Query Optimization Techniques for Partitioned Tables

Herodotos Herodotou
Duke University
Durham, North Carolina, USA
hero@cs.duke.edu

Nedyalko Borisov
Duke University
Durham, North Carolina, USA
nedyalko@cs.duke.edu

Shivnath Babu*
Duke University
Durham, North Carolina, USA
shivnath@cs.duke.edu

ABSTRACT

Table partitioning splits a table into smaller parts that can be accessed, stored, and maintained independent of one another. From their traditional use in improving query performance, partitioning strategies have evolved into a powerful mechanism to improve the overall manageability of database systems. Table partitioning simplifies administrative tasks like data loading, removal, backup, statistics maintenance, and storage provisioning. Query language extensions now enable applications and user queries to specify how their results should be partitioned for further use. 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 that are in wide use today. We have prototyped these techniques in the PostgreSQL optimizer. An extensive evaluation shows that our partition-aware optimization techniques, with low optimization overhead, generate plans that can be an order of magnitude better than plans produced by current optimizers.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms

Keywords

query optimization, partitioning

1. INTRODUCTION

Table partitioning is a standard feature in database systems today [13, 15, 20, 21]. 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.

Uses of Table Partitioning in Database Systems
Efficient pruning of unneeded data during query processing
Parallel data access (partitioned parallelism) during query processing
Reducing data contention during query processing and administrative tasks. Faster data loading, archival, and backup
Efficient statistics maintenance in response to insert, delete, and update rates. Better cardinality estimation for subplans that access few partitions
Prioritized data storage on faster/slower disks based on access patterns
Fine-grained control over physical design for database tuning
Efficient and online table and index defragmentation at the partition level

Table 1: Uses of table partitioning in database systems

The trend of rapidly growing data sizes has amplified the usage of partitioned tables in database systems. Table 1 lists various uses of table partitioning. Apart from giving major performance improvements, partitioning simplifies a number of common administrative tasks in database systems. In this paper, we focus on horizontal table partitioning in centralized row-store database systems such as those sold by major database vendors as well as popular open-source systems like MySQL and PostgreSQL. The uses of table partitioning in these systems have been studied [2, 24].

The growing usage of table partitioning has been accompanied by efforts to give applications and users the ability to specify partitioning conditions for tables that they derive from base data. SQL extensions from database vendors now enable queries to specify how derived tables are partitioned (e.g., [11]). Given such extensions, Database Administrators (DBAs) may not be able to control or restrict how tables accessed in a query are partitioned. Furthermore, multiple objectives—e.g., getting fast data loading along with good query performance—and constraints—e.g., on the maximum size or number of partitions per table—may need to be met while choosing how each table in the database is partitioned.

1.1 Query Optimization for Partitioned Tables

Query optimization technology has not kept pace with the growing usage and user control of table partitioning. Previously, query optimizers had to consider only the restricted partitioning schemes specified by the DBA on base tables. Today, the optimizer faces a diverse mix of partitioning schemes that expand on traditional schemes like hash and equi-range partitioning. *Hierarchical* (or *multidimensional*) partitioning is one such scheme to deal with multiple granularities or hierarchies in the data [4]. A table is first partitioned on one attribute. Each partition is further partitioned on a different attribute; and so on for two or more levels.

Figure 1 shows an example hierarchical partitioning for a table $S(a, b)$ where attribute a is an integer and attribute b is a date. S

*Supported by NSF grants 0964560 and 0917062

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

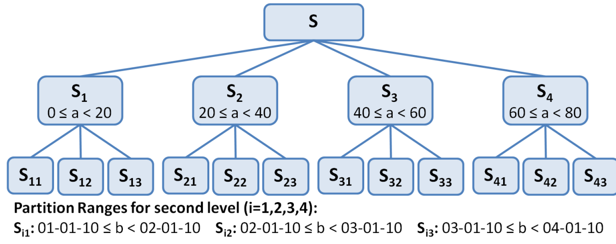

Figure 1: A hierarchical partitioning of table S

is equi-partitioned on ranges of a into four partitions S_1-S_4, each of which is further partitioned on ranges of b. Figure 2 shows how the hierarchical partitioning of table S can be interpreted as a two-dimensional partitioning. The figure also shows partitions for tables $R(a)$, $T(a)$, and $U(b)$. R, S, and T are all partitioned on a—typical for multiple related data sources or even star/snowflake schemas—but with different ranges due to data properties and storage considerations. For example, if the number of records with the same value of a is large in T (e.g., user clicks), then smaller ranges will give more manageable partitions.

Table U is partitioned using nonequi ranges on b for data loading and archival efficiency as well as workload performance. Daily partitions for daily loads are an attractive option since it is faster to load an entire partition at a time. However, maintenance overheads and database limitations on the maximum number of partitions can prevent the creation of daily partitions. Hence, 10-day ranges are used for recent partitions of U. Older data is accessed less frequently, so older 10-day partitions are merged into monthly ones to improve query performance and archival efficiency.

The flexible nature and rising complexity of partitioning schemes pose new challenges and opportunities during the optimization of queries over partitioned tables. Consider an example query Q_1 over the partitioned tables R, S, and T in Figure 2.

Q_1: Select *
From R, S, T
Where R.a=S.a and S.a=T.a
and S.b≥02-15-10 and T.a<25;

Use of filter conditions for partition pruning: An optimization that many current optimizers apply to Q_1 is to *prune* partitions T_4-T_8 and S_{11}, S_{21}, S_{31}, S_{41} from consideration because it is clear from the partitioning conditions that records in these partitions will not satisfy the filter conditions. 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.

Use of join conditions for partition pruning: Based on a transitive closure of the filter and join conditions, partition pruning can also eliminate partitions S_{32}, S_{33}, S_{42}, S_{43}, R_3, R_4, and U_1.

Most current optimizers will stop here as far as exploiting partitions during the optimization of Q_1 is concerned; and generate a plan like Q_1P_1 shown in Figure 3. In a plan like Q_1P_1, the leaf operators logically append together (i.e., UNION ALL) the unpruned partitions for each table. Each unpruned partition is accessed using regular table or index scans. The appended partitions are joined using operators like hash, merge, and (index) nested-loop joins.

Partition-aware join path selection: Depending on the data properties, physical design, and storage characteristics in the database system, a plan like Q_1P_2 shown in Figure 3 can significantly outperform plan Q_1P_1. Q_1P_2 exploits a number of properties arising from partitioning in the given setting:

- Records in partition R_1 can join only with $S_{12} \cup S_{13}$ and $T_1 \cup T_2$. Similarly, records in partition R_2 can join only with $S_{22} \cup S_{23}$ and T_3. Thus, the full $R \bowtie S \bowtie T$ join can be broken up into smaller and more efficient *partition-wise joins*.
- The best join order for $R_1 \bowtie (S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ can be different from that for $R_2 \bowtie (S_{22} \cup S_{23}) \bowtie T_3$. 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 $R_1 \bowtie (S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ may differ from that for $R_2 \bowtie (S_{22} \cup S_{23}) \bowtie T_3$, e.g., due to storage or physical design differences across partitions (e.g., index created on one partition but not on another).

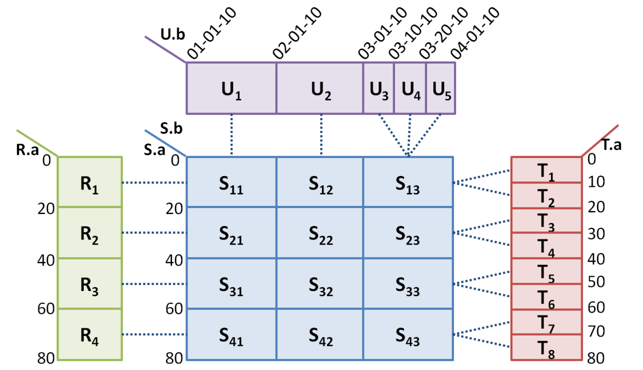

Figure 2: Partitioning of tables R, S, T, U. Dotted lines show partitions with potentially joining records

Let us now consider query Q_2 to further understand the challenges and opportunities arising while optimizing queries over partitioned tables. Q_2 is a typical query in traditional star schemas where a fact table is joined with several dimension tables on different attributes.

Q_2: Select *
From R, S, U
Where R.a=S.a and S.b=U.b
and U.b≥02-15-10 and R.a<25;

Plan Q_2P_1 from Figure 3 shows the plan that simply appends the unpruned partitions before performing the joins. Given the join and partitioning conditions for R and S, the optimizer has the option of creating the partition-wise joins $R_1 \bowtie (S_{12} \cup S_{13})$ and $R_2 \bowtie (S_{22} \cup S_{23})$. The output of these joins is logically partitioned on attribute a—which does not affect the later join with table U—leading to the plan Q_2P_2 in Figure 3. Alternatively, the optimizer has the option of creating partition-wise joins between S and U first, generating the plan Q_2P_3.

