A New Tool for Multi-Level Partitioning in Teradata

Young-Kyoon Suh*
Department of Computer Science
University of Arizona
Tucson, AZ, 85721
yksuh@cs.arizona.edu

Ahmad Ghazal, Alain Crolotte,
Pekka Kostamaa
Teradata Corporation
El Segundo, CA, 90245
{ahmad.ghazal, alain.crolotte,
pekka.kostamaa}@teradata.com

ABSTRACT

This paper introduces a new tool that recommends an optimized partitioning solution called Multi-Level Partitioned Primary Index (MLPPI) for a fact table based on the queries in the workload. The tool implements a new technique using a greedy algorithm for search space enumeration. The space is driven by predicates in the queries. This technique fits very well the Teradata MLPPI scheme, as it is based on a general framework using general expressions, ranges and case expressions for partition definitions. The cost model implemented in the tool is based on the Teradata optimizer, and it is used to prune the search space for reaching a final solution. The tool resides completely on the client, and interfaces the database through APIs as opposed to previous work that requires optimizer code extension. The APIs are used to simplify the workload queries, and to capture fact table predicates and costs necessary to make the recommendation. The predicate-driven method implemented by the tool is general, and it can be applied to any clustering or partitioning scheme based on simple field expressions or complex SQL predicates. Experimental results given a particular workload will show that the recommendation from the tool outperforms a human expert. The experiments also show that the solution is scalable both with the workload complexity and the size of the fact table.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Star Schema, Fact Table, Multi-Level Partitioning

*This work was done while Young-Kyoon Suh was at Teradata Corporation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
Copyright 2012 ACM 978-1-4503-1156-4/12/10 ...$15.00.

1. INTRODUCTION

Tables and materialized views in Teradata are hash distributed based on a user-specified column or set of columns called *primary index*. Each virtual processor (unit of parallelism called *AMP* in Teradata) receives a subset of the data and stores it in hash order. Teradata users typically choose primary index fields, so that the data is evenly distributed among the AMPs. The primary index fields are also chosen to reflect join fields in workloads to accomplish cheaper local joins that do not require data shuffling.

PPI (Partitioned Primary Index) [8] is an optional horizontal partitioning scheme applied locally on each AMP's data. Note that in other database products, this type of partitioning would most likely be called *clustering*. For the rest paper, we only use the term *partitioning*, not clustering. Database Administrators (DBAs) usually choose the columns for PPI, based on join fields and single table predicates to optimize the queries in the workload. The PPI columns are used to physically cluster data with the same values together in contiguous data blocks. This allows partition elimination in scans and joins, which improve query performance. PPI can be specified as single or multiple (or nested) levels. This type of partitioning scheme is referred to as Multi-Level PPI or *MLPPI* [3].

By allowing non-qualified partitions to be eliminated, MLPPI can reduce significantly the amount of data to be scanned to answer a query. However, a large number of partitions can create significant overhead especially in joins and table maintenance operations such as inserts or deletes, so that the selection of partitions is usually a balancing act.

Now, let us see how the DBAs can define an MLPPI on the fact table from the Star Schema Benchmark (SSB) [6]. The SSB [6] is an extension of the TPC-H [9] benchmark, and it provides a Star Schema based on a retail model. The fact table `LINEORDER` is representative of orders along with their associated line items. There are four dimension tables covering date, customers, parts and suppliers.

Figure 1 represents a query set Q consisting of only two queries $q1$ and $q2$ based on the SSB. Our focus is on providing an MLPPI solution for the fact table in the star schema. Based on $q1$ and $q2$ in Figure 1, there are five single table constraints on `LINEORDER`, as listed in Figure 2; the constraints are identified by the query number and the sequence number of the constraint in each query in Q.

The DBAs need to look at partitioning options based only on the two fields and a total of the five constraints. Again, the constraints of interest are for the fact table. There are many possibilities from a *fine-grained* partition set to

```
q1: SELECT SUM(l.LO_EXTENDEDPRICE*l.LO_DISCOUNT)
    FROM LINEORDER l, DDATE d
    WHERE l.LO_ORDERDATE = d.D_DATEKEY
    AND d.D_YEAR = '1993'
    AND l.LO_DISCOUNT IN (1, 4, 5)
    AND l.LO_QUANTITY <= 30
q2: SELECT c.C_NATION, SUM(l.LO_REVENUE)
    FROM CUSTOMER c, LINEORDER l
    WHERE l.LO_CUSTKEY = c.LO_CUSTKEY
    AND c.C_REGION='EUROPE'
    AND l.LO_DISCOUNT >= 7
    AND l.LO_QUANTITY >= 25 AND l.LO_QUANTITY <= 35
    GROUP BY c.C_NATION
    ORDER BY revenue desc
```

