MLPPI Wizard: An Automated Multi-level Partitioning Tool on Analytical Workloads

Young-Kyoon Suh[1], Alain Crolotte[2], and Pekka Kostamaa[2]

[1] Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA
yksuh@cs.arizona.edu
[2] Teradata Corporation, El Segundo, CA 90245, USA
{alain.crolotte,pekka.kostamaa}@teradata.com

Abstract. Typically, it is a daunting task for a database administrator (DBA) to figure out how to partition a huge fact table accessed by query workloads for better performance. To relieve such a burden, we introduce an intelligent partitioning tool to recommend an optimized partitioning on the fact table. This tool uses a greedy algorithm for search space enumeration. This space is driven by predicates of a given query workload. The tool takes advantage of the cost model of a query optimizer to prune the search space. The tool resides completely on a client and interacts with the optimizer via APIs. Thus, there is no overhead to instrument the optimizer code. Furthermore, the predicate-driven method can be applied to any clustering or partitioning scheme. We show that the tool's recommendation outperforms a human expert's solution. We also demonstrate that the recommendation scale very well with increasing workload and growing fact table.

Keywords: Star Schema, Fact Table, Multi-Level Partitioning.

1. Introduction

Over the past decades much attention has been paid to improve the performance on analytical workloads in a data warehousing environment. Typically, a relational database management system (DBMS) has been exploited to process such analytical queries faster and assist its customers to make a timely business-decision in rapidly changing enterprise data warehousing. Teradata DBMS has been a leading database product in this data warehousing marketplace that has been increasingly more competitive than ever.

To optimize the performance of analytical processing, tables and materialized views in the Teradata DBMS are hash-distributed based on a user-specified column or set of columns called *primary index*. Each virtual processor, a unit of parallelism called *AMP* in the Teradata DBMS, receives a subset of the data and stores it in hash order.

Users typically choose primary index fields in the DBMS, 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) [19] 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 to be called *clustering*. In the rest article we use the term *partitioning* only, not clustering, to avoid confusion. Database administrators (DBAs) usually choose the columns for PPI, based on join fields and single table predicates to optimize the queries included in the workload. The PPI columns are used to physically cluster

```
CREATE TABLE LINEORDER(
  LO_ORDERKEY INTEGER,
  LO_QUANTITY INTEGER,
  LO_DISCOUNT INTEGER
)
PRIMARY INDEX ( LO_ORDERKEY )
PARTITION BY (
  CASE_N(
    LO_DISCOUNT >= 7,
    NO CASE OR UNKNOWN),
  CASE_N(
    LO_QUANTITY < 25,
    LO_QUANTITY >= 25 AND LO_QUANTITY <= 30,
    LO_QUANTITY > 30 AND LO_QUANTITY <= 35,
    NO CASE OR UNKNOWN)
);
```

Fig. 1. An Example of an MLPPI Table in the Teradata DBMS

data with the same values together in contiguous data blocks. This allows "partition elimination" in scans and joins for performance improvement. PPI can be specified as single or multiple (or nested) levels. This type of partitioning scheme is known as Multi-Level PPI (*MLPPI)* [9].

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

Figure 1 exemplifies an MLPPI table with potential partitioning schemes. This example is a subset of the `LINEORDER` table from the Star Schema Benchmark (SSB) [14]. Note that the full table with modified data types was actually used in our experiments.

In the definition of `LINEORDER` in the figure, the primary index is `LO_ORDERKEY`. The primary index dictates the AMP on which a row will be located, while the partitioning of data will be dictated by the values and ranges associated with `LO_DISCOUNT` and `LO_QUANTITY`. `LO_DISCOUNT`, for example, has two ranges for values greater than or equal to 7 and one (`NO CASE OR UNKNOWN`) for all the other values, while `LO_QUANTITY` has four ranges. As a result, the relational table will have a total of $2 \times 4 = 8$ partitions as follows.

