PIT, we need to convert conditions into intervals. One or more interval will be created for each needed table attribute that appears in the expression. A condition could be any expression involving logical ands, ors, and binary expressions. Sub-expressions that are anded together are used to build a single interval, whereas sub-expressions that are ored together will produce multiple intervals. For example, suppose the condition is $R.a > 0 AND R.a \leq 20$. This will create the interval $(0, 20]$. The condition $R.a > 0 AND R.b > 5$ will create the interval $(5, \infty)$, since only $R.a$ appears in the condition. The condition $R.a < 0 OR R.a > 10$ will create the intervals $(-\infty, 0)$ and $(10, \infty)$. If the particular condition does not involve $R.a$, then the created interval is $(-\infty, \infty)$, as any value for $R.a$ is possible.

Our approach can also support nonequi joins, for example $R.a < S.a$. The PIT was adjusted in order to efficiently find all intervals in the PIT that are to the left or to the right of the provided interval. Suppose $A = (A1, A2)$ is an interval in the PIT and $B = (B1, B2)$ is the probing interval. The interval A is marked as an overlapping interval if it is possible for $A < B$. Note that this is equivalent to finding all intervals that overlap with the interval $(-\infty, B2)$.

Finally, we support complex join expressions involving logical ands and ors. Suppose the join condition is ($R.a = S.a$ AND $R.b = S.b$). In this case, two PITs will be built; one for $R.a$ and one for $R.b$. After probing the two PITs, we will get two sets of join pairs. We would then adjust the pairs based on whether the join predicates are anded or ored together. In the example above, suppose that based on $R.a$, the partition R_1 can join S_1, and that based on R_b, R_1 can join both S_1 and S_2. Since the two binary join expressions are anded together, we induce that R_1 can only join S_1. However, if the join predicate were ($R.a = S.a$ OR $R.b = S.b$), then we would induce that R_1 can join both S_1 and S_2.

Complexity analysis: Building an PIT requires $O(N * logN)$ time, where N is the number of intervals. Assuming each partition has a small, fixed number of intervals (which is usually the case), N can also represent the number of partitions for the build relation. Probing a PIT with a single interval takes $O(min(N, k*logN))$ time, where k is the number of matching intervals. Hence, the overall time to identify all possible partition pairs is $O(M*min(N, k*log(N)))$, where M is the number of probing intervals.

The space overhead introduced by a PIT is the same as any binary tree, that is $\theta(N)$. However, a PIT can be reused multiple times during the optimization process. For instance, suppose the join condition for tables R, S, and T is $R.a = S.a AND R.a = T.a$. A PIT built for $R.a$ can be used for performing the matching algorithm when considering the joins $R \bowtie S$, $R \bowtie T$, $(R \bowtie S) \bowtie T$, and $(R \bowtie T) \bowtie S$.

B.2 Details of the Parallel System

Our parallel system consists of PostgreSQL nodes managed by the Hadoop system. The data is stored in the local PostgreSQL instance of each node and accessed through the MapReduce framework. We implemented the join paths described in Section 6 as follows:

- Collocated join path - the (sub)query representing the collocated join is sent and executed by the local PostgreSQL instance.
- Directed join path - the smaller join table is moved to

Use of Partitioning	Use in Centralized Databases?	Use in Parallel Databases?
Parallel access to data during query processing (e.g., parallel scans, partitioned parallelism)	Possible	Yes
Efficient elimination of access (pruning) to unneeded data during query processing	Yes	Yes
Reducing data contention during query processing and administrative tasks	Yes	Yes
Efficient and less-intrusive data loading and removal (for archival)	Yes	Yes
Faster and low-contention data backup	Yes	Yes
Efficient statistics maintenance in response to data insert/delete/update rates	Yes	Yes
Efficient table and index defragmentation	Yes	Yes
Enabling fine-grained control over physical design based on the workload	Yes	Yes
Selective storage of data on slower/faster disks based on importance and access rates	Yes	Yes
More accurate cardinality estimation during query optimization	Yes	Yes

Table 1: Uses of partitioning in centralized and parallel database management systems

the PostgreSQL instance containing the larger one and the join is performed as collocated.

- Repartitioned join path - data is read from the database tables and redistributed to the nodes performing the join, using the MapReduce default hash partitioning.