1.2 Challenges and Contributions

The above examples illustrate the optimization possibilities for SQL queries over partitioned tables, which enlarge the plan space drastically. To our knowledge, no current optimizer (commercial or research prototype) takes this space into account to find efficient plans with low optimization overhead. We address this limitation by developing a novel *partition-aware SQL query optimizer*:

Dealing with plan space explosion: A nontrivial challenge we have to address in a partition-aware optimizer is to keep the additional computational and memory overheads of the optimization process in check while enabling good plans to be found.

Incorporation into state-of-the-art optimizers: The new techniques we propose are designed for easy incorporation into bottom-up query optimizers (like the seminal System R optimizer [19]) that are in wide use today. With this design, we leverage decades of past investment as well as potential future enhancements to these optimizers (e.g., new rewrite rules, new join operators, and improvements in statistics and cardinality estimation).

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

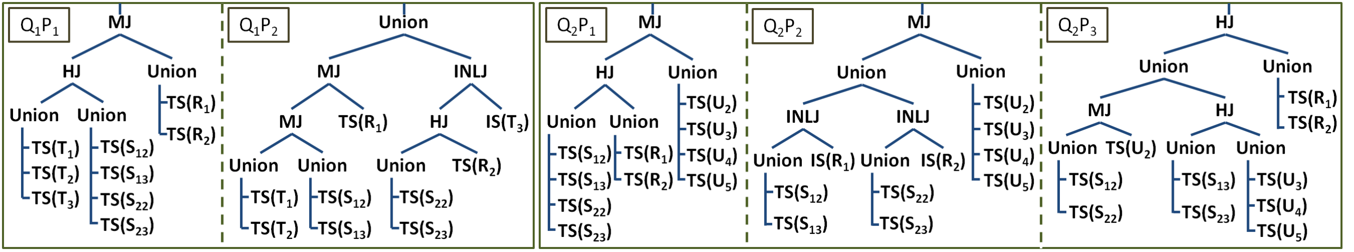

Figure 3: Q_1P_1 and Q_2P_1 are plans generated by current optimizers for our example queries Q_1 and Q_2 respectively. Q_1P_2, Q_2P_2, and Q_2P_3 are plans generated by our partition-aware optimizer. IS and TS are respectively index and table scan operators. HJ, MJ, and INLJ are respectively hash, merge, and index-nested-loop join operators. Union is a bag union operator

Partitions as physical or logical properties? The conventional wisdom in the database literature as well as implementation in commercial bottom-up query optimizers treat partitions as physical properties [18]. We show that treating partitions only as physical properties falls well short of making the best use of partitioned tables. Our optimizer considers partitions efficiently at both the logical and physical levels to get the best of two worlds: (a) generating plans like Q_1P_2, Q_2P_2, and Q_2P_3 in Figure 3, and (b) preserving *interesting partitions* [18] that may benefit operators (e.g., group-by) higher-up in the plan.

Supporting practical partitioning conditions: In addition to conventional DBA-specified partitioning conditions on base tables, our optimizer supports a wide range of user-specified partitioning conditions including multidimensional partitions, multi-level hierarchical partitions, and irregular ranges. The challenge here is to deal with complex join graphs arising at the partition level (like Figure 2) from the combination of the filter, join, and table-level partitioning conditions for a SQL query.

Improving cardinality estimates: A nonobvious effect arises from the fact that most database systems keep statistics (e.g., number of distinct values) at the level of individual partitions. Cardinality estimation for appended partitions necessitates combination of per-partition statistics. We have found that estimation errors from such combination are worse for a plan like Q_1P_1 compared to Q_1P_2.

Prototyping and evaluation: All our techniques have been prototyped in the PostgreSQL optimizer, and we report an extensive evaluation based on the popular TPC-H benchmark.

2. RELATED WORK

Various table partitioning schemes as well as techniques to find a good partitioning scheme automatically have been proposed as part of database physical design tuning (e.g., [2, 18]). In contrast, our goal is to fully exploit possible query optimization opportunities given the existing horizontal partitioning scheme in the database.

Partitioning in Centralized DBMSs: Commercial DBMS vendors (e.g., IBM, Microsoft, Oracle, and Sybase) provide support for different types of partitioning, including hash, range, and list partitioning, as well as support for hierarchical (multidimensional) partitioning. However, they implement different partition-aware optimization techniques. Most commercial optimizers have excellent support for per-table partition pruning. In addition to optimization-time pruning, systems like IBM DB2 support pruning of partitions at plan execution time, e.g., to account for join predicates in index-nested-loop joins [13]. 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 [15, 21]. 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.

UNION ALL views are a useful construct that can be used to support table partitioning [27]. The techniques proposed in this paper are related closely to pushing joins down through UNION ALL views. For example, when a UNION ALL view representing a partitioned table R=$R_1 \cup \ldots \cup R_n$ is joined with a table S, IBM DB2's query optimizer considers pushing the join down to generate a UNION of base-table joins $(R_1 \bowtie S) \cup \ldots \cup (R_n \bowtie S)$ [27]. However, unlike our techniques, the join pushdown is considered in the query-rewrite phase. As the authors of [27] point out, this step can increase the time and memory overheads of optimization significantly because of the large number of joins generated (especially, if multiple UNION ALL views are joined like in our example queries in Figure 3). The techniques we propose are designed to keep these overheads in check—even in the presence of hundreds of partitions—while ensuring that good plans can be found.

Partitioning in Parallel/Distributed DBMSs: While we focus on centralized DBMSs, the partition-aware optimization techniques we propose are related closely to *data localization* in distributed DBMSs [16]. Data localization is a query-rewrite phase where heuristic rules like filter pushdown are used to prune partitions and their joins that will not contribute to the query result. A join graph is created for the partitions belonging to the joining tables, and inference rules are used to determine the empty joins [8]. While our work shares some goals with data localization, a number of differences exist. Instead of heuristic rewrite rules, we propose (provably optimal) cost-based optimization of partitioned tables. In particular, we address the accompanying nontrivial challenge of plan space explosion—especially in the presence of hundreds of partitions per table (e.g., daily partitions for a year)—and the need to incorporate the new optimization techniques into industry-strength cost-based SQL optimizers. Section 8.7 compares our techniques empirically with an adaptation of data localization to centralized DBMSs.

The cost-based optimization algorithms we present are independent of the physical join methods supported by the DBMS. Parallel DBMSs support several partition-aware join methods including collocated, directed, broadcast, and repartitioned joins [6]. SCOPE is a system for large-scale data analysis that uses cost-based optimization to select the repartitioning of tables and intermediate results [25]. Query optimizers in these systems attempt to minimize data transfer costs among nodes, which is orthogonal to our work.

Dynamic partitioning: *Selectivity-based partitioning* [17], *content-based routing* [7], and *conditional plans* [10] are techniques that enable different execution 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. Easy incorporation into widely-used SQL optimizers is not a focus of [7, 10, 17].

Predicate optimization: *Predicate move-around* [14] is a query transformation technique that moves predicates among different relations, and possibly query blocks, to generate potentially better

plans. *Magic sets* [5] represent a complementary technique that can generate auxiliary tables to be used as early filters in a plan. Both techniques are applied in the rewrite phase of query optimization, thereby complementing our cost-based optimization techniques.

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 a SQL query can be equi or nonequi joins. The joined tables could have different numbers of partitions and could be partitioned on multiple attributes (like in Figure 2). Furthermore, the partitions between joined tables need not have one-on-one correspondence with each other. For example, a table may have one partition per month while another table has one partition per day.

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 [19] 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 physical plan for the full query is found. First, the best *access path* (e.g., table or index scan) is found and retained for each table in the query. The best *join path* is then found and retained 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 and retained for all three-way joins in the query; and so on.

Bottom-up optimizers pay special attention to physical properties (e.g., sort order) 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 System R optimizer stores the optimal join path for each *interesting sort order* [19] of $R \bowtie S$ that can potentially reduce the plan cost of any larger expression that contains $R \bowtie S$ (e.g., $R \bowtie S \bowtie U$).

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. 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 $(R_1 \cup \cdots \cup R_r) \bowtie (S_1 \cup \cdots \cup S_s)$, i.e., a physical join operator that takes the bag unions of the child tables as input. This approach leads to plans like Q_1P_1 and Q_2P_1 in Figure 3.

