Automated Partitioning Design in Parallel Database Systems

Rimma Nehme
Microsoft Jim Gray Systems Lab
Madison, WI 53703
rimman@microsoft.com

Nicolas Bruno
Microsoft
Redmond, WA 98052 USA
nicolasb@microsoft.com

ABSTRACT

In recent years, Massively Parallel Processors (MPPs) have gained ground enabling vast amounts of data processing. In such environments, data is partitioned across multiple compute nodes, which results in dramatic performance improvements during parallel query execution. To evaluate certain relational operators in a query correctly, data sometimes needs to be re-partitioned (i.e., moved) across compute nodes. Since data movement operations are much more expensive than relational operations, it is crucial to design a suitable data partitioning strategy that minimizes the cost of such expensive data transfers. A good partitioning strategy strongly depends on how the parallel system would be used. In this paper we present a partitioning advisor that recommends the best partitioning design for an expected workload. Our tool recommends which tables should be replicated (i.e., copied into every compute node) and which ones should be distributed according to specific column(s) so that the cost of evaluating similar workloads is minimized. In contrast to previous work, our techniques are deeply integrated with the underlying parallel query optimizer, which results in more accurate recommendations in a shorter amount of time. Our experimental evaluation using a real MPP system, Microsoft SQL Server 2008 Parallel Data Warehouse, with both real and synthetic workloads shows the effectiveness of the proposed techniques and the importance of deep integration of the partitioning advisor with the underlying query optimizer.

Categories and Subject Descriptors

H.2 [**Database Management**]: Query Processing
; H.2.2 [**Physical Design**]: Access Methods

General Terms

Algorithms

Keywords

distributed databases

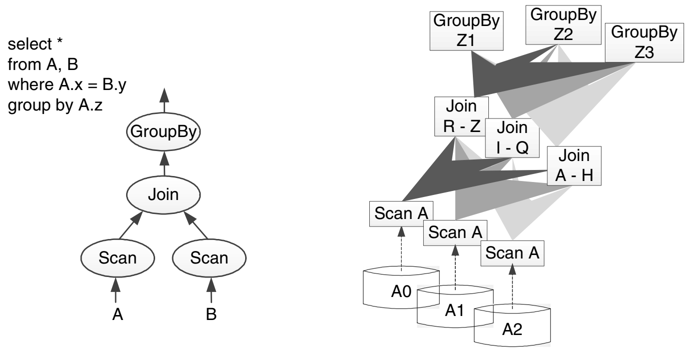


Figure 1: Sequential vs. parallel query execution.

1. INTRODUCTION

High-performance computing has undergone many changes in recent years. One of the major trends has been the wide adoption of massively parallel processing (MPP) systems. An MPP system is a distributed computer system which consists of many individual nodes, each of which is essentially an independent computer in itself. Each node, in turn, consists of at least one processor, its own memory, and a link to the network that connects all nodes together. Queries executed in such environments tend to be complex, involving many joins, nested sub-queries and aggregation and are usually long-running and resource-intensive (see Figure 1). During query execution, data often needs to be transferred across nodes, which is a relatively expensive operation compared to relational operators. Excessive data transfers can significantly slow down query execution, deteriorate the performance of the overall system, and negatively impact the user experience. If data is originally partitioned in an adequate way, such expensive data transfer operations can be minimized. Clearly, the selection of the best way to partition the data in a distributed environment is a critical physical database design problem.

Previous studies on automated partitioning design can be roughly classified into two categories: optimizer-independent (which interpret and model the optimization information outside of the database to perform the tuning [29, 31]), and "shallowly-integrated" with query optimizer (which largely use the optimizer as a black-box to perform the *what-if* optimization calls [25]). The problem with such loosely-integrated approaches is two-fold: first, the quality of the resulting partitioning recommendations is likely to suffer when the tuning tools are not completely in-sync with optimizer's decisions, and second, the performance of a tuning tool is likely to diminish due to narrow APIs between the tool and the DBMS.

We propose to address the problem of partition design tuning for parallel databases by introducing an advisor that is *deeply-integrated* with parallel query optimizer. Specifically, our partitioning advisor exploits the optimizer's cost model as well as its internal data structure, called the MEMO, to efficiently find the best possible partitioning configuration. Furthermore, we leverage the MEMO structure to infer lower bounds on partial partitioning configurations – the property that enables a very efficient branch and bound search strategy through the combinatorial search space of all feasible partition configurations.

In addition to using the partitioning advisor when loading a brand-new database into a distributed environment, a new partitioning strategy may also be recommended whenever:

- data is migrated to a new system.
- the workload on a database changes substantially.
- the database has been heavily updated (e.g., tables are added or removed, or statistical information changes).
- performance has significantly degraded.

We've implemented our partitioning advisor in Microsoft SQL Server 2008 Parallel Data Warehouse [4], a real MPP system[1], which is a scalable distributed engine that delivers performance at low cost through massive parallel processing. In summary, the primary contributions of our work include:

- We exploit the concept of a "*shell appliance*" to simulate a physically distributed database with various partitioning configurations stored on a single machine as if it were a regular database (but with *no* actual data).
- We leverage the MPP optimizer's concise optimization search space (the MEMO) to infer lower bounds on partial configurations and exploit this property for a better traversal of the partitioning configuration search space.
- We evaluate our partitioning advisor against the shallow integration approaches like genetic- and rank-based techniques from [25] to enumerate the search space.
- We implemented our partitioning advisor in a real MPP system and present our experimental evaluation using the TPC-H, TPC-DS benchmarks and several real-life datasets.

The rest of the paper is structured as follows. Section 2 provides the background necessary to better understand the technical aspects of this work. Section 3 gives the problem definition and states our main assumptions. Section 4 reviews shallowly-integrated approaches for finding the best partitioning configuration. Section 5 describes the details of our proposed deeply-integrated approach to partition design tuning. Section 6 reports our experimental evaluation, and Section 7 discusses possible extensions to the advisor. Section 8 reviews the related work.

2. BACKGROUND

In this section we provide the background information necessary to understand the remainder of the paper.

2.1 Overview of PDW

Many enterprises today use MPP architectures and vendors are focusing on making them faster and more scalable [1, 2, 3, 4, 5]. Shared-nothing parallel database systems are an example of such MPP systems. An installation of a shared-nothing parallel database

[1] For brevity, we will refer to it as PDW in the rest of the paper.

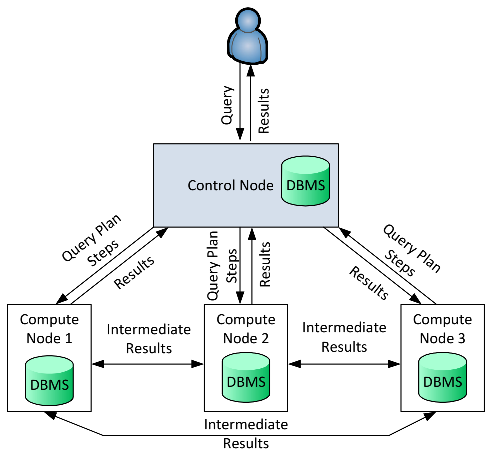


Figure 2: Overview of a 4 node appliance.