Partition	Condition
1	`LO_DISCOUNT >= 7 && LO_QUANTITY < 25`
2	`LO_DISCOUNT >= 7 && 25 <= LO_QUANTITY <= 30`
3	`LO_DISCOUNT >= 7 && 30 < LO_QUANTITY <= 35`
4	`LO_DISCOUNT >= 7 && LO_QUANTITY no case`
5	`LO_DISCOUNT no case && LO_QUANTITY < 25`
6	`LO_DISCOUNT no case && 25 <= LO_QUANTITY < 30`
7	`LO_DISCOUNT no case && 30 < LO_QUANTITY <= 35`
8	`LO_DISCOUNT no case && LO_QUANTITY no case`

Whatever partition set is chosen, the selection must be semantically correct; namely, the mapping of rows to partitions must be an injection. In other words, the constraints must form a covering of the entire range, so that a row will belong to exactly one and one partition. The Teradata optimizer then applies partition elimination for queries that specify conditions on `LO_DISCOUNT` and/or `LO_QUANTITY`. For example, only partitions 1 and 2 are needed for the query "`SELECT * FROM LINEORDER WHERE LO_DISCOUNT >= 7 AND LO_QUANTITY <= 30`." Similarly, partitions 1 and 5 are sufficient to answer the query "`SELECT * FROM LINEORDER WHERE LO_QUANTITY < 25`." The respective sizes of the partitions are a factor of the data distribution. To see MLPPI in greater detail, check our references [19, 9].

Next, using the full version of `LINEORDER` as defined in SSB [14] we provide examples of partitioning schemes that a DBA may define based on a small query set. Consider a query set Q consisting of two queries $q1$ and $q2$ from the SSB set [14], as shown below.

```
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
```

Let's now focus on the `LINEORDER` fact table and the predicates involving `LINEORDER` fields only. There are five constraints identified by the query number and the sequence number of the constraint in each query, as illustrated below.

$q1.1$	`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`

At this point the DBA needs to consider options based only on two fields and the five constraints. There are many possibilities from a *fine-grained* partition set to a loser definition.

Considering for the time being the column `LO_DISCOUNT`, one solution is 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 is 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 solution but still focuses sharply on the constraints with still relatively small partitions.

Another possibility is to use the partitioning associated with `LO_DISCOUNT` shown in the beginning of this section with only two partitions.

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 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 DBA 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 DBA.

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 an MLPPI definition. The *predicate-driven* technique used by the tool can be applied to any clustering or partitioning based on simple fields and expressions or complex SQL predicates. The wizard also borrows the optimizer's cost model to prune the search space and reach a final solution.

Figure 2 illustrates how our tool recommends the final MLPPI customized for a given workload consisting of a set of queries and corresponding weights. The wizard passes through a series of phases: *Preprocessing*, *Initial*, and *Optimized* Phases. In the preprocessing phase the tool produces the most granular partitioning (MGP) based on the predicates collected from every query. In the initial phase the wizard refines the passed MGP by merging a pair of partitions with the least *scan cost* (as will be explained in Section 2.2) and then yields an initial (feasible) MLPPI based on the refined partitions. In the optimization phase, the tool makes a further attempt to optimize the initial MLPPI. We provide in greater detail each phase, using the query set Q consisting of $q1$ and $q2$ in the rest of the article.

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 [3, 11, 13] requiring optimizer code extension. Our tool simply uses existing APIs to simplify the queries, 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.