Partition-aware optimization must consider joins among the child tables in order to get efficient plans like Q_1P_2 in Figure 3; 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 condition 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 $R_2 \bowtie (S_{22} \cup S_{23})$ child join in plan Q_1P_2 in Figure 3.

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 to prune the empty child joins. For $R \bowtie S$ in our running example query Q_1, this phase outputs $\{(R_1, S_{12}),(R_1, S_{13}),(R_2, S_{22}), (R_2, S_{23})\}$.

Clustering: Production deployments can contain tables with many tens to 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, the matching phase's output is clustered such that only the two child joins $R_1 \bowtie (S_{12} \cup S_{13})$ and $R_2 \bowtie (S_{22} \cup S_{23})$ are considered during path creation.

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 discuss how our techniques can be incorporated into the bottom-up optimization process.

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-R_r and S_1-S_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 as per the given partitioning and join conditions. Equivalently, this algorithm prunes out (from all possible join pairs) partition-wise joins that cannot produce any results. An obvious matching algorithm would enumerate and check all the $r \times s$ possible child table pairs. In distributed query optimization, this algorithm is implemented by generating a join graph for the child tables [8]. The real inefficiency from this quadratic algorithm comes from the fact that it gets invoked from scratch for each distinct join of parent tables considered throughout the bottom-up optimization process. Note that R and S can be base tables or the result of intermediate subexpressions.

We developed an efficient matching algorithm that builds, probes, and reuses *Partition Index Trees (PITs)*. We will describe this new data structure, and then explain how the matching algorithm utilizes it to generate the partition-wise join pairs efficiently. PITs apply to range and list partitioning conditions. Section 7 describes how our techniques can be extended to handle hash partitioning.

4.1 Partition Index Trees

The core idea behind Partition Index Trees is to associate each child table with one or more *intervals* generated from the table's partitioning condition. An interval is specified as a *Low* to *High* range, which can be numeric (e.g., $(0, 10]$), date (e.g., [02-01-10, 03-01-10)), or a single numeric or categorical value (e.g., $[5, 5]$, [*url,url*]). 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. The use of PITs provides two main advantages:

- For most practical partitioning and join conditions, building and probing PITs has $O(r \log r)$ complexity (for r partitions in a table). The memory needs are $\theta(r)$.

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

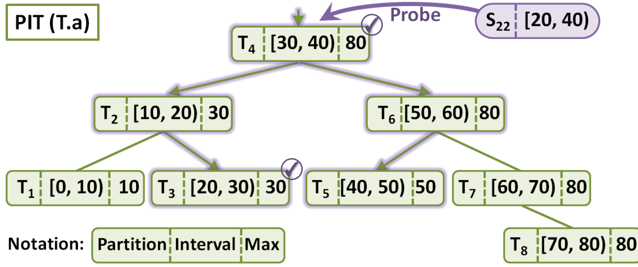

Figure 4: A partition index tree containing intervals for all child tables (partitions) of T from Figure 2

- Most PITs are built once and then reused many times over the course of the bottom-up optimization process (see Section 4.4).

Implementation: PIT, at a basic level, is an augmented *red-black* tree [9]. The tree is ordered by the *Low* values of the intervals, and an extra annotation is added to every node recording the maximum *High* value (denoted *Max*) across both its subtrees. Figure 4 shows the PIT created on attribute $T.a$ based on the partitioning conditions of all child tables of T (see Figure 2). The *Low* and *Max* values on each node are used during probing to efficiently guide the search for finding the overlapping intervals. When the interval $[20, 40)$ is used to probe the PIT, five intervals are checked (highlighted in Figure 4) and the two overlapping intervals $[20, 30)$ and $[30, 40)$ are returned.

A number of nontrivial enhancements to PITs were needed to support complex partitioning conditions that can arise in practice. First, PITs need support for multiple types of intervals: open, closed, partially closed, one sided, and single values (e.g., $(1, 5)$, $[1, 5]$, $[1, 5)$, $(-\infty, 5]$, and $[5, 5]$). 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.

4.2 Matching Algorithm

Figure 5 provides all the steps for the matching algorithm. The input consists of the two tables to be joined and the join condition. We will describe the algorithm using query Q_1 in our running example from Section 1. The join condition for $S \bowtie T$ in Q_1 is a simple equality expression: $S.a = T.a$. Later, we will discuss how the algorithm handles more complex conditions involving logical AND and OR operators, as well as nonequi join conditions. Since the matching phase is executed only if the Applicability Test passes (see Section 3), the attributes $S.a$ and $T.a$ must appear in the partitioning conditions for the partitions of S and T respectively.

The table with the smallest number of (unpruned) partitions is identified as the *build* relation and the other as the *probe* relation. In our example, T (with 3 partitions) will be the build relation and S (with 4 partitions) will be the probe one. Since partition pruning is performed before any joins are considered, only the unpruned child tables are used for building and probing the PIT. Then, the matching algorithm works as follows:

- *Build phase:* For each child table T_i of T, generate the interval for T_i's partitioning condition (explained in Section 4.3). Build a PIT that indexes all intervals from the child tables of T.
- *Probe phase:* For each child table S_j of S, generate the interval int for S_j's partitioning condition. Probe the PIT on $T.a$ to find intervals overlapping with int. Only T's child tables corresponding to these overlapping intervals can have tuples joining with S_j; output the identified join pairs.

Algorithm for performing the matching phase
Input: Relation R, Relation S, Join Condition
Output: All partition-wise join pairs (R_i,S_j) that can produce join results
For each (binary join expression in Join Condition) {
  Convert all partitioning 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 5: Matching algorithm

For $S \bowtie T$ in our running example query, the PIT on $T.a$ will contain the intervals $[0, 10)$, $[10, 20)$ and $[20, 30)$, which are associated with partitions T_1, T_2, and T_3 respectively (Figure 2). When this PIT is probed with the interval $[20, 40)$ for child table S_{22}, the result will be the interval $[20, 30)$; indicating that only T_3 will join with S_{22}. Overall, this phase outputs $\{(S_{12}, T_1)$, (S_{12}, T_2), (S_{13}, T_1), (S_{13}, T_2), (S_{22}, T_3), $(S_{23}, T_3)\}$; the remaining possible child joins are pruned.

4.3 Support for Complex Conditions

Before building and probing the PIT, we need to convert each partitioning condition into one or more intervals. A condition could be any expression involving logical ANDs, ORs, and binary expressions. Subexpressions that are ANDed together are used to build a single interval, whereas subexpressions that are ORed together will produce multiple intervals. For example, suppose the partitioning condition is ($R.a \geq 0$ AND $R.a < 20$). This condition will create the interval $[0, 20)$. The condition ($R.a > 0$ AND $R.b > 5$) will create the interval $(0, \infty)$, since only $R.a$ appears in the join conditions of query Q_1. 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 interval created 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 $\exists \alpha \in A, \beta \in B$ such that $\alpha < \beta$. Note that this check is equivalent to finding all intervals in the PIT 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 then adjust the pairs based on whether the join conditions are ANDed or ORed together. In the example above, suppose that R_1 can join with S_1 based on $R.a$, and that R_1 can join with both S_1 and S_2 based on $R.b$. Since the two binary join expressions are ANDed together, we induce that R_1 can join only with S_1. However, if the join condition were ($R.a = S.a$ OR $R.b = S.b$), then we would induce that R_1 can join with both S_1 and S_2.

4.4 Complexity Analysis

Suppose N and M are the number of partitions for the build and probe relations respectively. Also suppose each partition condition is translated into a small, fixed number of intervals (which is usually the case). In fact, a simple range partitioning condition will generate exactly one interval. Then, building a PIT requires $O(N \times \log N)$ time. Probing a PIT with a single interval takes

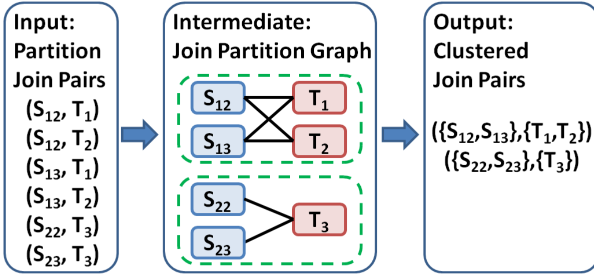

Figure 6: Clustering algorithm applied to example query Q_1

$O(min(N, k \times \log N))$ time, where k is the number of matching intervals. Hence, the overall time to identify all possible child join pairs is $O(M \times min(N, k \times \log N))$.

