2012 IEEE 28th International Conference on Data Engineering Workshops

Automatic Data Placement in MPP Databases

Carlos Garcia-Alvarado[#1], Venkatesh Raghavan[*2], Sivaramakrishnan Narayanan[*2], Florian M. Waas[*2]

[#]*Computer Science Department, University of Houston;* [*]*Greenplum, A Division of EMC*

[1]cgarcia@cs.uh.edu; [2]{firstname.lastname}@emc.com

***Abstract*— Physical design for shared-nothing databases includes decisions regarding the placement of data across a cluster of database servers. In particular, for each table in the database a *distribution policy* must be specified. In general, the choice of distribution policy affects the performance of query workloads significantly as individual queries may have to redistribute data on-the-fly as part of the execution. As is the case with a number of other physical design decisions, the problem is hard and poses substantial difficulties for database administrators.**

In this paper, we present FINDER, a design tool that optimizes data placement decisions for a database schema with respect to any given query workload. We designed FINDER with portability in mind: The tool is fully external to the target database system, i.e., does not require any code-level integration with the system, and avoids reverse engineering of query optimization techniques. Our experiments show FINDER converges quickly and delivers superior results compared to state-of-the-art solutions.

I. Introduction

In order to scale to multiple petabytes of capacity, shared-nothing parallel data warehouses leverage large clusters of commodity servers with local, direct attached storage. This architecture provides enormous scalability with respect to both, storage capacity and compute power to execute queries (see e.g. [9]). However, the shared-nothing nature of these systems also entails challenges when it comes to the physical design of databases. In particular, the assignment of data to the individual segment servers of a clusters presents an interesting opportunity for optimization. In such systems, the syntax for DDL statements is extended to allow administrators to specify a *distribution policy*. For example, consider the ORDERS table of the TPC-H benchmark. In Greenplum Database, the following statement will create the table using O_ORDERKEY as a hash key for the distribution, i.e., the rows of the table will be assigned to segments based on the hash value of their respective order key:

```
CREATE TABLE ORDERS (O_ORDERKEY INT, ...)
DISTRIBUTED BY (O_ORDERKEY)
```

Tables may be distributed by individual or multiple columns. Other shared-nothing database systems provide syntax extensions similar in nature (see e.g. [15]).

Choosing a distribution policy has immediate effect on query performance. By distributing frequently joined tables using the join keys, rows that are to be joined are co-located, i.e., matching rows are guaranteed to reside in the same segment server. The query optimizer can exploit this knowledge and create query plans that execute the join locally. In contrast, when joining tables whose data is not co-located, the optimizer is forced to create plans that redistribute data on-the-fly as part of the execution of the query. The redistribution may incur significant overhead and slow down query execution markedly. Another important scenario is aggregation, expressed as GROUP BY in SQL. Consider the following query to determine the revenue per order:

```
SELECT SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT))
FROM LINEITEM
GROUP BY L_ORDERKEY
```

In order to compute the aggregate function for each group, all data pertaining to a group must be located at the same segment. In Figure 1, the execution of a possible query plan for the example query—including *redistribution operators (REDIST)*—is shown for two segment servers. The redistribution operators route tuples accordingly to ensure groups are co-located. Note that different data placement of the underlying table may provide co-located data and make a redistribution unnecessary. In short, specifying distribution policies for all tables so that the data movement due to redistribution during querying is minimized is crucial and offers significant performance benefits. Besides join and aggregate operations a number of other operators are sensitive to the distribution of their input data, too.

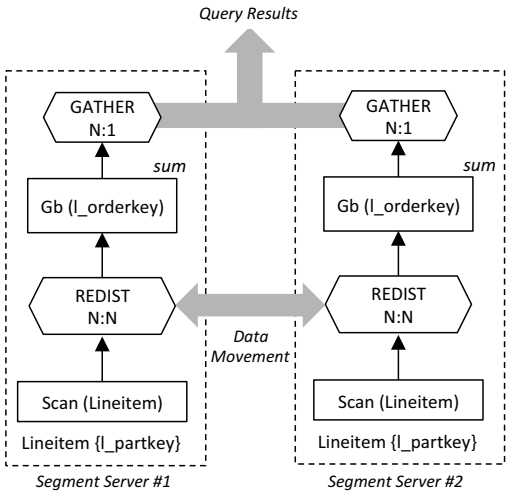

Fig. 1. Execution of sample query on two segment servers

978-0-7695-4748-0/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDEW.2012.45