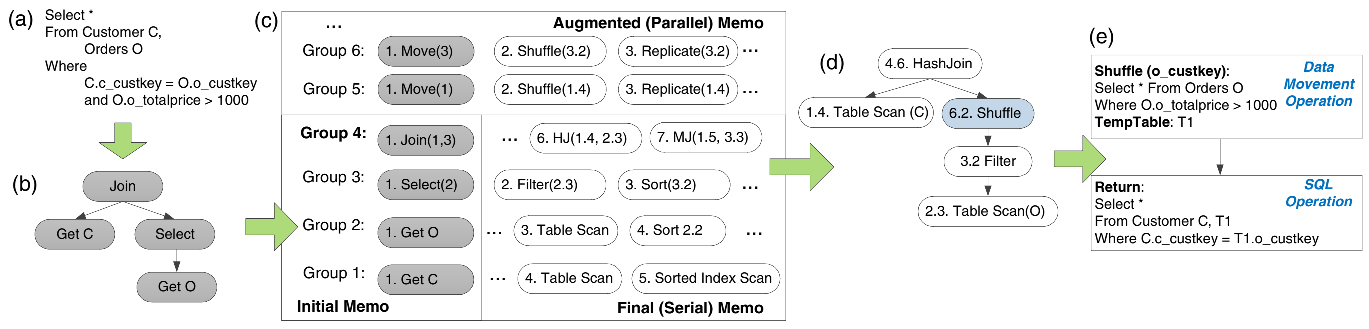
system like PDW is often called an *appliance*. An appliance consists of a single *control node* that manages one or more *compute nodes*. Figure 2 illustrates an appliance with three compute nodes and one control node. The key characteristics of a PDW appliance that we consider in this paper include:

- Running DBMS instances on the control and compute nodes to manage data locally.
- Leveraging DBMS functionality for authentication, authorization, query parsing and validation, schema management and locking instead of duplicating it in the middleware.
- Using the "shell appliance" abstraction of a distributed database to simulate it on a single machine as if it were a standard (non-distributed) database.
- Employing a statistics-driven, cost-based parallel query optimizer exploiting the shell appliance for optimization of parallel queries (an approach similar in spirit to *what-if* optimization technology [13]).

We describe the role of the control and compute nodes in an appliance architecture next.

2.2 Compute and Control Nodes in an Appliance

The Control Node: The control node provides the external interface to the appliance. All user interaction with the appliance flows through the control node, which is responsible for query parsing, creating a parallel execution plan for evaluating the query, distributing the plan steps to the compute nodes, tracking the execution steps of the plan, and assembling the individual "pieces" of the final results and packaging them up into the single result set that is returned to the user. As such, the control node masks the distributed nature of the system and presents users with a single system image of the appliance. This is a powerful abstraction, because it allows users to treat the appliance as if it were a standard, single-box DBMS. The control node has a DBMS instance running on it, but the only data stored permanently in this instance is the system metadata (no user data is stored permanently on the control node).
The Compute Nodes: Compute nodes provide the data storage and the query processing backbone of the appliance. Each compute node also has a DBMS instance running on it, and all permanent user data is stored there. All user queries, therefore, must access data stored in DBMS on some (or all) of the compute nodes.



(a)
Select *
From Customer C,
Orders O
Where
C.c_custkey = O.o_custkey
and O.o_totalprice > 1000
(b)
Join
Get C
Select
Get O
(c)
Augmented (Parallel) Memo
Group 6: 1. Move(3) 2. Shuffle(3.2) 3. Replicate(3.2)
Group 5: 1. Move(1) 2. Shuffle(1.4) 3. Replicate(1.4)
Group 4: 1. Join(1,3) 6. HJ(1.4, 2.3) 7. MJ(1.5, 3.3)
Group 3: 1. Select(2) 2. Filter(2.3) 3. Sort(3.2)
Group 2: 1. Get O 3. Table Scan 4. Sort 2.2
Group 1: 1. Get C 4. Table Scan 5. Sorted Index Scan
Initial Memo
Final (Serial) Memo
(d)
4.6. HashJoin
1.4. Table Scan (C)
6.2. Shuffle
3.2 Filter
2.3. Table Scan(O)
(e)
Shuffle (o_custkey):
Select * From Orders O
Where O.o_totalprice > 1000
TempTable: T1
Data Movement Operation
Return:
Select *
From Customer C, T1
Where C.c_custkey = T1.o_custkey
SQL Operation

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

2.3 Plan Generation and Execution

The parallel query optimizer in PDW employs the following query optimization approach (graphically depicted in Figure 4).

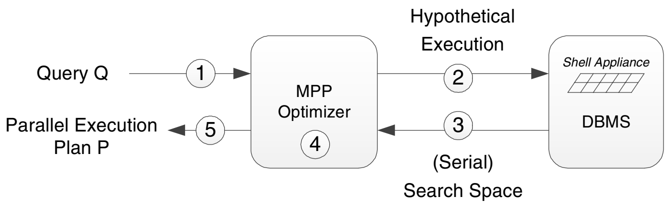


Figure 4: Parallel query optimization flow (all on the control node).

The DBMS instance on the control node contains the shell version of the actual physical appliance (i.e., an exact copy of all information including schemas, statistics, authorizations, but excluding the actual data tuples). Queries are parsed and validated semantically against the schemas and access privileges, and get optimized in a cost-based manner against the shell appliance.

For a given input query (Step 1), the MPP optimizer first issues a single optimization call against the shell appliance stored in the control node's DBMS (Step 2), and obtains back a compact representation of the *serial optimization search space* known as the MEMO [18, 19] (Step 3). Then, the parallel optimizer augments the extracted MEMO with the relevant partitioning information and the data movement operations and finds the best parallel query plan based on the cost-model tuned to the characteristics of the PDW architecture and the latest statistics (Step 4). Note that this approach generalizes the techniques that parallelize the best serial plan, by attempting the parallel versions of every serial plan considered by the optimizer and returned in the serial MEMO. Also note, that most of the work is done in the control node's DBMS (e.g., serial plan generation); the post-processing that parallelizes the MEMO is a relatively much smaller task. The main steps of MEMO parallelization consist of: (1) parallelization-specific MEMO "cleaning" and simplification; (2) interesting column propagation; (3) MEMO groups' optimization (enumeration, costing, pruning) done in a bottom-up manner; and finally, (4) best plan extraction. We omit the details of the parallel optimization, since it is outside the scope of this paper.

Parallel Query Execution: To execute a query, the control node transforms the query plan produced by the optimizer into a parallel execution plan (called *DSQL plan*) that consists of a sequence of steps (called *DSQL Operations*). At a high-level, every DSQL plan is composed of two types of DSQL operations: (1) the *SQL operations*, and (2) the *Data movement operations*. A SQL operation is simply an SQL statement to be executed against the underlying compute node's DBMS instance, and data movement operations are used to transfer data between DBMS instances on different nodes. A data movement operation specifies the source data to be moved, the strategy to use for distributing records among nodes (e.g., by replication or by hash-partitioning, etc.[2]), the name and the schema of the table to insert the records into upon arrival.

Example: Consider the following SQL query:

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

Figure 3 visually depicts the flow of parallel query optimization for the above query. We first parse the input query (Figure 3(a)) and transform it into a tree of logical operators (Figure 3(b)). Traditional query optimization is performed producing a final serial MEMO, which is augmented to additionally consider parallelism (see Figure 3(c)). A MEMO consists of two mutually recursive data structures, called *groups* and *groupExpressions*. A group represents all equivalent operator trees producing the same output. To reduce memory requirements, a group does not explicitly enumerate all its operator trees. Instead, it implicitly represents all the operator trees by using *groupExpressions*. A *groupExpression* is an operator having other groups (rather than other operators) as children. As an example, consider Figure 3(c), which shows a MEMO for the query example above (logical operators are shaded and physical operators have white background). In the figure, group 1 represents all equivalent expressions that return the contents of table *Customer* (or *C*, for short). Some operators in group 1 are logical (e.g., *Get C*), and some are physical (e.g., *Table Scan*, which reads the contents of *C* from the primary index or heap, and *Sorted Index Scan*, which does it from an existing secondary index). In turn, group 4 contains all the equivalent expressions for $C \bowtie O$. Note that *groupExpression* 4.1 (i.e., *Join*(1,3)), represents all operator trees whose root is *Join*, first child belongs to group 1, and second child belongs to group 3. In this way, a MEMO compactly represents a potentially very large number of operator trees. Children of physical *groupExpressions* point to the most efficient *groupExpression* in the corresponding groups. For instance, *groupExpression* 4.6 represents a hash join operator whose left-hand-child is the fourth *groupExpression* in group 1 and whose right-hand-child is the third *groupExpression* in group 2. In addition to enabling memoization (a variant of dynamic programming), a MEMO provides duplicate detection of operator trees, cost management, and other supporting infrastructure needed during query optimization. Additional details on the organization of the MEMO structure can be found in the literature [18, 19].