The space overhead introduced by a PIT is $\theta(N)$ since it is a binary tree. However, a PIT can be reused multiple times during the optimization process. Consider the join condition ($R.a$=$S.a$ AND $S.a$=$T.a$) for tables R, S, and T in Q_1. A PIT built for $S.a$ can be (re)used for performing the matching algorithm when considering the joins $R \bowtie S$, $S \bowtie T$, $(R \bowtie S) \bowtie T$, and $(S \bowtie T) \bowtie R$.

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 to avoid both optimization and execution inefficiencies. Join path creation introduces optimization-time overheads for enumerating join operators, accessing catalogs, and calculating cardinality estimates. During execution, if multiple child-join paths reference the same child table R_i, then R_i will be accessed multiple times; a situation we want to avoid.

The approach we use to reduce the number of join pairs is to cluster together multiple child tables of the same parent table. Figure 6 considers $S \bowtie T$ for query Q_1 from Section 1. The six partition-wise join pairs output by the matching phase are shown on the left. Notice that the join pairs (S_{22}, T_3) and (S_{23}, T_3) indicate that both S_{22} and S_{23} can join with T_3 to potentially generate output records. If S_{22} is clustered with S_{23}, then the single (clustered) join $(S_{22} \cup S_{23}) \bowtie T_3$ will be considered in the path creation phase instead of the two joins $S_{22} \bowtie T_3$ and $S_{23} \bowtie T_3$. Furthermore, because of the clustering, the child table T_3 will have only one access path (say, a table or index scan) in Q_1's plan.

Definition 1. **Clustering metric:** For an $R \bowtie S$ join, two (unpruned) child tables S_j and S_k of S will be clustered together iff there exists a (unpruned) child table R_i of R such that the matching phase outputs the join pairs (R_i, S_j) and (R_i, S_k). □

If S_j and S_k are clustered together when no such R_i exists, then the union of S_j and S_k will lead to unneeded joins with child tables of R; hurting plan performance during execution. In our running example in Figure 6, suppose we cluster S_{22} with S_{13}. Then, S_{22} will have to be considered unnecessarily in joins with T_1 and T_2.

On the other hand, failing to cluster S_j and S_k together when the matching phase outputs the join pairs (R_i, S_j) and (R_i, S_k) would result in considering join paths separately for $R_i \bowtie S_j$ and $R_i \bowtie S_k$. The result is higher optimization overhead as well as access of R_i in at least two separate paths during execution. In our example, if we consider separate join paths for $S_{22} \bowtie T_3$ and $S_{23} \bowtie T_3$, then partition T_3 will be accessed twice.

Clustering algorithm: Figure 7 shows the clustering algorithm that takes as input the join pairs output by the matching phase. The algorithm first constructs the *join partition 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 6 shows the join partition graph for our example. Breadth-First-Search is used to identify all the connected components in the join partition graph. Each connected component will give a (possibly clustered) join pair. Following our example in Figure 6, S_{12} will be clustered with S_{13}, S_{22} with S_{23}, and T_1 with T_2, forming the output of the clustering phase consisting of the two (clustered) join pairs $(\{S_{12}, S_{13}\}, \{T_1, T_2\})$ and $(\{S_{22}, S_{23}\}, \{T_3\})$.

Algorithm for clustering the output of the matching phase
Input: Partition join pairs (output of matching phase)
Output: Clustered join pairs (which will be input to path creation phase)
Build a bipartite join graph from the input partition join pairs where:
  Child tables are the vertices, and
  Partition join pairs are the edges;
Use Breadth-First-Search to identify connected components in the graph;
Output a clustered join pair for each connected component;

Figure 7: Clustering algorithm

6. PATH CREATION AND SELECTION

We will now consider how to create and cost join paths for all the (clustered) child joins output by the clustering phase, as well as the union of the best child-join paths. Join path creation has to be coupled tightly with the physical join operators supported by the database system. As discussed in Section 1.2, we will leverage the functionality of a bottom-up query optimizer [19] to create join paths for the database system. The main challenge is how to extend the enumeration and path retention aspects of a bottom-up query optimizer in order to find the optimal plan in the new *extended plan space* efficiently.

Definition 2. **Optimal plan in the extended plan space:** In addition to the default plan space considered by the bottom-up optimizer for an n-way ($n \geq 2$) join of parent tables, the extended plan space includes the plans containing any possible join order and join path for joins of the child tables such that each child table (partition) is accessed at most once. The optimal plan is the plan with least estimated cost in the extended plan space. □

We will discuss three different approaches on how to extend the bottom-up optimizer to find the optimal plan in the extended plan space. Query Q_1 from Section 1 is used as an example throughout.

Note that Q_1 joins the three parent tables R, S, and T. For Q_1, a bottom-up optimizer will consider the three 2-way joins $R \bowtie S$, $R \bowtie T$, $S \bowtie T$, and the single 3-way join $R \bowtie S \bowtie T$. For each join considered, the optimizer will find and retain the best join path for each interesting order and the best "unordered" path. Sort orders on $R.a$, $S.a$, and $T.a$ are the candidate interesting orders for Q_1. When the optimizer is considering an n-way join, it only uses the best join paths retained for smaller joins.

6.1 Extended Enumeration

The first approach is to extend the existing path creation process that occurs during the enumeration of each possible join. The extended enumeration includes the path representing the union of the best child-join paths for the join. For instance, as part of the enumeration process for query Q_1, the optimizer will create and cost join paths for $S \bowtie T$. The conventional join paths include joining the union of S's partitions with the union of T's partitions using all applicable join operators (like hash join or merge join), leading to plans like Q_1P_1 in Figure 3. At this point, extended enumeration will also create join paths for $(S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ and $(S_{22} \cup S_{23}) \bowtie T_3$, find the corresponding best paths, and create the union of the best child-join paths. We will use the notation P_u in this section to denote the union of the best child-join paths.

As usual, the bottom-up optimizer will retain the best join path for each interesting order (a in this case) as well as the best (possibly unordered) overall path. If P_u is the best for one of these categories, then it will be retained. The paths retained will be the only paths considered later when the enumeration process moves on to larger joins. For example, when creating join paths for $(S \bowtie T) \bowtie R$, only the join paths retained for $S \bowtie T$ will be used (in addition to the access paths retained for R).

Property 1. Adding extended enumeration to a bottom-up optimizer will not always find the optimal plan in the extended plan space. □

We will prove Property 1 using our running example. Suppose plan P_u for $S \bowtie T$ is not retained because it is not a best path for any order. Without P_u for $S \bowtie T$, when the optimizer goes on to consider $(S \bowtie T) \bowtie R$, it will not be able to consider any 3-way child join. Thus, plans similar to Q_1P_2 from Figure 3 will never be considered; thereby losing the opportunity to find the optimal plan in the extended plan space.

6.2 Treating Partitions as a Physical Property

The next approach considers partitioning as a physical property of tables and the output of partition-wise joins. The parallel edition of DB2 follows this approach. The concept of *interesting partitions* (similar to interesting orders) can be used to incorporate partitioning as a physical property in the bottom-up optimizer. Interesting partitions are partitions on attributes referenced in equality join conditions and on grouping attributes [18]. In our example query Q_1, partitions on attributes $R.a$, $S.a$, and $T.a$ are interesting.

Paths with interesting partitions can make later joins and grouping operations less expensive when these operations can take advantage of the partitioning. For example, partitioning on $S.a$ for $S \bowtie T$ could lead to the creation of three-way child joins for $R \bowtie S \bowtie T$. Hence, the optimizer will retain the best path for each interesting partition, in addition to each interesting order. Overall, if there are n interesting orders and m interesting partitions, then the optimizer can retain up to $n \times m$ paths, one for each combination of interesting orders and interesting partitions.

Property 2. Treating partitioning as a physical property in a bottom-up optimizer will not always find the optimal plan in the extended plan space. □

Once again we will prove the above property using the example query Q_1. When the optimizer enumerates paths for $S \bowtie T$, it will consider P_u (the union of the best child-join paths). Unlike what happened in extended enumeration, P_u will now be retained since P_u has an interesting partition on $S.a$. Suppose the first and second child joins of P_u have the respective join paths $(S_{12} \cup S_{13})$ HJ $(T_1 \cup T_2)$ and $(S_{22} \cup S_{23})$ HJ T_3. (HJ and MJ denote hash and merge join operators respectively.) Also, the best join path for $S \bowtie T$ with an interesting order on $S.a$ is the union of the child-join paths $(S_{12} \cup S_{13})$ MJ $(T_1 \cup T_2)$ and $(S_{22} \cup S_{23})$ MJ T_3.