Given a logical database design and a query workload, finding the optimal set of distribution policies is a non-trivial problem [12]. Moreover, the problem is heavily intertwined with other optimization decisions made by the query optimizer. For example, the redistribution of data may affect the choice of join order. Also, as the schema evolves over time, distribution policies may need to be revisited and adjusted to reflect evolving query patterns. In practice, distribution policies are often decided at the time of the initial implementation of the schema based on the data architect's intuition of what could be frequent query patterns. Assessing the quality of a configuration of distribution policies is challenging even for seasoned experts as distribution policies always have to be viewed in a broader context of a potentially large query workload. Hence, unfavorable configurations go often unnoticed for an extended period of time. Even if a data placement problem is suspected, determining the best possible configuration of distribution policies poses an often unsurmountable challenge to the administrator.

In this paper, we present FINDER, a tool that lets administrators analyze a given workload with respect to the amount of data that needs to be transferred between segment servers when processing the workload. FINDER utilizes the optimizer's *what-if* mode and obtains query plans for the workload. From the query plan, it extracts information about the estimated amount of data that will be redistributed during execution as well as the intermittent target distribution policy, i.e., the distribution enforced by the query plan. Based on this information, FINDER computes a set of candidate distribution policies and approximates an optimal solution over several iterations using a greedy algorithm. The tool is deliberately designed to operate as an entirely external third party tool to facilitate using it with different target database systems. We evaluated FINDER using different workloads and in this paper demonstrate that it produces high quality results within short and configurable running time.

We present an overview of existing techniques and discuss related work in Section II followed by a formal statement of the problem in Section III. In Section IV we describe the algorithms used by FINDER and detail how the data movement is captured and the problem's search space is explored. We present a detailed comparison of our approach with alternative solutions in Section V. Section VII concludes the paper.

II. Related Work

Automatic physical design is an active area of research. In particular, special attention has been paid to the automatic recommendation of indexes and materialized views in the context of conventional database systems. Most notably, Chaudhuri *et al.* have pioneered a number of techniques and developed a tuning advisor utility for Microsoft SQL Server [2], [3], [8], [6]. Similar tools are meanwhile available for other commercial database systems as well. Vertica's Database Designer [1] recommends a set of overlapping projections given data sets and workload and is closely related to materialized view recommendation tools. Recently, several extensions have been proposed to incorporate continuous workload monitoring and adaptive recommendations [14], [7], [4]. While related in nature, these techniques do not address the recommending of data distribution strategy in a MPP system.

In the literature, three algorithms or systems have been proposed to address the data movement problem, the problem we are concerned within this paper:

Zilio *et al.* proposed a *genetic algorithm* that starts out with an initial population of distribution policies—randomly generated or by using primary keys—and obtains new generations of policies by transforming the best distribution policies seen so far [17]. As is typically the case with genetic algorithms, transformations include crossover or mutation of the chromosomes, i.e., the distribution policies. We implemented this technique and will present a comparative analysis in Section V.

Nehme and Bruno presented an tightly integrated solution for Microsoft's Parallel Data Warehouse (PDW), referred to as MESA [12], [5]. MESA analyzes the workload and assigns a rank to every attribute based on the total cost of all queries that have this attribute as an interesting one. For each query, MESA exploits the optimizer's internal MEMO structure to store the query costs for different distribution keys. Unlike our approach, MESA does not take into consideration composite keys but includes table replication. MESA is tightly coupled to the optimizer implementation. This renders the approach not portable, and made it impossible for us to compare our approach with MESA.

Another solution was developed by Rao *et al.* and implemented in IBM DB2 [13]. The technique, referred to as *Rank-Based Algorithm*, is integrated with DB2's optimizer. Using the *RECOMMEND* mode, the optimizer suggests a set of good distribution policies for each workload query. The benefit of a set of attributes is computed as the difference in estimated costs of the query when evaluated under the current distribution policy versus when the recommended policy is implemented. An expansion step is then performed to add additional sets of attributes that can be identified when analyzing the entire workload rather than individual queries. Lastly, the search strategy combines candidate distribution policies for the different tables in different combination and computes the cumulative cost of the workload queries under such a combined distribution policy. We implemented a generalized version of this algorithm outside the optimizer and compared it with our approach in Section V.