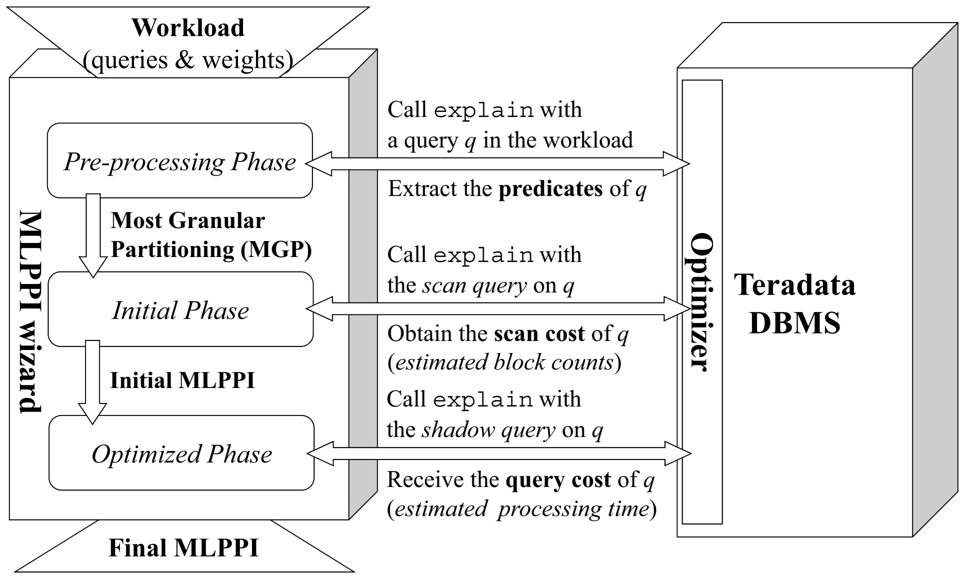


Fig. 2. The MLPPI wizard architecture

The article is a substantial extension of prior work [20]. The rest of this article is organized in the following way. In the following section we propose a multi-level partitioning algorithm—consisting of a series of phases—used by our MLPPI wizard, which is applicable to any clustering or partitioning scheme in an arbitrary DBMS. In turn, we conduct a detailed analysis of the complexity of the algorithms. Next, we report the performance evaluation results. We then elaborate on how our wizard is distinguished from several existing tools proposed by other DBMS vendors. Finally, we conclude this article by summarizing our discussion.

2. The MLPPI wizard

In this section we describe in detail each phase of the wizard using the query set Q provided in Section 1.

2.1. The Preprocessing Phase

Algorithm 1 sketches the steps in the preprocessing phase. This phase consists of a total of six steps.

Query Simplification The first step of this phase is to simplify the predicate(s) of each query by removing redundant conditions. For instance, if a query predicate has the condition of "`LO_DISCOUNT IN (1, 4, 5) AND LO_DISCOUNT IN (2, 3, 4, 5)`", then the predicate can be simplified as "`LO_DISCOUNT IN (4, 5)`." This can be done via an API call to the DBMS server. In the running example this simplification is not required.

Algorithm 1: The Preprocessing 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  Field Count Limit Check (on $R$)
5  Query-to-Range-Set Map ($M$) Construction
6  Partition Count Limit Check (on $R$)
```

Range Extraction The following step is to gather the simplified predicates and then obtain ranges from the collected predicates. This can also be done through an API call to the server. The query predicate is of the form

$$\langle \texttt{variable} \rangle \ \langle \texttt{op} \rangle \ \langle \texttt{constant} \rangle,$$

where $\langle \texttt{variable} \rangle$ is a field from LINEORDER, $\langle \texttt{op} \rangle$ is in

$$\{= \ , <, <=, >=, >, \texttt{IN}\},$$

and $\langle \texttt{constant} \rangle$ represents a constant value(s). All $\langle \texttt{op} \rangle$s are self-explanatory. In particular, 'IN' is a list predicate implying 'OR' operator. In the running example, we can collect a set of predicates $P = \{p1,p2,p3,p4\}$ from Q, where

$p1$: `l.LO_DISCOUNT IN (1, 4, 5)`
$p2$: `l.LO_DISCOUNT >= 7`
$p3$: `l.LO_QUANTITY <= 30`
$p4$: `l.LO_QUANTITY >= 25` AND `l.LO_QUANTITY <= 35`.