However, it can still be the case that the optimal plan for Q_1 is plan Q_1P_2 shown in Figure 3. Note that Q_1P_2 contains $(S_{12} \cup S_{13})$ MJ $(T_1 \cup T_2)$: the interesting order on $S.a$ in this child join led to a better overall plan. However, the interesting order on $S.a$ was not useful in the case of the second child join of $S \bowtie T$, so $(S_{22} \cup S_{23})$ MJ T_3 is not used in Q_1P_2. Simply adding interesting partitions alongside interesting orders to a bottom-up optimizer will not enable it to find the optimal plan Q_1P_2.

The optimizer was not able to generate plan Q_1P_2 in the above example because it did not consider interesting orders independently for each child join. Instead, the optimizer considered interesting orders and interesting partitions at the level of the parent tables (R, S, T) and joins of parent tables ($R \bowtie S$, $R \bowtie T$, $S \bowtie T$, $R \bowtie S \bowtie T$) only. An apparent solution would be for the optimizer to create union plans for all possible combinations of child-join paths with interesting orders. However, the number of such plans is exponential in the number of child joins per parent join, rendering this approach impractical.

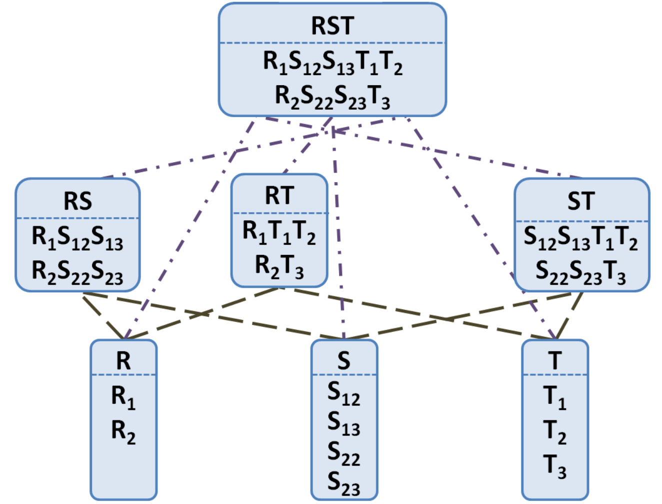

Figure 8: Logical relations (with child relations) enumerated for query Q_1 by our partition-aware bottom-up optimizer

6.3 Treating Partitions as a Logical Property

Our approach eliminates the aforementioned problems by treating partitioning as a property of the *logical relations* (tables or joins) that are enumerated during the bottom-up optimization process. A logical relation refers to the output produced by either accessing a table or joining multiple tables together. For example, the logical relation (join) RST represents the output produced when joining the tables R, S, and T, irrespective of the join order or the join operators used in the physical execution plan. Figure 8 shows all logical relations created during the enumeration process for our example query Q_1.

As illustrated in Figure 8, each logical relation maintains a list of logical child relations. A logical child table is created for each unpruned partition during partition pruning, whereas logical child joins are created based on the output of the clustering phase. For our example query Q_1, the child-join pairs $(\{S_{12}, S_{13}\}, \{T_1, T_2\})$ and $(\{S_{22}, S_{23}\}, \{T_3\})$ output by the clustering phase are used to create the respective logical child joins $S_{12}S_{13}T_1T_2$ and $S_{22}S_{23}T_3$. The logical child relations also maintain the partitioning conditions, which are propagated up when the child joins are created.

For each logical n-way join relation $J_n = J_{n-1} \bowtie J_1$, the logical child relations and partitioning conditions of J_{n-1} and J_1 form the input to J_n's matching phase (Section 4). The output of the matching phase forms the input to J_n's clustering phase (Section 5) which, in turn, outputs J_n's logical child joins. Note that both the matching and clustering phases work at the logical level, independent of physical plans (paths).

The logical relations are the entities for which the best paths found so far during the enumeration process are retained. The logical child joins behave in the same way as their parent joins, retaining the best paths for each interesting order and the best unordered path. Hence, the number of paths retained is *linear* in the number of child joins per parent join (instead of exponential as in the case when partitions are treated as physical properties). The optimizer considers all child-join paths with interesting orders during path creation for higher child joins, while ensuring the property:

Property 3. Paths with interesting orders for a single child join can be used later up the lattice, independent from all other child joins of the same parent relation. □

Suppose, the optimizer is considering joining ST with R to create paths for RST. The output of the clustering phase will produce the two child-join pairs $(S_{12}S_{13}T_1T_2, R_1)$ and $(S_{22}S_{23}T_3, R_2)$. Join paths for these two child joins will be created and costed independently from each other, using any paths with interesting orders and join operators that are available. The best join paths for $((S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)) \bowtie R_1$ and $((S_{22} \cup S_{23}) \bowtie T_3) \bowtie R_2)$ will be retained in the logical relations $R_1S_{12}S_{13}T_1T_2$ and $R_2S_{22}S_{23}T_3$ respectively (see Figure 8).

For each parent relation, the path representing the union of the best child-join paths is created only at the end of each enumeration level[3] and it is retained only if it is the best path. Hence, the optimizer will consider all join orders for each child join before creating the union, leading to the following property:

Property 4. The optimizer will consider plans where different child joins of the same parent relation can have different join orders and/or join operators. □

We have already seen how the optimizer created join paths $((S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)) \bowtie R_1$ and $((S_{22} \cup S_{23}) \bowtie T_3) \bowtie R_2$ when joining ST with R. Later, the optimizer will consider joining RS with T, creating join paths for $(R_1 \bowtie (S_{12} \cup S_{13})) \bowtie (T_1 \cup T_2)$ and $(R_2 \bowtie (S_{22} \cup S_{23})) \bowtie T_3$. It is possible that the best join path for $((S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)) \bowtie R_1$ is better than that for $(R_1 \bowtie (S_{12} \cup S_{13})) \bowtie (T_1 \cup T_2)$, while the opposite occurs between $((S_{22} \cup S_{23}) \bowtie T_3) \bowtie R_2$ and $(R_2 \bowtie (S_{22} \cup S_{23})) \bowtie T_3$; which leads to the plan Q_1P_2 in Figure 3.

Property 5. Optimality guarantee: By treating partitioning as a logical property, our bottom-up optimizer will find the optimal plan in the extended plan space. □

This property is a direct consequence of Properties 3 and 4. We have extended the plan space to include plans containing unions of child joins. Each child join is enumerated during the traditional bottom-up optimization process in the same way as its parent; the paths are built bottom-up, interesting orders are taken into consideration, and the best paths are retained. Since each child join is optimized independently, the topmost union of the best child-join paths is the optimal union of the child joins. Finally, recall that the union of the best child-join paths is created at the end of each enumeration level and retained only if it is the best plan for its parent join. Therefore, the full extended plan space is considered and the optimizer will be able to find the optimal plan (given the current database configuration, cost model, and physical design).

Traditionally, grouping (and aggregation) operators are added on top of the physical join trees produced by the bottom-up enumeration process [19]. In this case, interesting partitions are useful for pushing the grouping below the union of the child joins, in an attempt to create less expensive execution paths. With our approach, paths with interesting partitions on the grouping attributes can be constructed at the top node of the enumeration lattice, and used later on while considering the grouping operator.

Treating partitions as a property of the logical relations allows for a clean separation between the enumeration process of the logical relations and the construction of the physical plans. Hence, our algorithms are applicable to any database system that uses a bottom-up optimizer. Moreover, they can be adapted for non-database data processing systems like SCOPE and Hive that offer support for table partitioning and joins.

[3] Enumeration level n refers to the logical relations representing all possible n-way joins.

Name	Features
Basic	Per-table partition pruning only (like MySQL and PostgreSQL). Our evaluation uses the PostgreSQL 8.3.7 optimizer as the Basic optimizer
Intermediate	Per-table partition pruning and one-to-one partition-wise joins (like Oracle and SQLServer). The Intermediate optimizer is implemented as a variant of the Advanced optimizer that checks for and creates one-to-one partition-wise join pairs in place of the regular matching and clustering phases
Advanced	Per-table partition pruning and all the join optimizations for partitioned tables as described in the paper

Table 2: Optimizer categories considered in the evaluation

7. EXTENDING OUR TECHNIQUES TO PARALLEL DATABASE SYSTEMS

While this paper focuses on centralized DBMSs, our work is also useful in parallel DBMSs like Aster nCluster [3], Teradata [22], and HadoopDB [1] which try to partition tables such that most queries in the workload need intra-node processing only. A common data placement strategy in parallel DBMSs is to use hash partitioning to distribute tuples in a table among the nodes $N_1, \ldots, N_k$, and then use range/list partitioning of the tuples within each node. Our techniques extend to this setting: if two joining tables R and S have the same hash partitioning function and the same number of partitions, then a partition-wise join $R_i \bowtie S_i$ is created for each node N_i. If a secondary range/list partitioning has been used to further partition R_i and S_i at an individual node, then our techniques can be applied directly to produce child joins for $R_i \bowtie S_i$.