III. Problem Statement

The objective of this work is to determine the optimal data distribution policy for a given workload comprising a set of queries.

Let $T = \{T_1, \ldots, T_t\}$ be the set of tables in the schema. Every table T_i is distributed on a set of attributes X_i. The distribution policy of the entire schema is represented as $D = \{X_1, \ldots, X_t\}$. Let $Q = \{Q_1, \ldots, Q_q\}$ be a representative workload where Q_h's are queries on the set of tables T. Finally, let $cost(Q, D)$ be the execution cost of query Q with

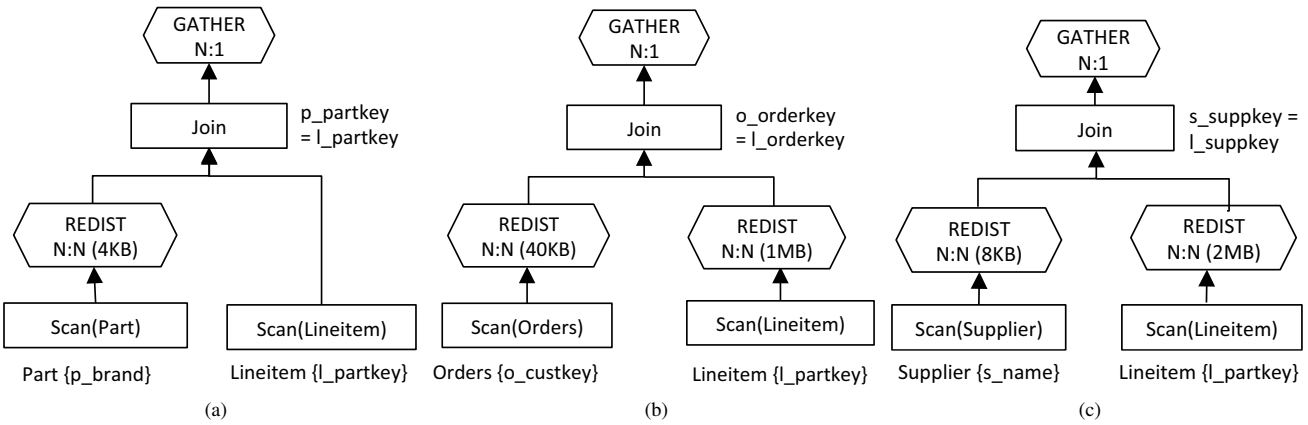

Fig. 2. (a) Execution Plan for Q_1 (b) Execution Plan for Q_2 and (b) Execution Plan for Q_3

a specific distribution policy D. The problem may be specified as finding the D that minimizes the objective function in Equation 1.

$$cost(D) = \sum_{h=1}^{q} cost(Q_h, D) \tag{1}$$

The space of valid solutions is exponential on the total number of columns over all tables. Furthermore, it is not feasible to test the actual execution time of every query with every candidate distribution policy. We rely on the optimizer to provide estimates of intermediate steps in query execution based on database statistics [17], [12]. This problem has been shown to be NP-complete [16].

IV. FINDER Algorithm

In this section, we present the FINDER algorithm as a solution to the distribution design problem as described in Section III. It is built on the intuition that data movement is a significant component of query execution in a shared-nothing parallel database system. It uses a cost model that is heavily based on optimizer estimates that captures the extent of data movement in a workload under a specific distribution policy. FINDER is designed for fast convergence to a good solution by aggressively pruning candidate solutions.

FINDER executes in phases and it maintains (a) the best distribution policy it has found thus far (D_{best}) (b) a set of distribution policies that need to be explored ($\mathfrak{D}$). Initially, D_{best} is a randomly chosen distribution policy and $\mathfrak{D} = \{D_{best}\}$. The algorithm then repeats the following steps:

Evaluation: A candidate distribution policy D_c is chosen from $\mathfrak{D}$ and its efficacy is evaluated using the function in Equation 1. If it is better than D_{best}, then it becomes the new D_{best}. Furthermore, a summary structure called $\mathfrak{M}$ is produced by analyzing the query execution plans produced for the workload Q given the distribution policy D_c.

Generation: The summary structure $\mathfrak{M}$ is analyzed to produce one or more candidates distribution policies which are added to $\mathfrak{D}$ for consideration in subsequent iterations.