Figure 1: Star Schema Workload Q

$q1.1$	LO_DISCOUNT IN (1,4,5)
$q1.2$	LO_QUANTITY <= 30
$q2.1$	LO_DISCOUNT >= 7
$q2.2$	LO_QUANTITY >= 25
$q2.3$	LO_QUANTITY <= 35

Figure 2: Single Table Constraints on LINEORDER

a loser definition. Considering for the time being the column LO_DISCOUNT, for instance, one solution could be to identify each value for the IN predicate in $q1.1$ and use $q2.1$ as is. This yields the following partitioning expression for LO_DISCOUNT :

```
CASE_N(
  LO_DISCOUNT = 1,
  LO_DISCOUNT = 4,
  LO_DISCOUNT = 5,
  LO_DISCOUNT >= 7,
  NO CASE OR UNKNOWN
)
```

The above expression minimizes the size of the partitions by focusing exactly on the values required to satisfy the constraints, but creates a large number of small partitions.

Another possibility would be to look at the maximum and minimum values in the IN set and to build an AND clause equivalent to a between clause yielding the following:

```
CASE_N(
  LO_DISCOUNT >= 1 AND LO_DISCOUNT <= 5,
  LO_DISCOUNT >= 7,
  NO CASE OR UNKNOWN
)
```

The above partitioning solution decreases the number of partitions, compared to the previous one, but still focuses sharply on the constraints with still relatively small partitions. Another possibility would be to make only two partitions by using the minimum and maximum values of the overall constraints on LO_DISCOUNT as shown below:

```
CASE_N(
  LO_DISCOUNT >= 1 AND LO_DISCOUNT <= 7,
  NO CASE OR UNKNOWN
)
```

Similar considerations apply to the partitioning associated with LO_QUANTITY this time with ranges and covering problems difficult to deal with. The resulting partitioning scheme for the table includes both LO_DISCOUNT and LO_QUANTITY, so that the number of possible combinations is the product

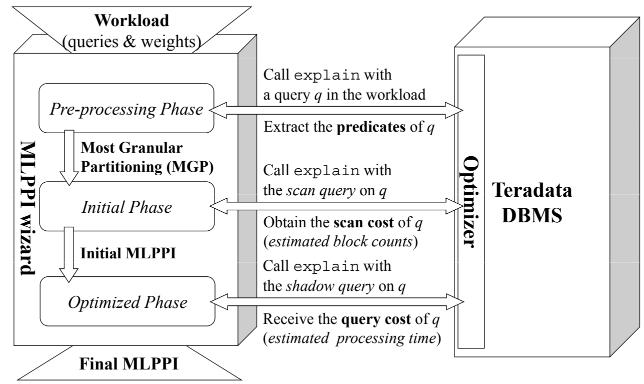

Figure 3: MLPPI Wizard Architecture

of the potential combinations for each field. Also, there are two queries, and one partitioning scheme may be better for one query or the other. As a result, the DBAs will be faced with a daunting combinatorial search problem and no clear basis to decide on which combination is the best. This state of affairs begs for a *tool* to assist the DBAs.

In the sequel, we introduce an automated tool, called *MLPPI wizard*. Indeed, the wizard is the first physical database design tool developed at Teradata. The tool is based on a novel technique using a greedy algorithm for search space enumeration. This tool based on a general framework allowing general expressions, ranges and case expressions for partition definitions is particularly well-suited for MLPPI definition. The *predicate-driven* technique used by the tool can be applied to any clustering or partitioning based on simple fields/expressions or complex SQL predicates. In addition, the wizard borrows the Teradata DBMS optimizer cost model, which is used to prune the search space for reaching a final solution.

Figure 3 illustrates how the tool recommends the final MLPPI customized for a given workload consisting of a set of queries and corresponding weights. The wizard goes through a series of phases: *Pre-processing Phase*, *Initial Phase*, *Optimized Phase*. We will provide more details about each phase in Section 2, using the query set Q in Figure 1 in the rest of the paper.