We implemented these join paths in two MapReduce jobs, called *copy* and *join*. The copy job consists of only a map phase and is responsible for copying all the smaller relations needed in the directed joins. Having a separate copy job for the directed join performed better than doing the copying in the join job, due to better network utilization. The copy job is executed only when the plan contains at least one directed join.

The join job consists of map and reduce phases, where the reduce phase is executed only when the plan contains at least one repartitioned join. In the map phase, the queries for the collocated and directed paths are executed with the results stored in the local PostgreSQL instances. For the repartitioned join, the map reads the relation tuples and transmits them to the reducers where the join is performed. The results produced from a repartitioned join are stored in the reducers' local PostgreSQL databases. To ensure that each child join is executed only once, we instrumented Hadoop to execute one mapper per each node.

Finally, we developed a bottom-up query optimizer for our parallel system. Since no other query optimizer was available, we only implemented an advanced partition-aware query optimizer supporting all techniques introduced in this paper; no basic or intermediate optimizer was built.

The optimizer is responsible for finding the best join path for each join, by creating and costing the three available join paths in the system. For the collocated joins and the join part of the directed join, we used the cost model of the PostgreSQL query optimizer. For the copy phase of the directed join and the repartitioned join, we extended the cost models of the PostgreSQL optimizer to include network transferring costs and the cost of performing a join at multiple reducers. In addition, we created a hinting mechanism that allowed us to use our optimizer to create the different combinations of parallel join operators used as part of our evaluation in Section 7.6.

B.3 Impact on Accuracy of Cardinality Estimation

In this section, we present an important benefit that child joins bring, namely better cardinality estimation. Cardinality estimation for filter and join conditions is based on data-level statistics kept by the database system for each table (e.g., distribution histograms, minimum and maximum

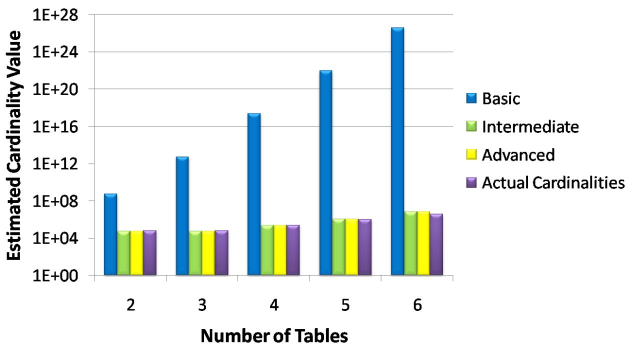

Figure 11: Cardinality estimations for the stress-testing workload as we vary the number of tables in a query

values, number of distinct values). For partitioned tables, databases like Oracle and PostgreSQL collect statistics for each individual partition. When the optimizer considers joining unions of partitions, the estimates are made by aggregating statistics over partitions.

This process however, can give inaccurate estimates since many typical statistics kept by the system cannot be easily or readily aggregated. For instance, estimating the number of unique values for all partitions is not possible by simply combining the number of unique values for each partition. Inaccurate cardinality errors can often lead to sub-optimal plan choices.

Figure 11 shows the cardinality estimates for the stress-testing workload as we vary the number of tables in a query. For the Basic Optimizer, we observe orders of magnitude in cardinality errors. In contrast, partition-wise joins provide much more accurate cardinality estimations as such joins increase the chances of using partition-level statistics directly for costing. The same pattern was observed with the default partitioning schema and the TPC-H queries.

C. PARTITIONING SCHEMAS FOR THE TPC-H DATABASE

In this section we will describe the motivation for the partitioning schema used for the TPC-H database, as well as the implications of its use. According to the TPC-H Standard Specifications [21], tables must be partitioned on primary and foreign key columns. Considering that the join conditions in most queries are equi joins over the primary and foreign keys, partitioning on these columns is the natural and correct choice.