These steps are repeated until no new candidates are generated or time expires. The best distribution policy seen thus far (i.e. D_{best}) is returned as the answer. We now describe these two steps in more detail.

A. Evaluation

In a shared-nothing parallel DBMS like Greenplum [11], joins and vector-aggregates are primary catalysts for data movement during query execution. When two relations are to be joined, then their distributions need to be compatible - this may result in movement of data on either side of the join depending on their distributions. Similarly, computation of vector-aggregates may require data movement to group rows correctly. The intuition behind evaluation is this: by analyzing the query plans produced for a candidate distribution policy D_c, we can associate data movement with specific attribute sets involved in joins or vector-aggregates. We can use this information to generate new candidate distributions.

Input: Q, D_c
Output: $\mathfrak{M}$
foreach $Q_h \in Q$ **do**
  $Plan \leftarrow OptimizeQuery(Q_h, D_c)$;
  foreach $node \in Plan$ **do**
    if *node is Join* **then**
      $(T_1, X_1, B_1) = extractDM(node.left)$;
      $\mathfrak{M}(T_1, X_1) += B_1$;
      $(T_2, X_2, B_2) = extractDM(node.right)$;
      $\mathfrak{M}(T_2, X_2) += B_2$;
    end
    if *node is Aggregate* **then**
      $(T_1, X_1, B_1) = extractDM(node.left)$;
      $\mathfrak{M}(T_1, X_1) += B_1$;
    end
  end
end
return $\mathfrak{M}$;

Fig. 3. Evaluation step calculates $\mathfrak{M}$ for distribution policy D_c

Candidate evaluation involves building a summary structure called $\mathfrak{M}$ (for data movement). This structure captures potential savings in data movement for different distributions and is organized as a key-value structure. The key is the pair $(Table, AttributeSet)$ and the value is $bytes$ of data moved. This structure is built up by iterating over all queries for the given distribution policy D_c (see Algorithm 3). Specifically, it looks for join (and aggregate) operators and extracts the amount of data that is moved before the join (and aggregation) is performed. This amount of data is associated with the join (and aggregation) attributes. We illustrate the working of Algorithm 3 using an example.

Example 1: Consider tables from the TPC-H benchmark [10]: $Part$ (distributed by $\{p\_brand\}$), $Orders$ (distributed by $\{o\_custkey\}$), $Supplier$ (distributed by $\{s\_name\}$) and $Lineitem$ (distributed by $\{l\_partkey\}$). The workload consists for three simple join queries $Q = \{Q_1, Q_2, Q_3\}$. For this distribution policy, the resulting query plans for the queries in Q are represented in Figure 2. As the algorithm in Figure 3 analyzes the plan for $Q1$ (Figure 2(a)), it extracts the information that no rows of $Lineitem$ were moved and 4 KB of Part were moved for the join on partkey attributes. Therefore, it updates $\mathfrak{M}(Lineitem, \{l\_partkey\}) = 0$ and $\mathfrak{M}(Part, \{p\_partkey\}) = 4KB$. Similar analysis of other query plans produces the final $\mathfrak{M}$ structure shown in Table I.

TABLE I
FINAL $\mathfrak{M}$ AFTER ANALYZING PLANS IN FIGURE 2

Key (Table, AttributeSet)	Value (bytes)
$(Part, \{p\_partkey\})$	4KB
$(Lineitem, \{l\_partkey\})$	0KB
$(Lineitem, \{l\_orderkey\})$	1MB
$(Lineitem, \{l\_suppkey\})$	2MB
$(Orders, \{o\_orderkey\})$	40KB
$(Supplier, \{s\_suppkey\})$	8KB

B. Generation

The generation step analyzes the $\mathfrak{M}$ structure to determine other interesting distribution policies to evaluate. For every table in the database, FINDER finds the top-K attribute set that *caused* the most data movement from the $\mathfrak{M}$ structure and creates additional distribution policies for consideration in the next iteration by adding them to $\mathfrak{D}$. K is a configurable parameter for the algorithm. For the $\mathfrak{M}$ in Table I, this would mean that the next distribution policy to explore would be $Part = \{p\_partkey\}$, $Lineitem = \{l\_suppkey\}$, $Orders = \{o\_orderkey\}$ and $Supplier = \{s\_suppkey\}$ for $K = 1$. Also see Section IV-D for other extensions that may be used in this step.