[2]For example, the data movement operation that re-partitions data across the compute nodes is called a *shuffle* operation in PDW.

Once the serial memo has been generated, the MPP optimizer augments the final serial MEMO with new data movement groups and operations with respect to the underlying data distributions, e.g., see groups 5 and 6 in the figure. Group 5, for instance, represents the data movement of the output of group 1 (i.e., the tuples from C). Assuming, C and O are distribution-*incompatible*, this operation would be considered by the parallel optimizer as one of the options in order to make both C and O distribution-compatible to perform the join $C \bowtie O$. Alternatively, group 6 depicts the data movement of the opposite input to the join, namely O. Similar, to relational operations, logical data movement operations may have a number of physical implementations, such as *Shuffle* (or re-distribution of data on a column(s)), *Replication*, and so on. The final execution plan, which consists of a tree of physical operators, is extracted from the MEMO (shown in Figure 3(d)). Finally, the plan is transformed into an executable DSQL plan that will be run in the appliance (shown in Figure 3(e)).

After generating DSQL plan, the control node distributes the plan to the compute nodes, one DSQL operation at a time. Contrary to the control node, whose sole responsibility is to execute DSQL operations, a compute node does not have knowledge of either the original user query or of the full execution plan for evaluating that query; it simply executes DSQL operations requested from the control node.

3. AUTOMATED PARTITIONING DESIGN PROBLEM

Although each MPP system may have its own unique features and characteristics, the problem of automated partitioning design in a parallel DBMS environment is common to all of them and can be generally stated as follows:

Given a database D, a query workload W, and a storage bound B, find a partitioning strategy (or configuration) for D such that (i) the size of replicated tables fits in B, and (ii) the overall cost of W is minimized.

Next we describe some simplifying assumptions that we consider in the rest of the paper. These assumptions are based on the current capabilities of the MPP engine we consider, and do not limit the applicability of our approach in general. First, we assume that tables can either be replicated or partitioned on the nodes in the appliance. When a table is partitioned, it is hash-partitioned on a single column across all compute nodes in an appliance (in Section 7 we describe how to relax these assumptions). Extending the partitioning approach to a subset of appliance nodes could be done similar to [25]. Finally, we assume that database statistics for cost estimation are always available. Statistics can be collected when data has already been loaded in the system, or alternatively, the DBA can supply statistics by generating them on the external data using one of the available tools provided by most database vendors.

4. TUNING WITH SHALLOW OPTIMIZER INTEGRATION

Figure 5 shows a generic and high-level architecture of the partitioning advisor interacting with the parallel query optimizer[3]. It consists of three key components: (1) the complex search space of all feasible partition configurations (depicted by the green cloud in the Figure), (2) the search algorithm to navigate the space of all possible partition configurations, and (3) the evaluation mechanism

[3]We want to note that the partitioning advisor is a separate client tool from the optimizer irrespective of how it is integrated with the optimizer.

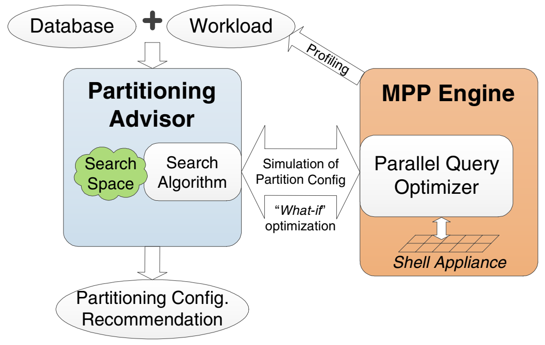


Figure 5: Partitioning advisor architecture.

to quantitatively compare the enumerated partitioning configurations. The complex search space needs to be traversed and for efficiency purposes a typical approach is to use a *what-if* optimization mode [13], which simulates hypothetical partition configurations for the appliance in the shell database without actually materializing them. This architecture is possible, because the query optimizer does not require the presence of fully materialized physical design structures in order to be able to generate plans that consider them. To search for the best partitioning configuration, we can apply various strategies from the field of combinatorial optimization, such as genetic search, simulated annealing, hill-climbing search, etc., or apply more tailored to the problem search algorithm, which is what we adopt in our work.

Current state-of-the-art physical design tuners can be described as *shallowly-integrated* with their underlying query optimizers. These tuners typically focus on speeding up the generation of feasible partitioning configurations (i.e., the search space), but afterwards only use the optimizer to perform the *what-if* optimization largely treating the optimizer as a black box. We next discuss two approaches that exhibit this shallow integration behavior with the underlying query optimizer in the context of parallel database tuning.

4.1 Rank-Based Algorithm

Rao et al. introduced a shallowly-integrated approach for partitioning tuning design in [25]. The idea is to first generate candidate configurations by invoking the optimizer to evaluate all workload statements in a special "*recommend*" mode. In this mode, the optimizer accumulates a list of partitions for each table that are potentially beneficial to processing of a given query and generates plans corresponding to each of these partitions. Optimization then proceeds normally to evaluate all of the alternative plans. Once the optimizer finds a plan that it considers optimal for the query, it extracts the partition of each base table and adds it to the *candidate partition* set. Subsequently, the advisor performs partition expansion to generate additional candidate partitions that might have been missed by each individual statement. Finally, the advisor combines candidate partitions from different tables and evaluates the workload in the regular *what-if* mode for each combination, returning at the end the best configuration for the entire workload.

For a search strategy, the technique uses a rank-based enumeration search, where partitions are first ranked based on their *benefit* value, which is approximated as the sum of the differences between the estimated cost of the query evaluated in the regular mode versus in the recommend mode. Then all configurations are ranked and organized into an ordered queue. The ranking function (referred to as *rank_best*) assigns the cost of a configuration to be the cost of its parent[4] minus the benefit of the changed partition, weighted by the relative table size. Since a partitioning configuration can be derived from multiple parents, the technique tries to always pick the higher rank for the configuration. The enumeration process evaluates configurations in the order of their benefit rank until a stopping condition is reached.

4.2 Genetic Algorithm

Another example of a shallowly-integrated parallel database tuning approach uses genetic algorithm (GA), a traditional combinatorial optimization technique. GA has several advantages: (1) it combines the elements of directed and stochastic search; (2) it tends to balance the two search objectives: exploiting the best solution(s) and exploring the search space; (3) it usually maintains a pool (population) of potential solutions while almost all other methods process only a single point in the search space. This property has two important implications: first, it efficiently "parallelizes" the search; and second, GA provides not only the best solution, but also a pool of good solutions that shed light on the effects of available alternatives. The major aspects of GA applied to partitioning design problem are described next, and Figure 6 depicts GA's execution flow.

Representation. To represent a chromosome (a candidate partition configuration), an array of 'genes', where each gene depicts the partitioning for a table in the database can be used. For instance, a chromosome for the TPC-H benchmark could be represented as follows: {`nation`, `supplier`, `region`, `lineitem`, `orders`, `partsupp`, `customer`, `part`} $\rightarrow$ {R,R,R,D_1,D_2,D_1,D_1,D_1}, where R stands for replicated, and D for distributed[5] (i.e., hash-partitioned), and the index next to D depicts the index of the column in the table's schema that serves as the distribution column. For instance, `nation` is replicated and `lineitem` is distributed on the first column (`l_orderkey`) here. Usually chromosomes tend to be represented at the lowest possible level: for a given order of tables, and a given order of columns for a table, a chromosome can be simply represented as an array of numbers, where each number represents a column for a table (if the number is 0, it means replication, otherwise, it is the index in the list of columns for the table).