Another data placement strategy popular in data warehouses is to replicate the dimension tables on all nodes, while the fact table is partitioned across the nodes. The fact-table partition as well as the dimension tables may be further partitioned on each node, so our techniques can be used to create child joins at each node. In such settings, multi-dimensional partitioning of the fact table can improve query performance significantly as we show in Section 8.

8. EXPERIMENTAL EVALUATION

The purpose of this section is to evaluate the effectiveness and efficiency of our optimization techniques across a wide range of factors that affect table partitioning. We have prototyped all our techniques in the PostgreSQL 8.3.7 optimizer. All experiments were run on Amazon EC2 nodes of m1.large type. Each node has 7.5GB RAM, dual-core 2GHz CPU, and 850GB of storage. We used the TPC-H benchmark with scale factors ranging from 10 to 40, with 30 being the default scale. Following directions from the TPC-H Standard Specifications [23], we partitioned tables only on primary key, foreign key, and/or date columns. We present experimental results for a representative set of 10 out of the 22 TPC-H queries, ranging from 2-way up to the maximum possible 8-way joins. All results presented are averaged over three query executions.

8.1 Experimental Setup

The most important factor affecting query performance over partitioned tables is the partitioning scheme that determines which tables are partitioned and on which attribute. We identified two cases that arise in practice:

1. The DBA has full control in selecting and deploying the partitioning scheme to maximize query-processing efficiency.
2. The partitioning scheme is forced either partially or fully by practical reasons beyond query-processing efficiency.

For evaluation purposes, we categorized query optimizers into three categories—*Basic*, *Intermediate*, and *Advanced*—based on how they exploit partitioning information to perform optimization. Details

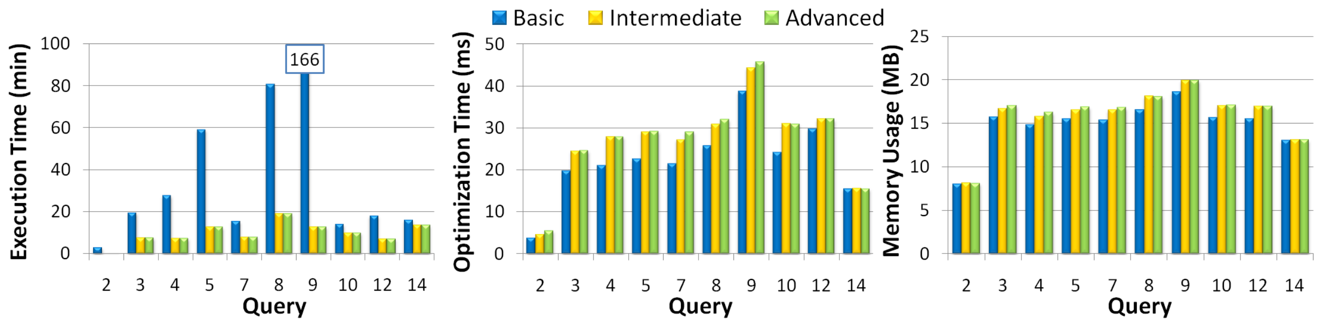

Figure 9: (a) Execution times, (b) Optimization times, (c) Memory usage for TPC-H queries over PS-J

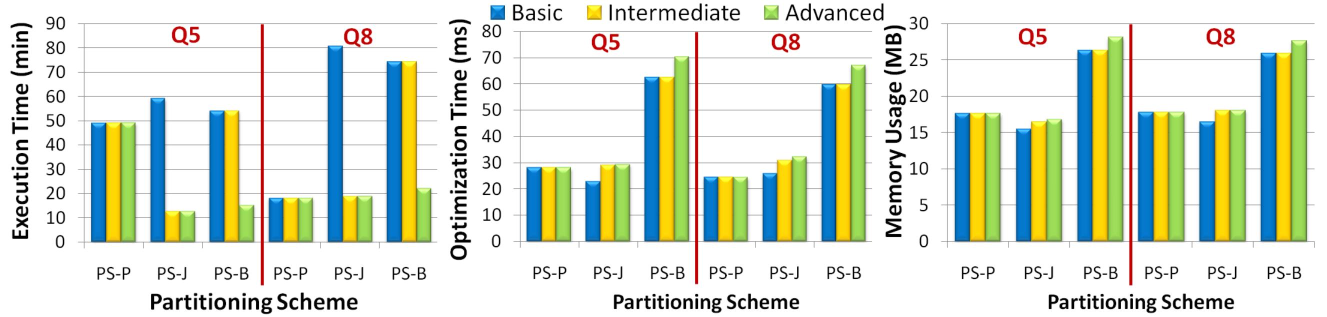

Figure 10: (a) Execution times, (b) Optimization times, (c) Memory usage for TPC-H queries 5 and 8 over three partitioning schemes

Partition Scheme	Table	Partitioning Attributes	Number of Partitions
PS-P	orders	o_orderdate	28
	lineitem	l_shipdate	85
PS-J	orders	o_orderkey	48
	lineitem	l_orderkey	48
	partsupp	ps_partkey	12
	part	p_partkey	12
PS-B	orders	o_orderkey, o_orderdate	72
	lineitem	l_orderkey, l_shipdate	120
	partsupp	ps_partkey	12
	part	p_partkey	6
PS-C	orders	o_orderkey, o_orderdate	36
	lineitem	l_orderkey, l_shipdate	168
	partsupp	ps_partkey	30
	part	p_partkey	6
	customer	c_custkey	6

Table 3: Partitioning schemes for TPC-H

are given in Table 2. We compare the optimizers on three metrics used to evaluate optimizers [12]: (i) query execution time, (ii) optimization time, and (iii) optimizer's memory usage.

8.2 Results from DBA-Controlled Schemes

Given the capabilities of the query optimizer, the DBA has a spectrum of choices regarding the partitioning scheme [26]. In one extreme, the DBA can partition tables based on attributes appearing in filter conditions in order to take maximum advantage of partition pruning. At the other extreme, the DBA can partition tables based on joining attributes in order to take maximum advantage of one-to-one partition-wise joins; assuming the optimizer supports such joins (like the Intermediate optimizer in Table 2). In addition, our techniques now enable the creation of multidimensional partitions to take advantage of both partition pruning and partition-wise joins. We will refer to the three above schemes as partitioning schemes respectively for pruning (PS-P), for joins (PS-J), and for both (PS-B). Table 3 lists all partitioning schemes used in our evaluation.

Figure 9(a) shows the execution times for the plans selected by the three query optimizers for the ten TPC-H queries running on the database with the PS-J scheme. The Intermediate and Advanced optimizers are able to generate a better plan than the Basic optimizer for all queries, providing up to an order of magnitude benefit for some of them. Note that the Intermediate and Advanced optimizers produce the same plan in all cases, since one-to-one partition-wise joins are the only join optimization option for both optimizers for the PS-J scheme.

Figure 9(b) presents the corresponding optimization times for the queries. The Intermediate optimizer introduces some overhead on optimization time—average of 17% and worst case of 21% due to the creation of child-join paths—compared to Basic. The additional overhead introduced by the Advanced optimizer over Intermediate is on average less than 3%. This overhead is due to the matching and clustering algorithms. Overall, the optimization overhead introduced by Advanced is low, and is most definitely gained back during execution as we can see by comparing the y-axes of Figures 9(a) and 9(b) (execution time is in minutes whereas optimization time is in milliseconds). The memory overheads shown in Figure 9(c) follow the same trend: average memory overhead of Advanced over Basic is around 7%, and the worst case is 10%.

Query performance is related directly to the optimizer capabilities and the partitioning scheme used in the database. Figure 10 shows the performance results for TPC-H queries 5 and 8 for the three optimizers over databases with different partitioning schemes. (Results for other queries are similar.) Since a database using the PS-P scheme only allows for partition pruning, all three optimizers behave in an identical manner. A PS-J scheme on the other hand, does not allow for any partition pruning since join attributes do not appear in filter conditions in the queries. Hence, the Basic optimizer performs poorly in many cases, whereas the Intermediate and Advanced optimizers take advantage of partition-wise joins to produce better plans with very low overhead.