In turn, the wizard constructs a bi-directional *map* between a query and associated predicates, so that for each query the corresponding predicate(s) can be found directly in the map, and vice versa. In the example, a map, called $M1$, is built between Q and P, and $M1$ is filled with the following entries:
$M1$:

$$\langle q1, \{p1, p3\} \rangle$$
$$\langle q2, \{p2, p4\} \rangle.$$

Next, we gather fields referenced by predicates in P, extract ranges from the predicates, and group the ranges on each of the fields. Assume I to be the list of fact table fields referenced by P. In the workload Q, we can see that there are two fields used by P: LO_DISCOUNT, LO_QUANTITY. These are added to I.

After that, we construct a set of distinct ranges on each field in I, using the predicates in P. Then, the whole range set becomes a kind of two-dimensional array, called R. The array R captures the range representation of the predicates. Each entry under a field in R represents a pair of (start and end) values forming the range. In the range representation, infinity (∞) or -infinity (-∞) can be used for unbounded ranges. "[" or "]" are used for closed ranges, and "(" or ")" for open ranges. In particular, IN predicate is represented by multiple ranges. For instance, the above $p1$ can be represented by the following three ranges $[1, 1]$, $[4, 4]$, $[5, 5]$; however, the last two consecutive ranges can be consolidated as $[4, 5]$. In the continuing example the range set R is formed by the entries below.

R:

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

Then, each predicate can be mapped to its associated ranges in R. That is, we can have another map, called $M2$, between P and R. Let $R[i,\ j]$ denote the interval value for the i-th field and j-th range in R. R[1, 3], for instance, indicates range $[7, \infty)$ under LO_DISCOUNT. In the running example, $M2$ is constructed as follows:

$M2$:

$\langle p1$, $\{R$[1, 1],R[1, 2]$\}\rangle$
$\langle p2$, $\{R$[1 ,3]$\}\rangle$
$\langle p3$, $\{R$[2, 1]$\}\rangle$
$\langle p4$, $\{R$[2, 2]$\}\rangle$.

Non-Overlapping Range Construction Since there could be multiple predicates on the same field across queries, extracted ranges may overlap each other. The overlapping ranges must be broken without any common portion, so that a pair of consecutive, non-overlapping ranges can be considered for a merge in further phases.

Algorithm 2 describes the way of breaking the overlap of ranges.

Algorithm 2: Splitting Overlapping Ranges