Fitness function. The fitness function interprets the chromosome and evaluates its 'fitness'. The definition of fitness function is crucial, because it must accurately measure the desirability of chromosome, and its evaluation should be as efficient as possible due the large number of invocations. We use the optimizer's estimated cost as the fitness function (or ∞, if the configuration does not satisfy the storage constraint).

Reproduction. The reproduction process includes the selection, crossover, and mutation of the chromosomes briefly described next.

Selection. Chromosomes are selected from the population and then recombined, producing offsprings. Parents are randomly chosen from the population favoring fitter chromosomes.

Crossover. Crossover cuts a pair of chromosomes at some randomly chosen point. The 'tail' segments of the two strings are then exchanged to generate new chromosomes, inheriting some genes (characteristics) from each parent.

Mutation. After crossover is completed, mutation is applied to each offspring individually. Mutation alters each gene with a typically small probability. For example, *Mutate*({R,R,R,D_1,D_2,D_1,D_1,D_1}) = {$\mathbf{D_2}$,R,R,D_1,D_2,D_1,D_1,D_1} results in changing the first table from being replicated to partitioned on the 2nd column. Crossovers tend

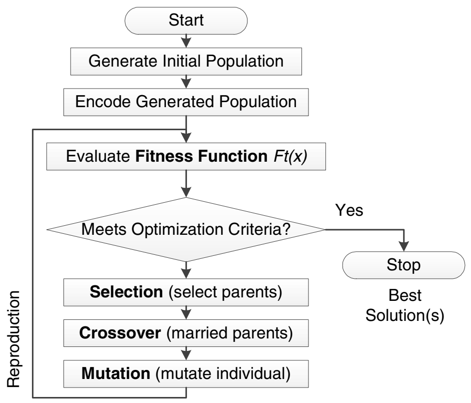

Start
Generate Initial Population
Encode Generated Population
Evaluate Fitness Function Ft(x)
Meets Optimization Criteria?
Yes
Stop
Best Solution(s)
Selection (select parents)
Crossover (married parents)
Mutation (mutate individual)
Reproduction

Figure 6: Flowchart for genetic algorithm.

to be more important than mutations for exploring the search space rapidly. Mutations provide a small amount of random search and are used to ensure that selection and crossover do not lose potentially useful genetic characteristics. Furthermore, mutations can help ensure that all points in the search space would eventually be examined.

4.3 Disadvantages of Shallowly-Integrated Approaches

A typical database is likely to have a large number of tables and a workload may contain many queries. Thus, the search space of all feasible partitioning configurations is likely to be extremely large due to combinatorial explosion of possibilities. Furthermore, each evaluation of a partitioning configuration is expensive because it involves optimizing (in the what-if mode) all queries in the workload. An unguided enumeration of partitioning configurations won't scale with respect to the number of tables and the workload size. Furthermore, a shallowly-integrated approach, while not requiring any (or little) changes to the MPP optimizer may result in substantial duplication of work each time a new partitioning configuration needs to be evaluated (this includes query parsing, validation, binding, security handling, join reordering, and any other partition-independent work) and only a little amount of configuration-specific work [10]. In the next section we present a different way to conduct the search for the best partition configuration for parallel databases that addresses these problems by means of a deeper integration with the query optimizer.

5. TUNING WITH DEEP OPTIMIZER INTEGRATION

In this section we propose a partition configuration tuning approach that exploits the parallel query optimizer in a deeply integrated manner. Specifically, we propose a new algorithm called the *<u>Me</u>mo-Based <u>S</u>earch <u>A</u>lgorithm* (or *MESA* for short), which is based on two key components: (1) leveraging the optimizer's internal `MEMO` data structure to perform the what-if optimization more efficiently, and (2) a bounding technique on the quality of partial partitioning solutions, which forms the basis for our branch and bound search enumeration. We describe these two components next.

5.1 Workload MEMO Data Structure

The parallel query optimization (described in Section 2.3) consists of two phases: (1) the *serial optimization against the shell appliance*, and (2) the *parallelization of the search space*, from which the best parallel execution plan is extracted. As discussed in Section 2, the `MEMO` provides a compact representation of the search space of plans. We extend the idea of `MEMO` further, and cre-

[4]A parent configuration corresponds to the configuration with a higher benefit than the current configuration *C* that differs from *C* in exactly one partition.

[5]We use the verbs "distributed" and "partitioned" interchangeably. Both mean the same thing in the context of this paper.

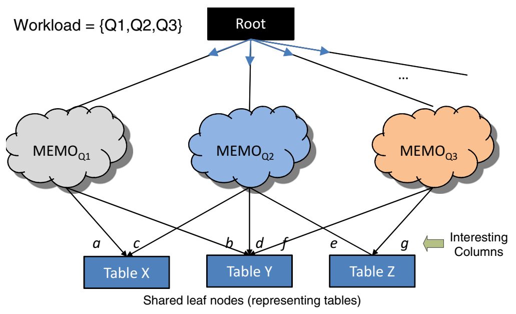


Figure 7: Workload MEMO data structure.

ate a "*workload* MEMO" data structure, which can be described as the "union" of the individual MEMOs for the queries in the workload. Figure 7 graphically shows the workload MEMO for three queries (Q1-Q3). To form a workload MEMO the following steps are executed: (1) we obtain an individual MEMO for every query in the workload; (2) we attach the global root node with the outgoing edges to each individual MEMO (the black node at the top of Figure 7); (3) we merge the leaf nodes – which represent the various access paths to the tables in the database by the workload queries. After the merge, we obtain distinct leaf nodes (one per each table) that are shared by the individual MEMOs in the workload MEMO (the nodes at the bottom of Figure 7).

The overall algorithm to re-optimize a query under an arbitrary partitioning configuration is similar to the parallel post-optimization step and proceeds as follows: plans (along with their required partitioning properties) are identified top-down starting with the root group; costs, in turn, are computed bottom-up. The required partitioning property of a node depends on its parent (e.g., a hash-join node induces required partitioning on join columns to each of its children). If a node does not satisfy its required properties, a data movement operator (e.g., a shuffle or a replication operator) is added to the plan. The cost of a MEMO *groupExpression* depends on its type: if it corresponds to a leaf node, we identify the partitioning of the underlying base table and estimate the cost of the operation with respect to it. If it is an internal node, we derive the required partitioning properties for the children nodes and calculate the best cost by adding to the *local cost* of the *groupExpression* the sum of the best costs of each of its children (calculated recursively). Among all alternative plans in a MEMO group, we pick the one that satisfies the required properties and has the lowest cost. More details on the cost computations and the best plan extraction from the MEMO, the interested reader is referred to [10, 18].

Note, that by leveraging the MEMO data structure, we can simulate the optimization of the workload under arbitrary configurations by simply repeating the parallel post-processing step on the MEMO by simply adjusting the initial partitioning configuration for the MEMO leaf nodes. In addition to this fast evaluation mode, we can also infer additional properties, described next.

5.2 Interesting Columns

Interesting columns in the parallel query optimizer represent an extension of the notion of *interesting orders* introduced in System R [23] (e.g., a subplan that returns results partitioned in a certain way can be preferable to a cheaper alternative, because later in the plan the partitioning is leveraged obtaining a globally optimum solution). The parallel query optimizer considers the following partitioning columns to be interesting: (a) columns referenced in equality join predicates, and (b) any subset of group-by columns. Join columns are interesting because they make *local* and *directed* joins possible[6], and group-by columns are interesting because aggregations can be done locally at each node and then combined. By definition, only such interesting columns should be considered as partitioning candidates. Other columns would not be useful as partitioning columns by any operator in the MEMO and thus can be safely omitted from consideration.