C. Complexity

The time complexity of FINDER comes from $\mathfrak{M}$ calculation and new distribution policy generation. The $\mathfrak{M}$ calculation is based on analyzing the aggregate and join nodes in each query plan. If a_h and j_h represent the number of aggregates and joins in a query Q_h, calculating $\mathfrak{M}$ in a single iteration employs $O(\sum (a_h + j_h))$ operations. FINDER may execute a user-determined number of iterations β. The $\mathfrak{M}$ structure is created on a per iteration basis and has utmost as many entries as the number of aggregates and joins. Thus, $O(\sum (a_h + j_h))$ is the space complexity of FINDER.

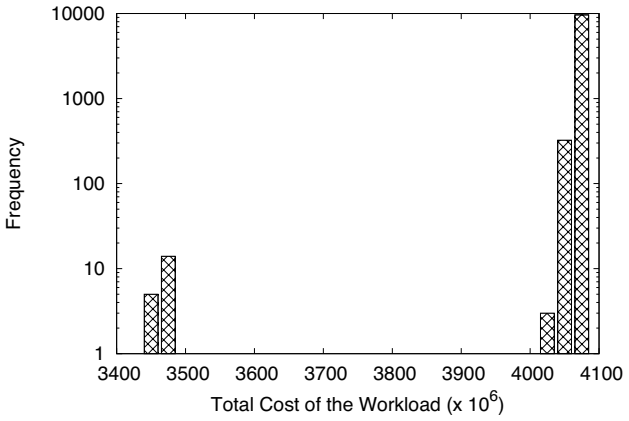

Fig. 4. Cost Distribution of Workloads using 10,000 Randomly Generated Distribution Policies

D. Extensions

The generation step in FINDER considers the top-K attribute sets for every table. For low K values and certain workloads, FINDER may get stuck in local minima. To alleviate this problem, it is possible to use techniques such as simulated annealing and genetic mutation. FINDER can maintain a temperature that models the threshold of the minimum data movement to consider per table. This threshold is reduced every iteration. Thus, earlier iterations produce more candidates distribution policies that are added to $\mathfrak{D}$. It is also possible to mutate the best solution by randomly swapping attributes out of the best solution to create new candidates that are not naturally highlighted in the evaluation step.

V. EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental evaluation, which includes the sampling the solution space, analysis of our proposed FINDER approach and the performance comparisons against existing techniques. The Greenplum Database (*GPDB*) instance has a single master node and 16 segment servers. The study focused on the well-known TPC-H synthetic data set and its workload. We present experiments with 500 GB TPC-H data sets.

A. Sampling the Solution Space

To demonstrate the effects of picking a poor distribution policy, we sampled the solution space by generating 10,000 randomly generated distribution policies. Figure 4 shows the distribution of the total workload cost of these randomly generated policies. For each randomly generated policy, we measure the total cost of the TPC-H workload. We see in Figure 4 only a small percentage of distribution policies (19 out of 10,000 randomly generated distribution policies) are close to the best