To the best of our knowledge, there is no existing industrial tool to solve the **multi-level partitioning** problem except our wizard. The wizard is contained completely on the client side, as opposed to previous work [1, 4, 5] requiring optimizer code extension. Our tool simply leverages existing APIs to simplify the queries and to capture fact table predicates and costs, and uses these items to make a recommendation. This makes the wizard extensible and portable to different releases of the Teradata DBMS server.

The paper is organized in the following way. Section 2 presents the algorithms for the wizard. Section 3 reports the performance evaluation results. Section 4 provides related work, and Section 5 concludes the paper.

2. THE MLPPI WIZARD

In this section, we describe each phase of the wizard in more details, using the query set Q provided in Figure 1.

2.1 Pre-processing Phase

Algorithm 1: Pre-processing Phase

```
  input  : Q (input query set)
  output: R (non-overlapping range set), M
          (query-to-range-set map)
1   Query Simplification (on Q)
2   Range Extraction (from Q)
3   Non-Overlapping Range (R) Construction
4   Query-to-Range-Set Map (M) Construction
5   Partition Count Limit Check (on R)
```

Algorithm 1 represents the pre-processing phase. First, we simplify the predicate(s) of each query by removing redundant conditions (in line 1). In the running example of Q, this simplification is not needed, as the queries are neat. Next, the tool locates the fact table fields in the simplified predicates, extracts from the predicates, and groups the ranges along with each field (in line 2). The ranges, for example, can be derived from the predicates in Figure 2. The extracted whole range set then becomes a kind of two-dimensional array, called R, in which each entry under a field represents a pair of (start and end) values forming the range. In the continuing example, the range set R is formed by the entries below:

R :

LO_DISCOUNT	LO_QUANTITY
$[1, 1]$	$(\infty, 30]$
$[4, 5]$	$[25, 35]$
$[7, \infty)$	

Any overlapping ranges found in R are broken into a pair of consecutive, non-overlapping ranges for a merge in further phases (in line 3). In the example, the split is applied on the overlapping ranges - $(\infty, 30]$, $[25, 35]$ - under LO_QUANTITY, and after that, their common range - $[25, 30]$ - is inserted into R. Then, R will be like as below:

R :

LO_DISCOUNT	LO_QUANTITY
$[1, 1]$	$(\infty, 25)$
$[4, 5]$	$[25, 30]$
$[7, \infty)$	$[31, 35]$

The tool in turn creates a bi-directional *query-to-range-set* map, called M, between Q and R (in line 4), as shown below:

M :

$\langle q1, \{R[1, 1], R[1, 2], R[2, 1], R[2, 2]\}\rangle$
$\langle q2, \{R[1, 3], R[2, 2], R[2, 3]\}\rangle$,

where $R[i, j]$ denotes the interval value for the i-th field and j-th range in R.

R may be used to define an MLPPI using each field as one level, as R produces a total of 16 partitions (= # ranges in LO_DISCOUNT × # ranges in LO_QUANTITY) below the partition count limit (65,536) allowed in an MLPPI (in line 5). Note that each field counts one more range corresponding to the "NO CASE OR UNKNOWN". In the running example, indeed, the tool can immediately enter the optimized phase. However, to continue our discussion, let us assume that the limit is *15* in the paper. Then, the tool needs to proceed to the initial phase, so that the total partitions by R can drop below the limit.

2.2 Initial Phase

In this phase, we will reach a feasible MLPPI solution by incrementally merging a range pair in R.

By and large, overall I/O cost may be increased by the merge. It is because when answering a query, from the merged partition we may read non-qualified rows that would not be found before the merge, thereby paying more I/O. For that reason, we will pick up and merge the range pair incurring the least I/O cost for all queries.

Algorithm 2: Initial Phase

```
   input  : M (query-to-range-set map), R (input range
            set)
   output: R with # partitions ≤ partition limit
1  while (# partitions by R > partition limit) do
2  |  foreach range pair (rp) in R do
3  |  |  Ts ← 0 // Tq:  The total scan cost on rp
4  |  |  foreach q in M do
5  |  |  |  L ← Get the range set mapped to q in M
6  |  |  |  if (rp ∩ L) ≠ ∅ then       /* Affected */
7  |  |  |  |  Ts += wq·computeScanCost(q,rp,L)
   |  |  |  |     // wq:  q's weight
8  |  |  |  else Ts += wq·scq // scq:  q's existing
   |  |  |     scan cost
9  |  |  end
10 |  |  Record Ts with rp.
11 |  end
12 |  Update M and R along with rp with the least Ts.
13 end
14 return R;
```