5.3 *-partitioning

We introduce a special type of partitioning in MESA, the so-called **-partitioning*, which means that "*every*" partition or replication option for a base table is simultaneously available. If a table is *-partitioned, the parallel optimizer can pick the concrete partitioning column that is best suited for every given partitioning request (i.e., one that does not requires moving data around) during parallelization post-processing. If the table size is below the storage bound, the optimizer can also consider replication. In this way, the optimizer simultaneously considers all possible partitioning alternatives for *-partitioned tables during a single post-processing step, and returns the execution plans with the smaller overall cost. Of course, not all resulting plans are valid when using *-partitioned tables. Specifically, if the same table delivers different concrete partitioning columns (for the same table) in the final execution plans of queries in the workload, the resulting configuration is not valid because in reality each table can be physically partitioned in a single way. However, this mechanism enables the optimizer to always pick the best concrete partition scheme for a *-partitioned table, and thus obtain lower bounds on the cost of configurations that are partially specified (i.e., configurations that include some *-partitions), and do so without issuing additional optimization calls. This bounding function is an important aspect of our search strategy, described next.

5.4 Branch and Bound Search

The *branch and bound* method [21, 22] is one of the most frequently used approaches to address large search space enumeration problems. The method is based on the observation that the enumeration of solutions has a *tree structure*, and the main idea in branch and bound is to avoid growing the whole tree as much as possible (see Figure 8). Instead branch and bound grows trees in stages, and grows only the most promising nodes at any stage. It determines which node is the most promising by estimating a bound on the best value of the objective function that can be obtained by growing that node to later stages. Another important aspect of branch and bound is *pruning*, which discards whole subtrees when a node or any its descendants will never be either *feasible* or *optimal*. Pruning prevents the search tree from growing too much.

To describe branch and bound in detail, we first need to introduce some terminology:

- *Node*: any partial or complete solution. Specifically, a node associates each table with either a concrete partitioning strategy, replication, or the *-partitioning option.
- *Leaf*: a complete solution in which no table is allowed to be *-partitioned (e.g., $\mathbf{S}_3$ in Figure 8).
- *Bud*: a partial solution with some *-partitioned tables. This is a node that might yet grow further by replacing a *-partitioned table with either a concrete partitioning scheme or replication (e.g., $\mathbf{S}_1$ and $\mathbf{S}_2$ in Figure 8).

[6]*Local join* is a join between tables that can be performed locally at each node. *Directed join* can be performed when one of the tables is already partitioned on the join key, and the other table can be dynamically re-partitioned on the join key to make it distribution-compatible.

```
MESA (W:workload, B:storage bound)
wMemo = CreateWorkloadMemo(W, B)
incumbent = null
bbTree = CreateRoot(wMemo)
while (!stop_condition())
  currConfig = SelectNode(bbTree) // DFS policy
  newConfig = CreateChildConfig(currConfig) // table/column selection policy
  if (newConfig violates B constraint)
    prune(newConfig)
  else
    cost = ParallelPostProcess(wMemo, newConfig)
    if (newConfig is leaf or can be promoted)
      if (cost < incumbent.cost)
        incumbent = newConfig
      prune(newConfig)
    else // partially defined configuration
      if (incumbent.cost < cost)
        prune(newConfig)
return incumbent
```

Figure 9: Memo-based search algorithm using branch and bound enumeration.

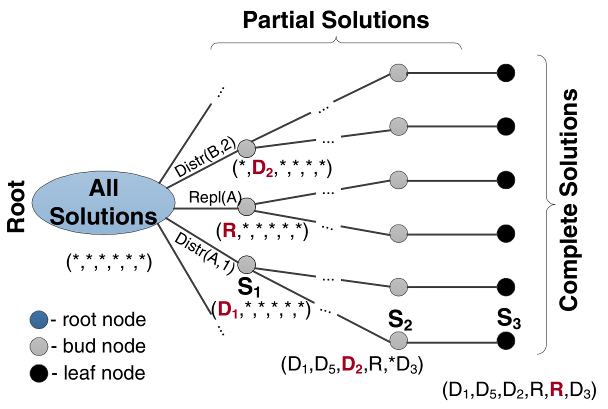


Figure 8: Branch and bound enumeration tree for partitioning configuration search problem.

- *Bounding function*: a lower bound on the cost of the workload under a given (partially specified) configuration. If the configuration is fully specified (i.e., without *-partitioned tables), the bounding function is exactly the expected cost of the workload under such configuration.

- *Incumbent*: the best fully specified (i.e., feasible) solution that satisfies the space constraint found at a given point during the execution of the algorithm.

Figure 8 shows an example of the enumeration tree for a partitioning configuration search problem. The root of the tree labeled "*all solutions*" consists of all *-partitioned tables. Leaf nodes represent fully specified partitioning configurations with associated costs. A bud node, e.g., $\mathbf{S}_1$ represents a partial solution, where the first table is partitioned on the first column (in the table's schema) and the rest of the tables are *-partitioned.

5.5 Branch and Bound Policies

In order to fully specify our search strategy, we need to define the following policies:

Node selection policy: The *node selection* policy governs how to choose the next bud node for expansion. There are several popular policies, and we use one based on depth-first search. That is, at each iteration we pick the last node we've expanded. When we reach a leaf node, we backtrack to the earliest ancestor that is not fully explored and continue from there. A good property of this policy is that the first incumbent is reached quickly, which in turn enables more aggressive pruning of subsequent portions of the search space, and speeds up the overall search (as described in Section 6).

Table/column selection policy: Once a bud node has been chosen for expansion, the next question that must be addressed is how to pick a *-partitioned table to instantiate, and how to partition or whether to replicate that table. For that purpose, we rank all interesting columns (defined in Section 5.2) and pick them in order. The ranking of a column *c* is the total cost of all queries that have *c* as an interesting column (more advanced ranking functions can be used [25], but this simple strategy already gives good experimental results). Note that before trying any partitioning on a table we attempt to replicate it (for small tables it is usually a good idea to use replication, and large tables would immediately fail the storage constraint and such configuration would not be searched further).

Pruning strategy: There are two main reasons to prune a bud node. The first case is when no descendent node will be feasible. Specifically, if the total space used for replication exceeds the storage constraint, we know that no descendant of the current bud would fit either, and we can prune the subtree. The second case is when no descendant will be optimal. If the bounding function of the current bud node is worse than the objective function of the current incumbent, we know that no feasible solution that is a descendant of the current bud would be optimal, and we can prune the subtree as well.

Bud Node Promotion: The procedure described in Section 5.3 not only returns a lower bound on the cost of a partially specified configuration, but in some cases, can fully specify the optimal configuration as well. Specifically, if every table in the database delivers a unique partitioning column for all execution plans in the workload, then this is the optimal configuration that would eventually be found by the branch and bound technique after instantiating all remaining *-partitioned tables. In such case, it is not necessary to actually conduct the search, but we can instead replace the bud node *b* with the corresponding leaf node that is optimal when varying all of *b*'s *-partitioned tables, and, in effect, fully prune the corresponding subtree.

Stopping condition: This condition stops the search and returns the incumbent in case it is not possible to exhaustively enumerate the whole search space. This policy could be either time-based, or iteration-based, or if no improvement over a certain period of time has been obtained.

5.6 MESA Algorithm

Figure 9 depicts a pseudocode of the MESA algorithm. First, we create the workload `MEMO` and set the current incumbent to `null` (lines 1-2 in the figure). Next, we create the branch and bound tree root, which contains all *-partitioned tables (line 3). Until the stop condition is met (line 4), we perform the following steps. We first select the next promising configuration based on the DFS policy (line 5). Then, we pick a *-partitioned table and create a child configuration by either replicating or partitioning such table based on the table selection policy (line 6). If the resulting configuration does not fit in the storage bound *B*, we prune it. The reason is that after we exceed the budget *B*, the remaining *-partitions would be subsequently resolved into replication – adding more space still – or partitioning, which would not change the space consumed by replicated tables, thus keeping the configuration invalid. If otherwise the configuration satisfies the replication space budget, we perform the parallelization post-processing of the `MEMO` with respect to the new configuration as explained in Section 5.3 (line 10). If the configuration is a leaf node (or can be promoted to one), it is fully specified, and the value of `cost` represents the actual cost of evaluating the workload under such configuration. In this case (lines 12-14) if the cost of the new configuration is smaller than that of the incumbent we make the incumbent equal to the new configuration (note that cost of `null` is ∞). In line 14, we prune the new configuration in case it was promoted from a bud, since we already inferred the optimal configuration and thus do not need to continue searching from that point. If, instead, the configuration is partially specified in line 10, the value of `cost` represents a lower bound on the cost of any configuration derived from the new one. If `cost` is worse than that of the incumbent, we prune the new configuration (lines 16-17), because no solution derived from it would be better than the current incumbent. When the stopping condition is met, we return in line 18 the current incumbent solution, which satisfies the storage constraint with minimum cost among the explored partition configurations.