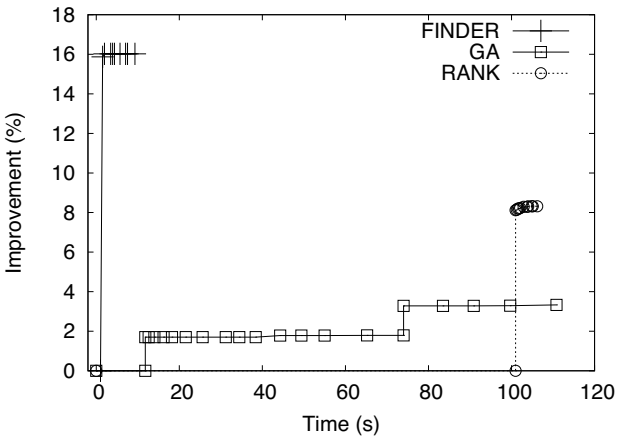

Fig. 5. Comparing Solutions Generated over Time

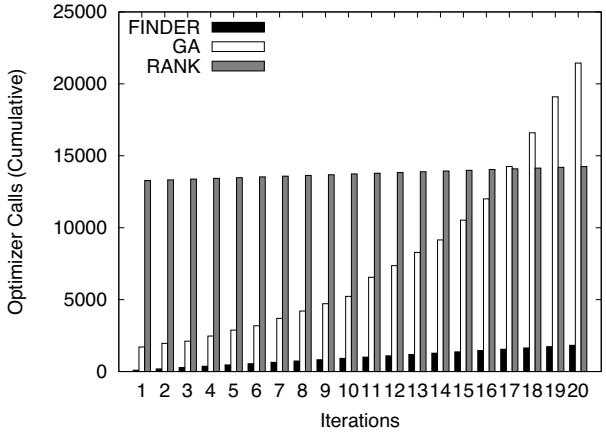

Fig. 6. Number of Optimizer Calls Performed in Each Iteration of the Algorithm

scenario. Moreover, most of sample distribution policy will result in plans whose costs are 1.18 times the cost of the workload when the best sampled distribution policy is chosen.

B. Comparison With Existing Techniques

In our study, we compared the distribution policies recommended by existing techniques such as *genetic approaches* (GA) [17] and *rank-based* (RANK) [13] technique. We however did not compare our work against MESA [12] since the later is deeply integrated with the Microsoft SQL Server 2008 Parallel Data Warehouse optimizer.

TABLE II
EXPERIMENTAL PARAMETERS

Algorithm	Parameter	Value
GA	Population	30
	Selection	0.20
	Crossover	0.10
	Mutation	0.10
RANK	# of distribution policies tested per query	20

Table II summarizes the parameters used during in our experimental evaluation for each algorithm. All of the compared techniques were implemented in JAVA as a recommendation layer on top of the GPDB instance. In addition, we implemented tools by which the techniques can interact with the GPDB query optimizer and extract statistical and distribution information from the database instance.

In Figure 5 we compare the distribution policies recommended by the different techniques over time. The results presented in Figure 5 is the average of 20 runs where for each run, the starting point of all three algorithms is the same randomly chosen distribution policy. The initial distribution policy is termed henceforth as the *baseline* solution. Both *GA* and *RANK* algorithms iteratively pick a better solution and over time recommend a distribution policy that is at most 3.33% and 8.32% better than the *baseline*. In contrast, FINDER performs a quick evaluation of the initial distribution policy. After the second iteration, FINDER takes the distribution policy that exhibited the maximum data movement to converge to a good solution. FINDER is able to recommend a distribution strategy that is 16% better than the *baseline* policy in under 10 seconds.

Figure 6 reports the cumulative number of optimizer calls each algorithm performs over every iteration to evaluate the candidate distribution policies as well as obtain statistics. Figures 5 and 6 show that the initial analysis conducted by *RANK* is an expensive operation since it requires the testing of several distribution policies for each individual query in the TPC-H workload. After the initial analysis, it iteratively finds policies by combining the different candidate solutions for each table. These checks can be performed with a fewer calls to optimizer in comparison to initial analysis by *RANK*. Over time the population of candidate solutions maintained by the *GA* increases resulting in an step growth in its number of optimizer calls. In contrast, by investigating the estimated data movement in the plans generated, FINDER is able to arrive at a good solution with relatively few calls to the optimizer.

The performance results observed by our implementation of these algorithms are consistent with those reported in [12]. We observed similar performance characteristics when we altered the baseline distribution policy from a random set of attributes to primary and foreign keys.

C. Analysis of FINDER

During each iteration, FINDER maintains the top-K attribute sets to be considered to be further investigation for each table in the database. This allows FINDER to avoid exploding the search space. Limiting the number of candidate attribute sets can however affect the solution generated by the algorithm. To study the effects of *K*, we varied K, a user controlled variable, in the range of 1 to 7. For TPC-H workload, we believe that our data movement based approach is effective in finding a good solution when *k* is small ($= 3$). Augmenting k to 7 increases the runtime of the algorithm, however it returns a distribution policy with an identical execution cost to the policy recommended when $k = 3$.