Algorithm 2 describes the initial phase. Given M and R, partitions get incrementally reduced, as we consolidate the range pair producing the least I/O when merged. While R has more partitions than the limit (in line 1), for each range pair (rp) in R, the wizard examines whether each query (q) is affected by the merge of rp (in lines 2-6). To do so, the tool 1) gets the range set (L) mapped to q from M, and 2) sees if L includes either ranges of rp (lines 5-6). If q gets affected by the merge of rp, we obtain the *scan cost* of q by sending a scan cost query (s) of q to the server via an API call (in line 7). s can be built by 1) copying L into L', 2) removing and merging the ranges of rp from L', 3) adding the merged range in L', and 4) restoring the equivalent conjuncts (C) from L', and 5) constructing a full scan query like '`SELECT * FROM` F `WHERE` C'. Thereafter, the tool extracts the *estimated block counts* from the result of s, and uses it as the *scan cost* of q. If the merge of rp does not affect q, then the previously computed q's scan cost is recycled to avoid the expensive API call (in line 8).

In this way, we can compute the total I/O cost (T_s) incurred when rp is merged, by adding up the weighted scan costs of queries (in line 10). Once all range pairs are examined, we pick up rp with the least T_s for a merge. In the example, suppose that the range pair of $R[2, 2]$ and $R[2, 3]$ produces the least T_s. Then, the ranges are merged, and M and R are accordingly updated along with the chosen range pair (in line 12) as shown below:

R :

LO_DISCOUNT	LO_QUANTITY
$[1, 1]$	$(\infty, 25)$
$[4, 5]$	$[25, 35]$
$[7, \infty)$	

M:

$\langle q1, \{R[1, 1], R[1, 2], R[2, 1], R[2, 2]\}\rangle$
$\langle q2, \{R[1, 3], R[2, 2]\}\rangle$

We repeat the above steps until the number of partitions in R drops below the maximum number of partitions that can be defined in an MLPPI. Now that the total partitions (12=4×3) by R are under the assumed limit (15), we face the optimized phase, carrying the up-to-date M and R.

2.3 Optimized Phase

In this regard, we can have an *initial* MLPPI recommendation for the `LINEORDER` fact table, using R. However, having fewer partitions may enhance the overall workload performance, as the Teradata query optimizer has an operational threshold within which it can handle partitions simultaneously. Hence, we attempt to incrementally improve the initial MLPPI solution by further merging partitions in R.

Algorithm 3: Optimized Phase

input : M (query-to-range-set map), R (input range set)
output: MLPPI such that the total query cost is minimal

1. $T_p \leftarrow$ The total query cost obtained using the initial MLPPI by R
2. **while** *(Pre-defined iterations or* $R \neq \varnothing$*)* **do**
3. | **foreach** *range pair* rp *in* R **do**
4. | | $T_q \leftarrow 0$ `//` T_q: `The total query cost on` rp
5. | | **foreach** q *in* M **do**
6. | | | $L \leftarrow$ Get the range set mapped to q in M
7. | | | **if** $(rp \cap L) \neq \varnothing$ **then** `/* Affected */`
8. | | | | T_s += $w_q \cdot computeQueryCost(q,rp,M,R)$
9. | | | **else** T_q += $w_q \cdot qc_q$ `//` qc_q: q`'s existing query cost`
10. | | **end**
11. | | Map T_q to rp.
12. | **end**
13. | $T \leftarrow$ Find the least T_q
14. | **if** $T \leq T_p$ **then** Reduce T_p to T, and update M and R along with rp yielding T.
15. | **else break**
16. **end**
17. **return** an (optimized) MLPPI by the current R ;