One of the key benefits of our approach is that we do not call the optimizer multiple times, but instead gather all the common information once, and then perform the light-weight parallelization using the `MEMO` data structure considering various partition configurations. Furthermore, the by-product of the `MEMO` analysis and our newly introduced *-partitioning scheme gives us the capability to compute a lower bound on the costs of partial configurations that allow the pruning of many alternatives without loss in quality of the resulting partitioning recommendations.

6. EXPERIMENTAL EVALUATION

In this section we report our experimental evaluation of the techniques presented in the paper. We implemented our approach in a real MPP system, namely the Microsoft SQL Server 2008 Parallel Data Warehouse, with 8 compute nodes. The partitioning advisor was run on a machine with 12GB, Intel(R) Xeon(R) CPU E5540 @2.53GHz, with Windows 7 OS. For our experiments, we use several real and synthetic benchmarks shown in Table 1. Based on the current capabilities of the parallel optimizer we consider, we've picked a subset of queries from some benchmarks (e.g., TPC-DS, MSSales).

The goal of the experimental evaluation is to:

- Compare the partitioning advisor with shallow and deep integration approaches in terms of quality of recommendations and performance efficiency.
- Measure the internal overheads and the scalability of our proposed MESA algorithm.

Benchmark (scale)	# Tables	Workload (# queries)
TPC-H (1TB)	8	22
TPC-DS (1TB)	25	50
L'Oreal (88GB)	573	29
MSSales (800GB)	346	27

Table 1: Experimental benchmarks

There are several control parameters that govern the GA approach, which we list in Table 2. Parameters for the Rank-based approach are the same as in [25], and default MESA parameter values are depicted in Table 3.

Parameter	Value	Description
# of generations	100	# of times the population will be replaced through reproduction.
Population size	30	# of chromosomes available for use during the search. If the size is too big, GA will spend unnecessarily long time evaluating chromosomes, if it is too small, GA may have no chance to adequately cover the search space.
Crossover rate	0.1	the probability of crossover between two chromosomes.
Mutation rate	0.1	the probability that values of genes of a newly created (or selected) off-springs will be randomly changed.
Selection rate	0.2	the percentage of the worst of the current population that will be discarded (after re-generation)

Table 2: GA parameters

Parameter	Value	Description
Node selection	DFS	the forward- and the back-tracking policy in the branch and bound tree
Variable selection	replicate, distribute by rank	See Section 5.5 for details.
Stop condition	150	the number of iterations after which the search terminates

Table 3: MESA parameters

6.1 Shallow Vs. Deep Integration Approaches

We first measure the quality of the techniques discussed in Section 4 and Section 5, namely, the rank-based search, the genetic search and our proposed MESA algorithm. In the rest of the section, we will refer to them as *Shallow (Rank)*, *Shallow (GA)*, and *Deep (MESA)*. We compare the quality of the recommendations produced by each technique using the optimizer's estimated cost for the best partitioning configuration. We go with the notion that the optimizer's costs represent an accurate measure of the quality of the final execution plans and are considered as a viable evaluation mechanism in physical design tuning tools (without the need to actually execute the queries in the appliance). This is a de facto standard both in the industry and academia for estimating the quality of physical design tuning[7]. In the experiment, we set the replication storage bound to 0, so that all tables are required to be partitioned in the final configuration recommendation.

Table 4 shows the relative quality of the final recommendations[8] produced by the three techniques. The third column ("Quality(Q)")

[7]However, in practice, we've observed that optimizers' query cost estimates may contain errors, as the optimizer need to be correct about relative plan costs, not actual costs (to correctly rank the plans and choose good plans over bad ones).

[8]Here, we let each approach run to its completion.

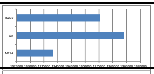
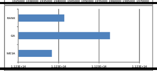
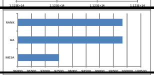
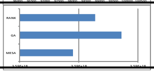
	Approach	Quality(Q) Visual Depict.	MESA Imp(Q)	Total Time(T)	MESA Imp(T)
	RANK		1.1x	6 min 42 sec	1.73x
TPC-H	GA		1.3x	4 hrs 28 min	69.3x
	MESA		–	3 min 52 sec	–
	RANK		1.1x	3 h 53 min	7.13x
TPC-DS	GA		1.6x	25 h 28 min	46.7x
	MESA		–	32 min 42 sec	–
	RANK		1.2x	45 min 25 sec	9.05x
L'Oreal	GA		1.2x	14 h 12 min	169x
	MESA		–	5 min 1 sec	–
	RANK		1.1x	6 h 11 min	3.37x
MSSales	GA		1.2x	27 h 39 min	15.1x
	MESA		–	1 h 50 min	–

Table 4: Comparison of techniques

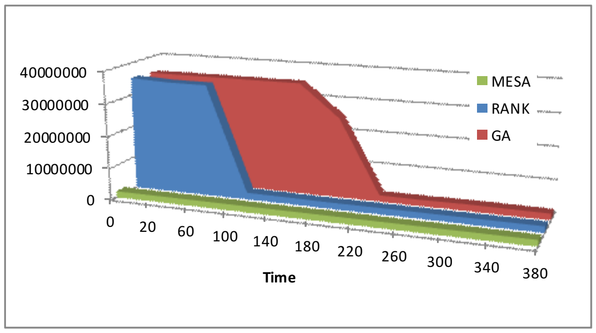


Figure 10: Quality over time: TPC-H.

visually illustrates the relative quality of each technique, and the fourth column ("MESA Improvement (Q)") summarizes the improvement in the quality by MESA (over the shallow approaches) quantitatively. We note that if time is not a constraint, each algorithm eventually converges to a relatively good recommendation. However, even without the time restriction, we've observed that in some cases both RANK and GA might still not get to the best recommendations as compared to MESA[9]) (even after hours of execution). At the same time, MESA takes significantly less time to arrive to good quality partitioning configurations due to the efficient reoptimization strategy reusing the workload `MEMO` structure. The last column "MESA Imp(T)" in the table depicts the improvement by MESA over these shallowly-integrated approaches. As one can observe, it is several orders of magnitude for various workloads. We give detailed explanation for MESA's efficiency compared to the alternatives in Section 6.3.

Figures 10-13 depict the quality over time of the three techniques with the workloads we've considered. We observe that MESA produces high quality recommendations in a much shorter period of time. This is due to faster evaluation of configurations using workload `MEMO` and branch and bound pruning which results in evaluating fewer sub-optimal partitioning configurations. Also, notice that if we were to abrupt the search early (and not let the shallow approaches run to their completion), the relative quality of MESA would be several orders of magnitude higher (e.g., 3-4x higher for TPC-H at 80 seconds). This can be explained as follows. GA enumerates and evaluates a large number of chromosomes (where each chromosome represents a partitioning configuration) to maintain the required population size and continuously performs random changes to the chromosomes (through mutation and crossover operations). Given that GA is a randomized search, it can often

[9]Furthermore, note that some workloads contain a few queries that, even under an optimal partitioning strategy, fully dominate the cost of the workload. Removing these queries from the workload would increase the gap between MESA and the remaining techniques (in some cases, significantly so).