```
input  : $R$ having overlapping ranges
output: $R$ with consecutive, non-overlapping ranges
1  foreach field $i \in R$ do
2      $L \leftarrow$ Get the range set of $i$ from $R$ and sort by start value
3      $j = 0$ ;
4      while $j < |L|-1$ do
5          $r_j, r_{j+1} \leftarrow$ Adjacent ranges in $L$
6          if $r_j.end \geq r_{j+1}.start$ then
7              Make a split by modifying values of $r_j$ and $r_{j+1}$.
8              Insert into $L$ an intermediate range if any.
9              Re-sort ranges in $L$ and reset $j$ to 0.
10         end
11         else $j$++;
12     end
13     Update $R$ with the final $L$.
14 end
```

For each field, the tool gets its associated ranges and sorts them by start value. It then checks whether or not adjacent ranges overlap each other. If that is the case, then we make a split between the ranges. In the example, the split is applied on the overlapping ranges, $(\infty, 30]$ and $[25, 35]$, under LO_QUANTITY. Subsequently, a common, intermediate range ($[25, 30]$) is newly created and inserted into the range set belonging to the LO_QUANTITY field, in order to fill the gap. Then, R in the example is shown as follows.

R:

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

The update of R affects the existing map $M2$. In the running example, $p3$ and $p4$ influenced by the split get mapped to new range sets, $\{R[2, 1], R[2, 2]\}$ and $\{R[2, 2], R[2, 3]\}$, respectively. Of course, the range sets mapped to $p1$ and $p2$ referencing LO_DISCOUNT remain unchanged. As a result, in the example we obtain the updated $M2$ as follows:

$M2$:

$$\langle p1, \{R[1, 1], R[1, 2]\}\rangle$$
$$\langle p2, \{R[1, 3]\}\rangle$$
$$\langle p3, \{R[2, 1], R[2, 2]\}\rangle$$
$$\langle p4, \{R[2, 2], R[2, 3]\}\rangle.$$

Field Count Limit Check At present, the Teradata DBMS has a limit (64) of the fields that can be used in an MLPPI definition. If the number of fields present in R exceeds the field count limit, we determine which field(s) should be thrown away to satisfy the limit. (The star schema fact table we use has much fewer fields than the limit, and thus, this step will not be executed.) To choose victim fields, the tool computes the weighted sum of *query cost* of queries regarding each field. The sum can be obtained by adding up every query cost on an MLPPI using only the ranges under the field. (We will cover how to measure the query cost in Section 2.3.) We incrementally discard a field with the highest query cost sum until the limit is reached. Accordingly, we can update the existing $M2$. One might say that the MLPPI may not be exploited if too many ranges, exceeding a partition count limit (or 65,536 as of now) are found in one field. But we assume that such an extreme case is not expected. Even if it happens, we can utilize the merges of consecutive ranges to make the MLPPI feasible.

Query-to-Range-Set Map Construction Now, the tool can create a bi-directional *query-to-range-set* map, called M, between Q and R, using the existing maps $M1$ and $M2$. It leverages a transitive property from $M1$ to $M2$. Once M is constructed, the intermediate maps ($M1$ and $M2$) are not used in the rest of phases. M is shown as below:

M:

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

In the subsequent phases M gets used for computing costs and updated along with a merge of ranges, and R is refined for an MLPPI recommendation.

Partition Count Limit Check In this regard, the range set R can be used to define an MLPPI using each field as one level. However, if the current number of partitions by R is greater than the partition count limit, then it is not possible to make a feasible MLPPI based on ranges in R.

The actual partition count limit is ample enough to pass the running example, but to continue our discussion, assume that in our discussion the limit is *15*. From R in the running example, we can obtain a total of (number of ranges in `LO_DISCOUNT`) ·(number of ranges in `LO_QUANTITY`) = (3+1) · (3+1) = *16* partitions. Because the total partitions surpass the limit, the tool should pass through the initial phase in which fewer partitions than the limit are made. (Otherwise, the wizard will proceed to the optimized phase immediately.) Note that one more range is counted per field in the calculation. The added range is equivalent to the "`NO CASE OR UNKNOWN)`" case in Figure 1.

The tool is now ready to proceed to the next phases, with R (input range set, or MGP) and the prepared M (query-to-range-set map).

2.2. The Initial Phase

In this phase, we incrementally merge a range pair in R to reduce partitions. The merge continues until the number of ongoing partitions drop below the partition count limit. Once reaching the limit, we can have a feasible MLPPI with the remaining partitions.

By and large, overall I/O cost may be increased by the merge. Since every row is hash-sorted in each AMP of the Teradata DBMS server, typically the full scan on a partition is done to retrieve all the rows matching a given query. In the case of a merged partition, we may read the non-qualifying rows that would not be seen before the merge, thereby paying more I/O to answer the query. To minimize the merge overhead, we pick up the range pair incurring the least I/O increase in a heuristic fashion.

To choose the desirable range pair for a merge, the tool leverages the scan cost of a query on a range pair. Algorithm 3 represents how the scan cost can be computed on the range pair that influences a query. The scan cost represents the I/O cost to answer the query when the range pair is merged. It is modeled as the *number of blocks* that are to be read for the target query. To compute the scan cost of a query q, we use a corresponding scan cost query (s), and it can be constructed as follows:

s: `SELECT * FROM` F `WHERE` ***CP***,

where F is a fact table, and *CP* indicates the predicates restored from the ranges mapped to q from M.

If q turns out to be affected by the merge of a range pair, then the range set associated with q in M is temporarily updated by removing the parent ranges and adding the merged one. The altered range set is remapped to q in M and then translated to the equivalent predicates, so that *CP* for s based on the predicates can be built. Unless the merge range pair influences q, then *CP* can be simply built based on the predicates restored from the existing ranges mapped to q.

To help understand the scan cost construction, let us consider a range pair (rp) of R[2, 2] and R[2, 3] in the running example. rp produces the merged range, $[25, 35]$, under `LO_QUANTITY`. The merge by rp affects both $q1$ and $q2$ in the workload.

Thus, the range sets of $q1$ and $q2$ on `LO_QUANTITY` are altered to $\{(\infty, 25), [25, 35]\}$ and $\{[25, 35]\}$, respectively. Of course, no change is made to the existing range sets of the

Algorithm 3: Scan Cost Computation