Algorithm 3 describes the optimized phase. We first obtain the weighted *query cost* sum (T_p) of Q using an initial MLPPI by R (in line 1). (The way of obtaining the query cost will be shortly described.) Next, until the given number of iterations reaches, or R has no ranges (in line 2), for each range pair (rp) in R, the tool computes the weighted query cost sum (T_q) on rp (in lines 3-12). If q is affected by the merge of rp (in line 7), the query cost of q, instead of the scan cost of q, is (re)computed and added to T_q (in line 8). To get the query cost, we reflect the merge of rp into M and then update R. After that, via an API call, the tool requests the server to return the query cost - *estimated processing time* - of q on the fact table supposedly applying an MLPPI definition using R. (This sounds similar to the "*what-if*" mode proposed in previous literature [2].) Unless the merge affects q, the existing query cost of q is recycled, and its weighted cost is added to T_q (in line 9). Once all range pairs are examined, the tool finds the least T_q (T) (with the range pair producing T) (in line 13). If T is less than or equal to T_p, then the tool reduces T_p to T, updates M and R along with the range pair with T, and repeats the iteration (in line 14). Otherwise, the tool finishes the phase, recommending an (optimized) MLPPI solution using the most up-to-date R (in lines 15-17). In the running example, suppose that in the first iteration ranges $R[1, 1]$ and $R[1, 2]$ are chosen and merged, but in the next iteration there is no further merge that can reduce T_p. Then, we can see the final MLPPI recommendation for Q, as shown below.

```
ALTER TABLE LINEORDER
MODIFY PRIMARY INDEX
PARTITION BY(
  CASE_N(
    LO_DISCOUNT ≥ 1 AND LO_DISCOUNT ≤ 5,
    LO_DISCOUNT ≥ 7,
    NO CASE OR UNKNOWN),
  CASE_N(
    LO_QUANTITY < 25,
    LO_QUANTITY ≥ 25 AND LO_QUANTITY ≤ 35,
    NO CASE OR UNKNOWN)
);
```

3. EXPERIMENTS

In this section, we present our performance evaluation results. Section 3.1 describes our environment settings - the workload, software and hardware used. Section 3.2 shows the actual results of our solution, and compares them with those of no partitioning and typical solutions by DBAs.

3.1 Environment Settings

The performance evaluation was done on the Teradata DBMS server machine running Unix. We generated two workloads consisting of 10 queries (10Q) or 20 queries (20Q) for the experiments. Each query is based on a template that joins the `LINEORDER` fact table and the `DDATE` dimension one with constraints defined by the predicates. This template is common in customer cases like reports and form templates. We ran the above 10Q and 20Q query sets on two size variants of the fact table `LINEORDER` in the SSB data model. In the first case, `LINEORDER` is loaded with 1 Terabyte of data (1TB), and in the second case it is populated with 3 Terabytes of data (3TB).

3.2 Performance Evaluation

We compared the performance of our partitioning solutions (WIZARD) with the other solutions - no partitioning (NOPPI) and the partitioning (EXP) by a human expert. We obtained EXP and WIZARD solutions for 10Q and 20Q on the populated 1TB and 3TB tables. In turn, we applied the corresponding EXP and WIZARD solutions to, and then ran 10Q and 20Q on the altered tables. For the NOPPI solution, we just executed 10Q and 20Q on the raw 1TB and 3TB fact tables with no MLPPI.

Figure 4 represents the measured total elapsed times (in secs) of 10Q and 20Q on each of the fact tables. As shown in Figure 4(a), we observe in our experiments on the 1TB table that WIZARD shows around 4.36x and 2.30x (running time) improvements on 10Q, and 4.34x and 1.85x on 20Q, compared with those of NOPPI and EXP, respectively. The improvements still maintained in spite of doubly increased workload size (from 10Q to 20Q). Even in the experiment

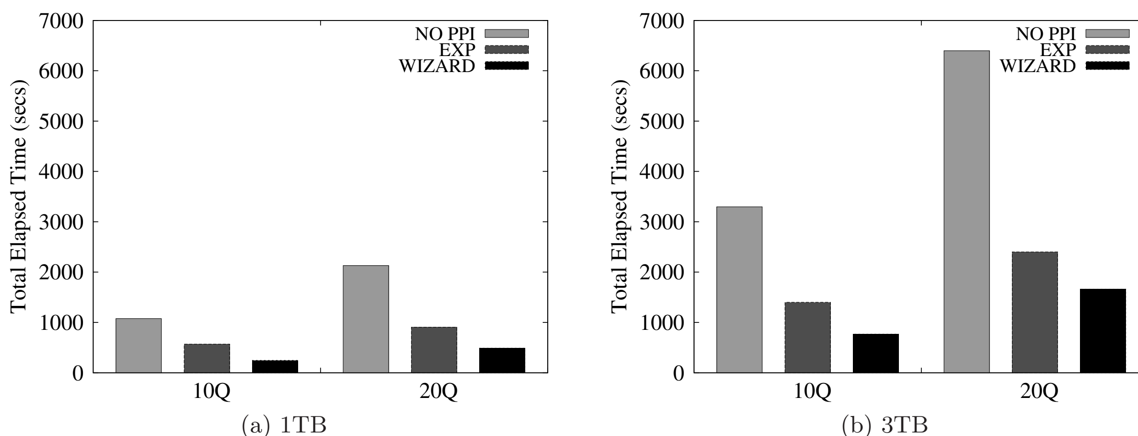

Figure 4: Total Elapsed Time

using the 3TB fact table, the WIZARD solutions yielded about 4.29x and 1.82x (running time) speedups on 10Q, and 3.85x and 1.44x on 20Q in comparison with those of the NOPPI and EXP ones, as seen in Figure 4(b). Although the size of the fact table became 3x larger, the performance of WIZARD did not suffer from any degradation.