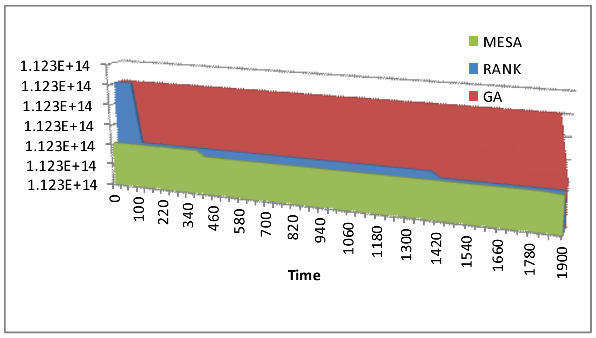


Figure 11: Quality over time: TPC-DS.

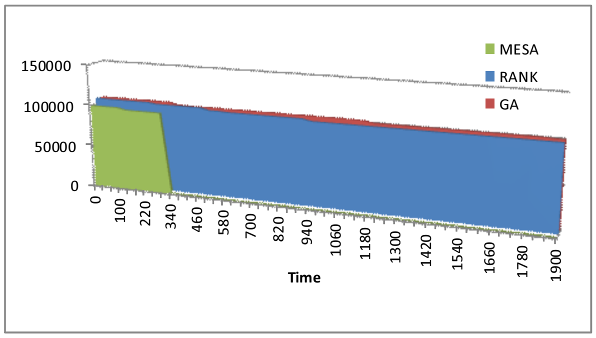


Figure 12: Quality over time: L'Oreal.

lead to trying various partitioning configurations that have little or no improvement. Rank-based approach, on the other hand, relies on the high quality of benefit values of the candidate partitions in the candidate set. Obtaining these values accurately is a challenge. It requires multiple calls to the optimizer to re-optimize queries under various partitions (to obtain those benefit values to associate with each partitioning scheme). The more choices per table there are, the larger this overhead becomes. Furthermore, RANK still uses regular what-if optimization calls to estimate the cost of each possible partitioning configuration, which is expensive in terms of elapsed time and slows down the overall search.

6.2 Impact of Replication Bounds

Figure 14 shows the impact of the storage bound B (from the Problem Definition in Section 3) on the quality of the partition recommendations for the TPC-H workload when using MESA. From Figure 14 we can see that 1GB and 5GB replication bounds improve the quality of the recommendations by MESA over 0GB (when no replication is allowed). This is due to replicating `nation` and `region` (which are under 1GB bound) and additionally `supplier` (under 5GB bound). However, the subsequent increases in the storage bound do not result in smaller cost final recommendations. While `customer` and `part` tables could be replicated under 30GB and 50GB bounds, the optimizer returns higher costs for such configurations.

6.3 Performance of MESA

In this section we report several experimental results on the performance aspects of our technique.

6.3.1 *Workload MEMO Construction Overhead*

To evaluate the overhead of creating the workload `MEMO`, we optimize each of the workloads in normal mode and compared it to the overhead of the first optimization with MESA (which includes creating workload `MEMO`). Figure 15 shows the overheads of

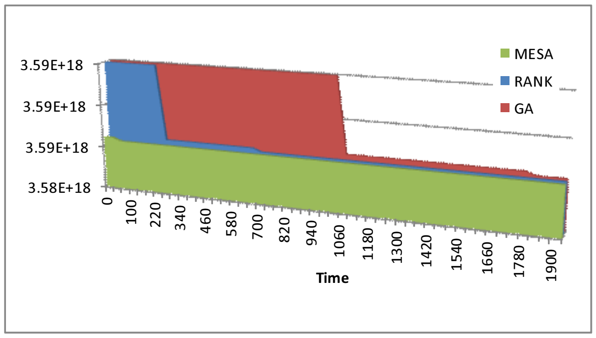


Figure 13: Quality over time: MSSales.

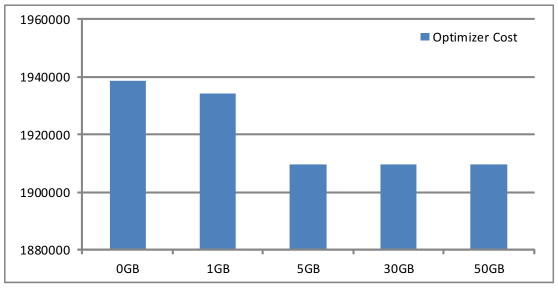


Figure 14: Quality of recommendations under various replication bounds.

the workload MEMO construction compared to the total optimization time. While this overhead is specific to the MESA algorithm and does not appear in alternative approaches, the subsequent speedup in reoptimization time quickly amortizes this initial overhead. We can see that the workload MEMO construction time is no more than 1.4 times that of a regular what-if optimization call (and in most cases is around 1.2x). Assuming that subsequent optimization calls for the same query using workload MEMO are cheap (see the next sub-section), after just a couple of optimization calls we can completely amortize the additional overhead of creating workload MEMO. In contrast, shallow integration approaches, like GA and RANK, will incur a traditional optimization call overhead (depicted in red in Figure 15) for *every new partitioning configuration*.

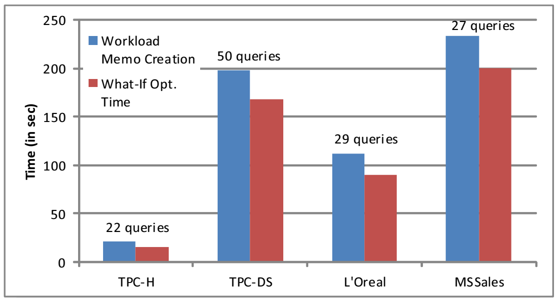


Figure 15: Time overhead of workload MEMO creation.

6.3.2 Subsequent Reoptimization Calls

We now measure the average time to produce plans and costs in regular optimization mode and when using workload MEMO. For that purpose, we compared the time to optimize each of the queries under the workload MEMO against the regular optimizer. We used several different configurations considered by partitioning advisor and averaged the results. Figure 16 shows that obtaining the workload MEMO is effective in reducing the total time spent in producing a parallel query plan. We can see from Figure 16 that the average speedup per query when using workload MEMO varies from 3.5x to 19x. To put these numbers in perspective, Table 5 shows the total number of optimizations with the workload MEMO that are possible per regular optimization for a randomly picked set of queries (from various workloads), including the first call to obtain the workload MEMO. From Table 5 we can see that we get 6x improvement for TPCH-H(Q2), 90x for TPC-DS(Q4), 15x for L'Oreal(Q3) and 14x for MSSales(Q5) when using MESA's fast evaluation mechanism.

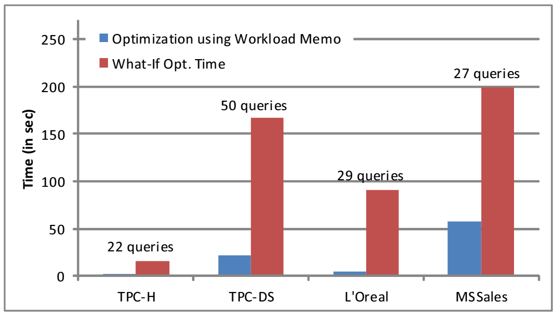


Figure 16: Speedup of subsequent optimizations using workload MEMO.

Orig.	TPCH-H(Q2)	TPC-DS(Q4)	L'Oreal(Q3)	MSSales(Q5)
1	0	0	0	0
2	13	180	30	28
3	20	270	46	43
4	27	360	61	57
5	34	450	77	71

Table 5: Total number of optimizations with workload MEMO per regular optimization call

7. EXTENSIONS

For ease of presentation, we so far have addressed a slightly simplified scenario. We now discuss extensions that generalize the applicability of our approach.