The presence of multidimensional partitions in a PS-B scheme prevents the Intermediate optimizer from generating any one-to-one partition-wise joins, but it can still perform partition pruning

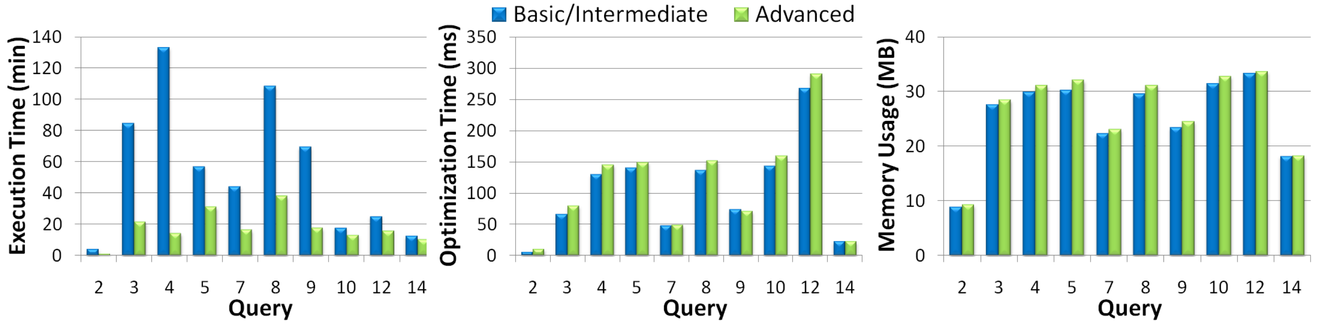

Figure 11: (a) Execution times, (b) Optimization times, (c) Memory usage for TPC-H queries over PS-C with partition size 128MB

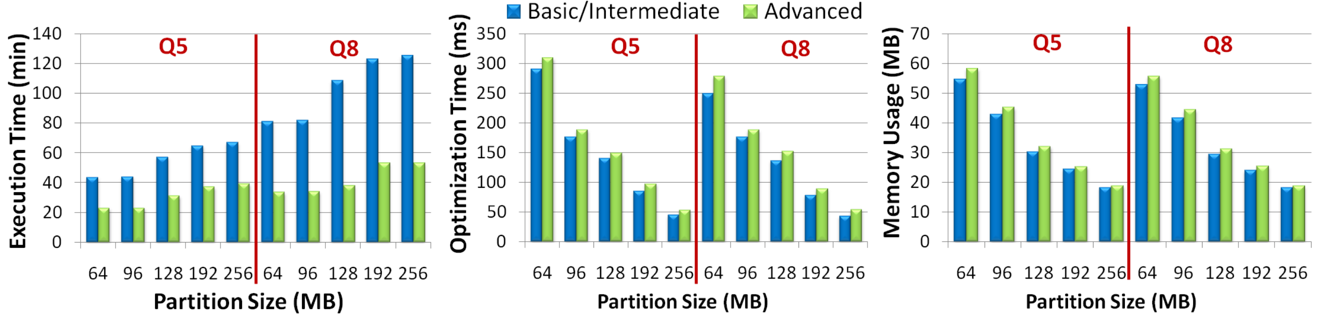

Figure 12: (a) Execution times, (b) Optimization times, (c) Memory usage as we vary the partition size for TPC-H queries 5 and 8

like the Basic optimizer. The Advanced optimizer utilizes both partition pruning and partition-wise joins to find better-performing plans. Consider the problem of picking the best partitioning scheme for a given query workload. The best query performance can be obtained either from (a) partition pruning (PS-P is best for query 8 in Figure 10), or (b) from partition-aware join processing (PS-J is best for query 5 in Figure 10), or (c) from a combination of both due to some workload or data properties. In all cases, the Advanced optimizer enables finding the plan with the best possible performance.

8.3 Results from Constrained Schemes

As discussed in Section 1, external constraints or objectives may limit the partitioning scheme that can be used. For instance, data arrival rates may require the creation of daily or weekly partitions; file-system properties may impose a maximum partition size to ensure that each partition is laid out contiguously; or optimizer limitations may impose a maximum number of partitions per table.

For a TPC-H scale factor of 30, biweekly partitions of the fact table lead to a 128MB partition size. We will impose a maximum partition size of 128MB to create the partitioning scheme PS-C used in this section (see Table 3). Figure 11 shows the results for the TPC-H queries executed over a database with the PS-C scheme. The constraint imposed on the partitioning scheme does not allow for any one-to-one partition-wise joins. Hence, the Intermediate optimizer produces the same plans as Basic, and is excluded from the figures for clarity. Once again, the Advanced optimizer was able to generate a better plan than the Basic optimizer for all queries, providing over $2x$ speedup for 50% of them. The average optimization time and memory overheads were just 7.9% and 3.6% respectively.

8.4 Effect of Size and Number of Partitions

In this section, we evaluate the performance of the optimizers as we vary the size (and thus the number) of partitions created for each table, using the PS-C scheme. As we vary the partition size from 64MB to 256MB, the number of partitions for the fact table vary from 336 to 84. Figure 12(b) shows the optimization times taken by the two optimizers for TPC-H queries 5 and 8. As the partition size increases (and the number of partitions decreases), the optimization time decreases for both optimizers. We observe that (i) the optimization times for the Advanced optimizer scale in a similar way as for the Basic optimizer, and (ii) the overhead introduced by the creation of the partition-wise joins remains small (around 12%) in all cases.

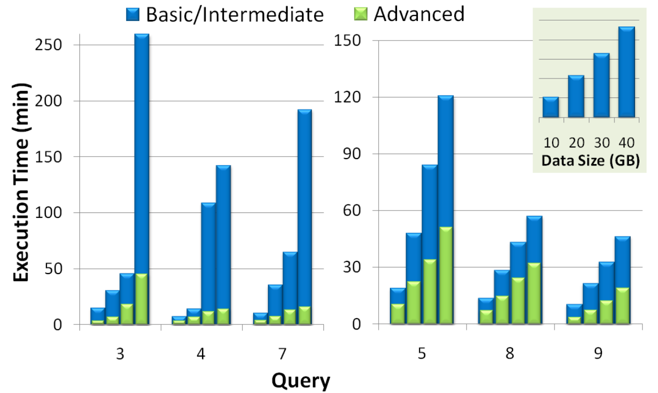

Figure 13: Execution times as we vary the total data size

The overhead added by our approach remains low due to two reasons. First, Clustering bounds the number of child joins for $R \bowtie S$ to *min(number of partitions in R,S)*; so we cause only a linear increase in paths enumerated per join. Second, optimizers have other overheads like parsing, rewrites, scan path enumeration, catalog and statistics access, and cardinality estimation. Let us consider Query 5 from Figure 12(b). Query 5 joins 5 tables, including *orders* and *lineitem* with 72 and 336 partitions respectively. In this case, Basic enumerated 2317 scan and join paths in total, while Advanced enumerated 2716 paths. The extra 17% paths are for the 72 partition-wise joins created by Advanced. The trends are similar for the memory consumption of the optimizers as seen in Figure 12(c).

Decreasing the partition size for the same total data size has a positive effect on plan execution times as seen in Figure 12: smaller

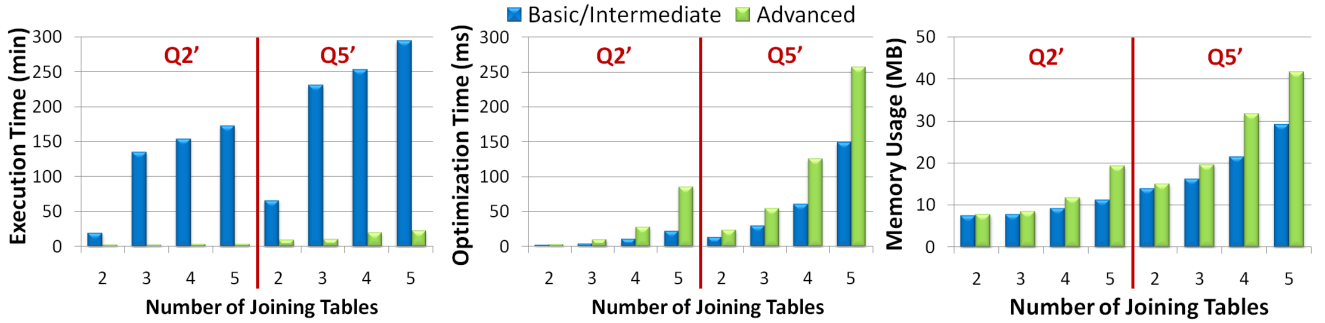

Figure 14: (a) Execution times, (b) Optimization times, (c) Memory usage as we vary the number of tables joined on the same attribute for a modified TPC-H schema and queries 2 and 5

Figure 15: (a) Execution times, (b) Optimization times, (c) Memory usage for enabling and disabling clustering

partition sizes force finer-grained partition ranges, leading to better partition pruning and join execution. Looking into execution times at the subplan level, we observed that PostgreSQL was more effective in our experimental settings when it accessed partitions in the 64MB range. It is worth noting that current partitioning scheme recommenders [2, 18, 26] do not consider partition size tuning.