4. RELATED WORK

The MLPPI wizard seems similar to some DBMS vendors' tools from Oracle [7], IBM DB2 [4], and MS SQL Server [1]. However, there is no published technical detail about Oracle Partitioning Advisor [7], and as far as the other tools are concerned, our wizard is fundamentally different from them in light of problem scope and approach.

DB2 MDC Advisor [4] actually tackles a different problem of automatically recommending the most well-suited MDC keys for a given workload. Agrawal's work [1] discusses another problem of merging *single-level range partitionings* on objects such as tables or indexes. This paper, however, addresses the *multi-level partitioning* problem. Hence, the existing solutions cannot be directly applied to our wizard.

Regarding the approach, DB2 MDC Advisor [4] uses the search space driven by *fields*. Meanwhile, our search space is driven by *query-predicates*, which is superior because predicates are more *customized* to a workload. Agrawal's scheme [1] also uses the search space driven by simple range predicates. However, his approach cannot reach a globally-optimized solution, as his work first produces a solution customized to each query and then consolidates solutions between queries. On the other hand, our wizard generates the whole search space upfront and subsequently merges partitions, leading to a globally-optimized recommendation. Moreover, only a single column is considered in his work [1], whereas our tool deals with multiple fields.

Furthermore, implementations of previous tools [1, 4, 5] required instrumentation for optimizer code. These instrumentations are needed to facilitate the required information for the physical design tools API calls. The instrumentation code need to be enhanced and tested for new database releases which add complexity and additional cost for software upgrades. The Teradata optimizer has a rich set of existing APIs originally coded for system and workload management tools. These APIs are sufficient for the MLPPI functionality which helped us avoid the costly optimizer code change and instrumentation.

5. SUMMARY

Given workloads, it is very hard for DBAs to select the appropriate fields in partitioning the fact table due to large search space. To help relieve the DBAs' concern, we presented the Teradata MLPPI wizard that recommends a fact table multi-level partitioning solution customized for a workload. The tool derives its search space from the query predicates of the input workload, incrementally reduces the space by merging the range pair with the least scan or query cost, and eventually recommends an optimized MLPPI solution for the workload. We demonstrated that the produced MLPPI solutions by the wizard could improve the workload execution cost by up to about more than a factor of four and two, compared with those of no partitioning approach and partitioning done by a human expert, respectively. The MLPPI solutions of our wizard also scaled well with increasing workloads and fact table sizes.

6. ACKNOWLEDGMENTS

We would like to thank Prof. Bongki Moon for his insightful comments.

7. REFERENCES

[1] Agrawal, S. et. al. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD* (2004), ACM, pp. 359–370.

[2] Chaudhuri, S. et. al. AutoAdmin 'What-If' Index Analysis Utility. *SIGMOD Record 27*, 2 (1998), 367–378.

[3] Klindt, J. Single-level and Multilevel Partitioned Primary Indexes. http://www.teradata.com/white-papers/Single-level-and-Multilevel-Partitioned-Primary-Indexes-eb1889/, 2009.

[4] Lightstone, S. et. al. Automated Design of Multi-dimensional Clustering Tables for Relational Databases. In *VLDB* (2004), VLDB, pp. 1170–1181.

[5] Nehme, R. et. al. Automated Partitioning Design in Parallel Database Systems. In *SIGMOD* (2011), ACM, pp. 1137–1148.

[6] O'Neil, P. et. al. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/ poneil/StarSchemaB.PDF, 2007.

[7] Oracle. Partitioning Advisor. http://www.oracle.com/technetwork/database/options/partitioning/twp-partitioning-11gr2-2009-09-130569.pdf.

[8] Sinclair, P. Using PPIs to Improve Performance. http://www.teradata.com/tdmo/v08n03/pdf/AR5731.pdf, 2008.

[9] TPC-H. TPC-H. http://www.tpc.org/tpch/default.asp.