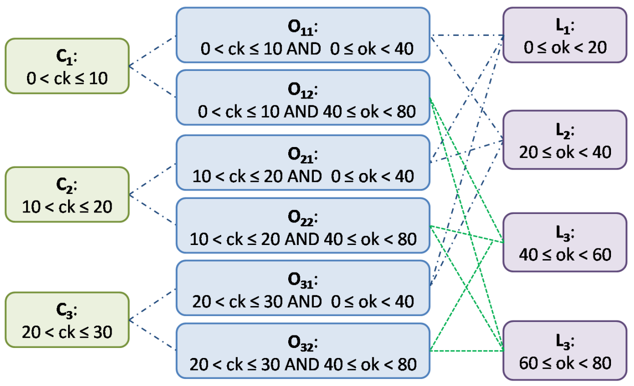
C_1: 0 < ck ≤ 10
C_2: 10 < ck ≤ 20
C_3: 20 < ck ≤ 30
O_{11}: 0 < ck ≤ 10 AND 0 ≤ ok < 40
O_{12}: 0 < ck ≤ 10 AND 40 ≤ ok < 80
O_{21}: 10 < ck ≤ 20 AND 0 ≤ ok < 40
O_{22}: 10 < ck ≤ 20 AND 40 ≤ ok < 80
O_{31}: 20 < ck ≤ 30 AND 0 ≤ ok < 40
O_{32}: 20 < ck ≤ 30 AND 40 ≤ ok < 80
L_1: 0 ≤ ok < 20
L_2: 20 ≤ ok < 40
L_3: 40 ≤ ok < 60
L_3: 60 ≤ ok < 80

Figure 12: Sample horizontal partitions of tables C, O, and L

Table	Partitioning Attributes
Customer	c_custkey (PK)
Part	p_partkey (PK)
Supplier	s_suppkey (PK)
Orders	o_orderkey (PK), o_custkey (FK)
Partsupp	ps_partkey (PK,FK), ps_suppkey (PK,FK)
Lineitem	l_orderkey (PK,FK), l_partkey (FK), l_suppkey (FK)

Table 2: Partition schema used for the TPC-H database in our experimental evaluation

However, partitioning each table on a single column is not an efficient option because most tables join with other tables on multiple columns. We will explain the issue through the use of an example. In the TPC-H schema, table *customer* (C) has the primary key *c_custkey* ($C.ck$), table *orders* (O) has the primary key *o_orderkey* ($O.ok$) and the foreign key *o_custkey* ($O.ck$), and *lineitem* (L) has the foreign key *l_orderkey* ($L.ok$). The join condition for $C \bowtie O$ is $C.ck = O.ck$ and for $O \bowtie L$ is $O.ok = L.ok$. Suppose we decide to partition C on $C.ck$ and L on $L.ok$. The question now is how do we partition O? If we partition O on $O.ck$, then any join $C \bowtie O$ will benefit from partition-wise joins. However, queries that involve the join $O \bowtie L$ will not since the join condition $O.ok = L.ok$ will not match the partitioning key $O.ck$.

The solution is to create a two-dimensional partition for O involving both attributes $O.ok$ and $O.ck$. Hence, any query involving the join $C \bowtie O$ or $O \bowtie L$ can take advantage of partition-wise joins. It is important to note that this partitioning is equivalent to partitioning O on $O.ok$ and then partitioning each new partition on $O.ck$ because the attributes $O.ok$ and $O.ck$ are not correlated.

Figure 12 shows a simple partitioning scenario where we created 3 partitions for C, 6 partitions for O, and 4 for L. When a query involving $C \bowtie O$ is optimized, our approach will induce that C_1 can only join with partitions O_{11} and O_{12}, C_2 withs O_{21} and O_{22}, and C_3 with O_{31} and O_{32} (illustrated using dotted lines in Figure 12).

Table 2 shows the actual partitioning schema we used for our experimental evaluation. According to our partitioning schema, 3 tables were partitioned on their primary key column, 2 tables where partitioned using two primary-foreign key columns, and 1 table using three foreign-key columns. Using multidimensional partitioning allowed us to maximize the use of partition-wise joins across all TPC-H queries.

An important limitation of the TPC-H schema is the fact that most tables join with other tables on different keys. For example, table O joins C on ck but joins L on ok. Therefore, it is very hard to create N-way partition-wise joins, where $N > 2$. In fact, the only common join key across three tables is *partkey*, which means it is not even possible to create N-way partition-wise joins for $N > 3$, regardless of the partitioning schema.

To alleviate the above limitations of the TPC-H database schema, we decided to vertically partition tables *lineitem*, *part*, and *partsupp* into two pieces each. Each vertical partition retained the column *partkey*, which was then used to horizontally partition the vertical partitions. This synthetic partitioning schema allowed us to stress-test our approach as well as to better evaluate our approach against the other optimizers, since we were able to create queries involving up to 6-way one-to-one partitions-wise joins. This simple exercise also demonstrates the seamless integration of our approach with systems that support row-stored vertical partitioning.