8.5 Effect of Data Size

We used the PS-C scheme with a partition size of 128 MB to study the effects of the overall data size on query performance. Figure 13 shows the query execution times as the amount of data stored in the database increases. For many queries, the plans selected by the Basic optimizer lead to a quadratic or exponential increase in execution time as data size increases linearly. We observed that joins for large data sizes cause the Basic optimizer to frequently resort to index nested loop joins (the system has 7.5GB RAM only).

On the other hand, the Advanced optimizer is able to generate smaller partition-wise joins that use more efficient join methods (like hash and merge joins); leading to the desired linear increase in execution time as data size increases linearly. For the queries where the Basic optimizer is also able to achieve a linear trend, the slope is much higher compared to the Advanced optimizer. Figure 13 shows that the benefits from our approach become more important for larger databases. Note that optimization times and memory consumption are independent of the data size.

8.6 Stress Testing on a Synthetic Benchmark

There exist practical scenarios where multiple tables may share a common joining key. In Web analytics, for example, customer data may reside in multiple tables—storing information such as page clicks, favorites, preferences, etc.—that have to be joined on the customer key. However, in traditional star and snowflake schemas, the fact tables join with the dimension tables on different attributes; so it is hard to create n-way child joins for $n \geq 3$. No TPC-H query plan, regardless of the partitioning schema, contains n-way child joins for $n \geq 4$. To evaluate our approach in non-star schemas, as well as to stress-test our approach, we came up with a synthetic partitioning schema where the tables *part* and *orders* from TPC-H are partitioned vertically into four tables each. Once again, we use the PS-C scheme with a partition size of 128 MB.

We modified TPC-H queries 2 and 5 to join all the vertical tables for *part* and *orders* respectively. Figure 14(a) shows the execution times for the two queries with increasing number of joining tables. We observe how the Advanced optimizer was again 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 joining tables in the query increases, the execution times for the plans from the Advanced optimizer increase barely (due to efficient use of child joins); unlike the Basic optimizer's plans whose execution times increase drastically.

Figures 14(b) and 14(c) show the optimization times and memory consumption, respectively, as the number of "same-key-joining" tables in the query increases. Both metrics increase non-linearly for both optimizers; but the increase is more profound for the Advanced optimizer. The increasing optimization overhead comes from the non-linear complexity of the path selection process used by the regular PostgreSQL query optimizer (which we believe can be fixed through engineering effort unrelated to our work). With optimization times still in milliseconds, the additional overhead is certainly justified by the drastic reduction in execution times.

8.7 Effect of the Clustering Algorithm

Clustering (Section 5) is an essential phase in our overall partition-aware optimization approach that is missing from the data localization approach discussed in Section 2. When matching is applied without clustering, our optimizer implements a rough equivalent of the four-phase approach to distributed query optimization [16]. Figures 15(b) and 15(c) compare the optimization time and memory consumption of the optimizer when clustering is enabled and disabled in a database with the PS-C scheme. Disabling clustering causes high overhead—as seen in both figures—since the optimizer must now generate join paths for each child join produced by the matching phase. This issue shows why clustering is essential for

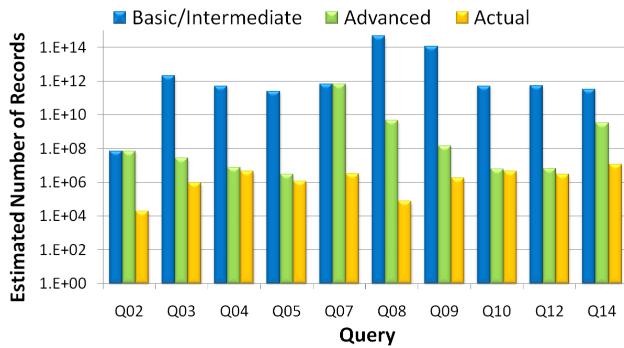

Figure 16: Estimated (and actual) number of records of TPC-H queries over PS-C

our optimizer to perform well in the presence of hundreds of partitions per table (e.g., daily partitions for a year).

Figure 15(a) shows the execution times for the plans generated when enabling and disabling clustering. In all cases shown, the plan generated without clustering is worse than the plan generated when clustering is used, since the generated plans scan the same partitions multiple times (in different joins). The queries that are not shown failed to complete because the system runs out of memory during plan execution. Note that with multidimensional partitioning and without clustering, literally thousands of child join paths are created, each requiring a small amount of memory during their initialization phase. We conclude that the use of clustering is crucial for finding good execution plans.

8.8 Impact on Cardinality Estimation

An additional benefit that child joins bring is better cardinality estimation for costing during path creation. 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 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 cardinality estimates needed are derived by aggregating statistics over partitions.

Figure 16 shows the estimated and actual number of records of TPC-H queries over the PS-C scheme. For the Basic Optimizer, we observe large cardinality errors. In contrast, partition-wise joins provide much more accurate cardinality estimation because these joins increase the chances of using partition-level statistics directly for costing. The same pattern was observed with all the partitioning schemes and queries used.

9. SUMMARY

Query optimization technology has not kept pace with the growing usage and user control of table partitioning. We addressed this gap by developing novel partition-aware optimization techniques to generate efficient plans for SQL queries over partitioned tables. We extended the search space to include plans with multiway partition-wise joins, and provided techniques to find the optimal plan efficiently. Our techniques are designed for easy incorporation into bottom-up query optimizers. An extensive experimental evaluation showed that our optimizer, with low optimization-time overhead, can generate plans that are an order of magnitude better than plans generated by current optimizers.

10. ACKNOWLEDGMENTS

We thank John Cieslewicz and Eric Friedman of Aster Data for introducing us to the problem and for many valuable discussions. We thank the anonymous reviewers and shepherd for helping us improve the paper in many ways.

11. 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 *VLDB*, 2009.

[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *ACM SIGMOD*, 2004.

[3] *Aster nCluster*. www.asterdata.com/product/ncluster_cloud.php.

[4] C. Baldwin, T. Eliassi-Rad, G. Abdulla, and T. Critchlow. The Evolution of a Hierarchical Partitioning Algorithm for Large-Scale Scientific Data: Three Steps of Increasing Complexity. *SSDB*, 2003.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of the 5th ACM Symp. on Principles of Database Systems*, 1986.

[6] C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. DB2 Parallel Edition. *IBM Systems Journal*, 34(2), 1995.

[7] P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based Routing: Different Plans for Different Data. In *VLDB*, 2005.

[8] S. Ceri and G. Gottlob. Optimizing Joins Between two Partitioned Relations in Distributed Databases. *Journal of Parallel and Distributed Computing*, 3, 1986.

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

[10] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting Correlated Attributes in Acquisitional Query Processing. In *ICDE*, 2005.

[11] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-Describing, Polymorphic, and Parallelizable User-Defined Functions. In *VLDB*, 2009.

[12] L. Giakoumakis and C. Galindo-Legaria. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. In *IEEE Data Engineering Bulletin*. IEEE Computer Society, 2008.

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

[14] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query Optimization by Predicate Move-Around. In *VLDB*, 1994.

[15] 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.

[16] T. M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[17] N. Polyzotis. Selectivity-based Partitioning: A Divide-and-union Paradigm for Effective Query Optimization. In *CIKM*, 2005.

[18] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating Physical Database Design in a Parallel Database. In *SIGMOD*, 2002.

[19] 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 *ACM SIGMOD*, 1979.

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

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

[22] *Teradata*. http://www.teradata.com.

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

[24] B. Zeller and A. Kemper. Experience Report: Exploiting Advanced Database Optimization Features for Large-Scale SAP R/3 Installations. In *VLDB*, 2002.

[25] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, 2010.

[26] D. Zilio, A. Jhingran, and S. Padmanabhan. Partitioning Key Selection for a Shared-Nothing Parallel Database System. *IBM Research Report RC 19820*, 1994.

[27] C. Zuzarte, others Partitioning in DB2 Using the UNION ALL View. *IBM developerWorks*, 2002. http://www.ibm.com/developerworks/data/library/techarticle/0202zuzarte/0202zuzarte.pdf.