```
   input : $q$ (query), $rp$ (range pair), $M$ (query-to-range-set map)
   output: Scan cost for answering $q$ when considering a merge of $rp$
1  $r_m \leftarrow$ Consolidate $rp$ ;
2  $L \leftarrow$ Copy the range set mapped to $q$ from $M$;
3  Delete ranges in $rp$ from and insert $r_m$ into $L$.
4  $s \leftarrow$ "SELECT * FROM FACT_TABLE WHERE ";
5  foreach range $r \in L$ do
6      Restore a predicate $p$ from $r$.
7      Add $p$ to WHERE clause of $s$.
8  end
9  Make an API call with $s$ to the server.
10 Extract the spool size (in bytes) from the result.
11 $blc \leftarrow$ Calculate the block counts from the spool size.
12 Update $q$'s scan cost to $blc$.
13 return $blc$ ;
```

queries on LO_DISCOUNT. Therefore, the corresponding scan cost queries for $q1$ and $q2$ can be built as below:

```
s1: SELECT * FROM LINEORDER l
    WHERE ((l.LO_DISCOUNT = 1)
    OR (l.LO_DISCOUNT >= 4
    AND l.LO_DISCOUNT <= 5))
    AND l.LO_QUANTITY <= 35
s2: SELECT * FROM LINEORDER l
    WHERE l.LO_DISCOUNT >= 7
    AND (l.LO_QUANTITY >= 25 AND
    l.LO_QUANTITY <= 35).
```

It is possible to combine all the (consecutive) ranges in the (temporarily) altered set of $q1$, and hence, a single predicate (`l.LO_QUANTITY <= 35`) can be built.

The wizard sends such a scan cost query to the server via an API call, and it extracts from the server's response the *spool size* (in bytes), or the size of scan query result. Then, the tool computes as scan cost the *block counts* using the spool size. If the merge of a range pair does not impact on any query in the workload, the existing (previously computed) scan cost of queries will be reused for the range pair. That is, the wizard will only recompute the scan cost of a query *iff* the query is affected by the merge of two consecutive ranges.

In this way, the weighted sum of scan cost of queries, T_s can be computed for each range pair. T_s can be defined as follows:

$$T_s = \sum_{i=1}^{n}(sc_i \cdot w_i),$$

where n is the number of queries in Q, sc_i is the scan cost of a query q_i in Q, and w_i denotes the (non-negative) weight associated with q_i.

Once all range pairs are examined, the tool chooses the range pair with the least T_s for a merge. If several range pairs end up with the same least T_s, then the tool applies

Algorithm 4: The Initial Phase

```
input  : $M$ (query-to-range-set map), $R$ (input range set)
output: $R$ with # partitions $\leq$ partition limit
1  $W \leftarrow$ Query weights
2  while (# partitions by $R >$ partition limit) do
3      foreach a range pair ($rp$) in $R$ do
4          $T_s \leftarrow 0$ ;
5          foreach $q$ in $M$ do
6              $w \leftarrow$ Get $q$'s weight from $W$
7              $L \leftarrow$ Get the range set mapped to $q$ in $M$
8              if (($rp \cap L) \neq \varnothing$) then                    // Affected
9                  $T_s$ += $w \cdot (getSC(q, rp, M))$ ; // See Algo. 3 regarding getSC()
10
11         end
12         Associate $T_s$ with $rp$.
13     end
14     Find $rp$(s) with the least $T_s$.
15     If a tie happens, select the $rp$ to make fewer partitions when consolidated.
16     Update $M$ and $R$ by the chosen $rp$.
17 end
18 return $R$;
```

the heuristic of favoring the range pair that produces fewer number of partitions when merged, to make it faster to reach the partition count limit.

In the example, suppose that the running range pair rp produces the least T_s, and thus, the tool merges rp. Thus we can update both R and M as follows.

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]}$\rangle$

Note that we see that $q2$ may have the most customized partition based on the updated M. Only the single partition formed by (R[1, 3]$\cap R$[2, 2]) is sufficient to retrieve all the qualifying rows for $q2$; thus, the other partitions can be simply eliminated. In the meantime, to answer $q1$, we need to read the four partitions formed by the top two ranges of each field. Unfortunately, because the partitions formed by ((R[1, 1]$\cup R$[1, 2])$\cap$(R[2, 2]) contain the non-qualifying rows for $q1$, R cannot provide $q1$ with as much benefit as $q2$. However, R can be a good compromise to satisfy both queries, in that the MLPPI derived by R can potentially minimize the total execution cost of Q.

Algorithm 4 represents the initial phase that has been proposed so far. The algorithm starts with a query-to-range set map (M) and an input range set (R). Until the number of

Algorithm 5: Query Cost Computation