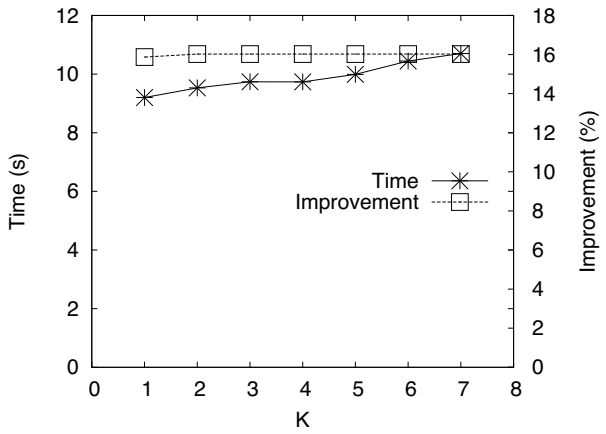

Fig. 7. Varying K, the # of attribute sets to consider in each iteration

TABLE III
PROFILING FINDER

FINDER Task	Percentage of Time Spent
Evaluation	96.96
Generation	3.04

Table III shows the distribution of time for the two steps in the algorithm. In our experiments, most of the time is spent in evaluating new distribution policies. This is expected because the TPC-H workload contains several join and aggregates that can cause data movement. The generation step is relatively inexpensive as we are able to find a good solution in relatively few iterations in contrast with other techniques.

VI. FINDER IN PRACTICE

One of the foremost design goals of FINDER is portability, i.e., the ability to run the tool against different MPP databases. In order to use FINDER with other database systems the following functionality is required:

- *Cost estimate per query*
  All major commercial database systems offer some functionality to access the query plan and its estimated cost.
- *Information regarding data movement per query*
  Again, this information is incorporated in the query plan including information such as the exact columns that were used for the data movement, as applicable.
- What-if *interface to alter distribution policies*
  Greenplum Database exposes this information via the system catalog which may be altered given sufficient privileges. Thus, evaluating different distribution policies can be accomplished without actually redistributing potentially large amounts of data.

Greenplum Database makes this functionality readily available, other database systems provide similar facilities which allows for a rather straight-forward deployment of FINDER to these systems.

VII. CONCLUSIONS

The problem of determining data placement is an important one in shared-nothing MPP database systems. We presented FINDER, a system that aims at finding the optimal distribution policy for a set of tables given a target workload. FINDER is based on the intuition that primary catalysts for data movement in a MPP system are joins and aggregates. FINDER analyzes query execution plans and finds interesting attribute sets from joins and aggregations and considers them in generation of candidate distribution policies. FINDER is designed to run as a third party tool and interacts with the optimizer in a *what-if* mode. FINDER is extremely lightweight and is shown to identify the best distribution policies much faster than other approaches.

Future work includes determining an order of evaluation for candidates and considering replicas of tables with different distribution policies.

Acknowledgements

We would like to thank Ravi Balasubramanian for his assistance in conducting the experimental evaluation.

REFERENCES

[1] The Vertica Analytic Database - Technical Overview White Paper. Technical report, 2010.
[2] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *Proceedings of SIGMOD*, pages 683–694, 2006.
[3] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *Proceedings of SIGMOD*, pages 359–370, 2004.
[4] I. Alagiannis, D. Dash, K. Schnaitter, A. Ailamaki, and N. Polyzotis. An Automated, Yet Interactive and Portable DB Designer. In *Proceedings of SIGMOD*, pages 1183–1186, 2010.
[5] J. Blakeley. Microsoft SQL Server Parallel Data Warehouse Architecture Overview. In *BIRTE*, 2011.
[6] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of SIGMOD*, pages 227–238, 2005.
[7] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *Proceedings of ICDE*, pages 826–835, 2007.
[8] N. Bruno and S. Chaudhuri. Constrained Physical Design Tuning. *PVLDB*, 1(1):4–15, 2008.
[9] J. Cohen, J. Eshleman, B. Hagenbuch, J. Kent, C. Pedrotti, G. Sherry, and F. Waas. Online Expansion of Largescale Data Warehouses. *PVLDB*, 4(11):1249–1259, 2011.
[10] J. Gray. TPC Benchmark^{TM} A: Standard Specification. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.
[11] Greenplum. http://www.greenplum.com/.
[12] R. Nehme and N. Bruno. Automated Partitioning Design in Parallel Database Systems. In *Proceedings of SIGMOD*, pages 1137–1148, 2011.
[13] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating Physical Database Design in a Parallel Database. In *Proceedings of SIGMOD*, pages 558–569, 2002.
[14] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-Line Index Selection for Shifting Workloads. In *ICDE Workshops*, pages 459–468, 2007.
[15] Teradata. http://www.teradata.com/.
[16] D. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1997.
[17] D. Zilio, A. Jhingran, and S. Padmanabhan. *Partitioning Key Selection for a Shared-nothing Parallel Database System*. IBM Research Report RC 19820 (87739) 11/10/94, 1994.