Updates: Throughout the presentation we assumed that workloads do not contain updates. While shared-nothing parallel systems are typically used for mostly read workloads, we still need to address updates in the partitioning tool. The main impact of an update is that some (or all) partitions of a distributed table must be updated, and if a table is replicated, all copies must be kept in sync with each other. The cost estimations by the parallel optimizer take this aspect into account when evaluating a given partitioning configuration. The only variation applies to *-partitioned tables. To obtain the lower bounds, we do not assign an update cost to tables that are *-partitioned in the input configuration, but instead calculate the minimum update cost for such table (under any possible configuration) and use that value to obtain the lower bound.

Multi-Column Partitioning: The original MESA algorithm considers single-column partitioning strategies to align with the current query processing capabilities of the underlying MPP engine. In general, however, we can partition a table using multiple columns. During parallel post-processing step, we identify candidate column sets that we can use to partition each table (e.g., multi-join predicates or group-by clauses). Then, during the branch-and-bound algorithm, we additionally instantiate such alternatives in Line 6 (in Figure 9). An important consideration is that some partitioning strategies which are not optimal for any query in the workload can be part of the globally optimal configuration. Suppose that some query requires partitioning on columns (a,c) and another query requires (a,d). If one of such strategies is implemented, one query would not be able to leverage the layout and would resort to large data movements. Alternatively, if we partition by column (a), both queries would be able to leverage such strategy resulting in better performance. To incorporate such candidates, we *intersect* the original partitioning column sets and incorporate the results as new candidates in line 6 (such *merged* candidates are always ranked after the original ones). This candidate generation can be done by the merging technique of [25], or by lazily generating alternatives.

Range Partitioning: So far, we considered hash-based partitioning strategies. However, other partitioning methods could be useful in an appliance as well. In range partitioning, rows are mapped to partitions based on ranges of column values. This type of partitioning is useful when dealing with data that has logical partitioning ranges (e.g., months of the year). Performance is best when the data evenly distributes across the range. If partitioning by range causes partitions to vary dramatically in size because of unequal distribution, then other partitioning methods should be considered. Similar to hash-based partitioning, range-based partitioning option can be considered in MESA. To represent range partitions, we need to additionally specify the partition bounds (which can be obtained by leveraging equi-depth histograms).

Interaction With Other Physical Design Structures: Modern database systems support a number of different physical structures, such as indexes, materialized views, etc. These structures can contribute substantially to the combinatorial explosion of physical design search space, and can be recommended either in isolation or integrated into a combined search space together with partitioning. Some studies in the literature suggest that the degree of dependency among different physical structures can be leveraged in tuning [14]. Specifically, some physical structures can be *strongly* dependent on each other (if a change in selection of the former affects the latter), while others *weakly* depend on each other. Mutual strong dependencies are difficult to *break* and thus are better handled by using a combined approach. If only B strongly depends on A, we can iteratively search A followed by B, so that B is properly influenced by A. Weakly coupled components can be scheduled separately in any order. In [14], it is argued that partitioning (specially in the context of a MPP) weakly interacts with indexes. Therefore, a practical design advisor might choose to recommend partitioning (using MESA) followed by a subsequent index tuning step.

8. RELATED WORK

While there has been a lot of work done in the area of automating physical database design on a single database server [8, 12, 9, 25, 10, 27], much less attention has been given to the problem of database tuning in distributed environments [11, 31, 25]. To the best of our knowledge, the closest study to this paper is by Rao et.al [25]. Section 4 contains a detailed description of this work.

Vertica Database Designer [5] automatically selects physical design for vertically partitioned tables, specifically, it finds a good set of overlapping projections for a table based on representative workload. It determines projections that optimize performance of sample workload, which columns and joins they will contain, and how each projection should be sorted and compressed. Unfortunately, the technical details of the Vertica DB Designer are not public.

Agrawal et.al introduced an approach that integrates vertical and horizontal partitioning into automated physical database design on a single node environment [9]. The focus of this work significantly varies from our problem statement, because [9] focuses on manageability on a single node machine, focusing on the index alignment aspect, where indexes are horizontally partitioned in the same way as the underlying tables. This problem is orthogonal to the problem we address in this paper. Our work and [9] are complementary, and we could integrate their approach to incorporate alignment together with partitioning on multiple nodes.

Recently, Curino et.al proposed Schism partitioning system for OLTP workloads [11]. The main idea in Schism is to represent a database and its workload as a graph, where tuples (or groups of tuples) are represented by nodes and transactions are represented by edges connecting the tuples used within the transaction. The idea is to apply a graph partitioning algorithm to find balanced partitions that minimize the weight of cut edges (to minimize the number of multi-sited transactions). While the proposed approach is targeting a related problem, its focus is on extremely fine-grained partitioning beneficial for OLTP workloads, where transactions are short-running and touch very few tuples. We, however, focus on complex long-running SQL queries that access large amounts of data (a typical scenario in large data warehouse environments).

Ghandeharizadeh et.al proposed the hybrid-range partitioning strategy in [26], which is a hybrid approach to hash- and range-partitioning based on query analysis. The approach attempts to decluster (run in parallel on several nodes) long-running queries and localize small range queries. The applicability, however, is limited to single-table range queries over one dimension, and do not consider replication.

The second broad area that relates to our work includes distributed architectures and distributed query processing [15, 17, 7, 20, 16, 30, 24, 28, 6]. While most of these techniques focus on optimizing parallel query execution in distributed environments, largely they assume that databases have been optimally partitioned to start with. However, if initially data is placed in sub-optimal way, the subsequent query executions may be performing a lot of data movements and the overall system performance may dramatically decrease.

9. CONCLUSION

In this paper we describe techniques for finding the best partitioning configuration in distributed environments. Our approach relies on deep integration with the parallel query optimizer, using its internal MEMO data structure for faster evaluation of partitioning configurations and to provide lower bounds during a branch and bound search strategy. Our experiments show that MESA produces high-quality recommendations in reasonable amounts of time outperforming shallowly-integrated tuning approaches, which face long delays waiting for the optimizer to perform "what-if" optimizations.

10. REFERENCES

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

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

[3] Netezza. http://www.netezza.com/.

[4] SQL Server 2008 R2 Parallel Data Warehouse. http://www.microsoft.com/sqlserver/2008/en/us/parallel-data-warehouse.aspx.

[5] The Vertica Analytic Database Technical Overview White Paper. http://www.vertica-solutions.com/files/10986 /verticaarchitecturewhitepaper.pdf.

[6] A. Abouzeid et.al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.

[7] A. Kiran et.al. Two techniques for on-line index modification in shared nothing parallel databases. In *SIGMOD*, 1996.

[8] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[9] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[10] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD*, 2008.

[11] C. Curino et.al. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, 2010.

[12] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *VLDB*, 1997.

[13] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' index analysis utility. In *SIGMOD*, pages 367–378, 1998.

[14] D. C. Zilio et.al. DB2 design advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.

[15] D. DeWitt and J. Gray. Parallel database systems: the future of high perf. database systems. *Com. ACM*, 35(6):85–98, 1992.

[16] G. Cong et.al. Distributed query evaluation with performance guarantees. In *SIGMOD*, pages 509–520, 2007.

[17] Ganguly, Sumit et.al. Efficient and accurate cost models for parallel query optimization. In *PODS*, pages 172–181, 1996.

[18] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

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

[20] Isard, Michael et.al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[21] W. H. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *J. ACM*, 21(1):140–156, 1974.

[22] T.-H. Lai and A. P. Sprague. Performance of parallel branch-and-bound algorithms. In *ICPP*, pages 194–201, 1985.

[23] P. G. Selinger et.al. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[24] R. Chaiken et.al. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[25] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[26] S. Ghandeharizadeh et.al. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *VLDB*, pages 481–492, 1990.

[27] S. Papadomanolakis et.al. Efficient use of the query optimizer for automated database design. In *VLDB*, 2007.

[28] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Map-Reduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[29] T. Stöhr et.al. Multi-dimensional database allocation for parallel data warehouses. In *VLDB*, 2000.

[30] Y. Hung-Chih et.al. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[31] D. C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1998.