```
   input : $q$ (query), $rp$ (range pair), $M$ (query-to-range-set map), $R$ (input range set)
   output: Query cost to answer $q$ when considering a merge of $rp$
1  Create an empty shadow table $H$ having the same definition as the fact table $F$, including
   indexes and all constraints (check and referential integrity constraints).
2  Propagate all (field and index) statistics of $F$ to $H$.
3  if $F$ has materialized views (MVs) then
4      Create the equivalent MVs on $H$.
5      Propagate the statistics of the MVs on $F$ to those of $H$.
6  end
7  $r_m \leftarrow$ Merge $rp$ ;
8  $L \leftarrow$ Copy ranges associated with $q$ from $M$;
9  Remove ranges in $rp$ from and add $r_m$ to $L$.
10 $R' \leftarrow$ Update $R$ with $L$
11 Alter $H$ using a fictitious MLPPI with $R'$.
12 Construct and send to the server an $H$-based shadow query $h$ equivalent to $q$.
13 $ept \leftarrow$ Estimated processing time of $h$ extracted from the server response ;
14 Update $q$'s query cost to $ept$.
15 return $ept$
```

the partitions in R falls below the partition limit threshold, the algorithm determines the range pair yielding the least total execution cost (T_s) and updates M and R. At the end, the algorithm produces a final recommendation based on R.

Now that the total partition counts (4×3=12) by R falls below the assumed limit (15), the wizard proceeds to the optimized phase with R without repeating the initial phase.

2.3. The Optimized Phase

In this regard, using R we have an initial MLPPI recommendation for the LINEORDER fact table. The recommendation can be used sufficiently, but having fewer partitions, induced by further merges, can enhance the overall workload performance for the following reasons. First, multiple file contexts by many partitions can incur huge overhead that impacts on the query optimizer. There also exists an operational threshold that the optimizer can handle the maximum number of the partitions at a time. Therefore, further reducing partitions greatly helps the optimizer to manage these partitions.

A similar algorithm, as suggested in the initial phase, can be applied to this optimized phase. Instead of the scan cost, the *query cost* can be used thanks to a feasible MLPPI. The query cost is modeled as the estimated processing time of a query on a "faked" fact table applying the MLPPI definition reflecting the merge of a range pair. This cost estimation technique is similar to the "what-if" approach [4].

Algorithm 5 illustrates the steps for computing the query cost of a query q, given a range pair (rp) affecting q when merged. First, we create an empty shadow table H with the same definition as F, including all indexes and constraints such as check and referential integrity ones. Next, we propagate all statistics of F to H. This covers field and index statistics. If F has materialized views (MVs), then we create the equivalent MVs on H and subsequently, propagate the statistics of the MVs on F to the new MVs

```
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)
);
```

Fig. 3. An MLPPI Recommendation for Workload Q

on H. After that, we alter H by an MLPPI with R' applying the merge of rp. The wizard then builds a *shadow* query h equivalent to the input query q but replacing F in q with H, and makes an API call to the `EXPLAIN` tool with h. Eventually, the tool extracts from the result and returns the estimated elapsed time of h as the query cost of q. As done in the initial phase, the query cost of a query is recomputed only if the merge affects the query.

For each range pair the wizard computes the weighted sum (T_q) of query cost of queries. T_q can be similarly defined as T_s, described in Section 2.2. Let the current least T_q be T and the existing least T_q be T_p. If $T <= T_p$, then for the next iteration T_p is updated to T. In turn, the tool merges the range pair that produces T and updates M and R along with the merge. The wizard repeats the process until 1) no ranges in R remain, or 2) pre-defined iterations are reached. The optimized phase is finished if $T \geq T_p$.

In the running example, suppose that in the first round, ranges R[1, 1] and R[1, 2] are chosen for a merge. Then, the wizard produces a merged range $[1, 5]$ under `LO_DISCOUNT` and proceeds to the next round. If a range pair selected for the subsequent merge fails to improve T_p, then, the tool exits the optimized phase and produces the final MLPPI recommendation for the given workload Q as shown in Figure 3.

Algorithm 6 exhibits the proposed optimized phase, which also applies the same heuristic to favor a range pair that produces fewer partitions to break a tie. The order of the different partitioning levels may have some impact on the execution time. A range condition on lower levels like the query in Section 1 "`SELECT * FROM LINEORDER WHERE LO_QUANTITY < 25`" requires scanning non-consecutive partitions. This may incur some overhead at run time. To reduce this effect, we sort the partition levels based on number of partitions in descending order before making a final recommendation. The solution in Figure 1 follows this heuristic making the two-partition case in the first level followed by the four-partition case in the second level. The order in Figure 3 can go either way since both levels have the same number of partitions.

3. Analysis

In this section we analyze the time complexity of the proposed preprocessing, initial, and optimized phases. For the time complexity we compute the logical running time of the wizard, using the *number of API calls* made to the server during the phases.

Algorithm 6: The Optimized Phase

```
input  : $M$ (query-to-range-set map), $R$ (input range set)
output: MLPPI such that the total query cost is minimal
W ← Query weights
$T_p$ ← Total query cost sum on an initial MLPPI by $R$
while (Pre-defined iterations or $R \neq \emptyset$) do
    foreach range pair $rp$ in $R$ do
        $T_q \leftarrow 0$ ;
        foreach $q$ in $M$ do
            $w$ ← Get $q$'s weight from $W$
            $L$ ← Get the range set mapped to $q$ in $M$
            if (($rp \cap L$) $\neq \emptyset$) then                // Affected
                $T_q$ += $w$·(getQC($q$,$rp$,$M$,$R$)) ;   // See Algo. 5 regarding
                getQC()
            else $T_q$ += $w$·($q$'s existing query cost)
        end
        Map $T_q$ to $rp$.
    end
    $T$ ← The least $T_q$ that has been so far seen.
    if $T > T_p$ then break ;
    else
        $T_p \leftarrow T$ ;                       // Query Cost Sum Update
        Find $rp$(s) with $T$.
        If a tie happens, select the $rp$ to make fewer partitions when consolidated.
        Update $M$ and $R$ by the chosen $rp$.
    end
end
return an MLPPI by $R$ ;
```

3.1. The Preprocessing Phase

Lemma 1. *The running time complexity for building a query-to-range map is O($N \cdot C \cdot V$), where N is the number of queries in a workload, C is the total number of fields in the fact table, and V is the max number of values in a field.*

Proof. Let us consider a worst case such that given a workload of N queries, each query references all the fields of the fact table, and each field is associated with at most V values by the predicates of the queries. Each query may have a single-value range. A range consisting of only a single value per field can be mapped to each query. Thus, the running time complexity for the map construction is O($N \cdot C \cdot V$). □

Comment: In practice, our experiments showed that the bound O($N \cdot C \cdot V$) is overly pessimistic.

3.2. The Initial and Optimized Phases

Lemma 2. *The running time complexity for the initial or optimized phases is O($M^2 \cdot N$), where M is the number of range pairs, and N is the number of queries in a given workload.*

Proof. Suppose that the merge of every range pair influences all the queries. Every iteration, either the scan or query costs need to be recomputed for each query. In the first round the re-computation cost is paid as many times as $M \cdot N$, and a chosen range pair is merged. In the subsequent round M-1 range pairs remains, and the following cost amounts to $(M\text{-}1) \cdot N$. In the worst case we may end up consolidating all range pairs, thus having no partition on a target table. The total calls to the server can be increased up to

$$M \cdot N + (M\text{-}1) \cdot N + \cdots + N = (\textstyle\sum_{i=1}^{M} i) \cdot N = \frac{M(M-1)}{2} \cdot N.$$

Hence, the running time complexity is $O(M^2 \cdot N)$. □

Our experiments showed that $O(M^2 \cdot N)$, the running time complexity, is overly pessimistic. In practice, the number of calls made was much fewer than the theoretic bound, since it was very rare for a range pair to be associated with all queries in a workload. In Section 4.2 we will show how many API calls take place on the workloads used for evaluation.

4. Experiment

In this section, we describe our environment settings, report the statistics measured in our experiments, and demonstrate the performance of the MLPPI wizard (WIZARD), compared with that of no partitioning and the partitioning by a human expert (EXP).

4.1. Environment Settings

The MLPPI wizard was implemented as a prototype on top of the Teradata DBMS server. It was written in Java. The performance of the wizard was evaluated on the Teradata DBMS server machine running Unix.

When it comes to workload generation, we took advantage of our simple star schema query generator. The generator assumes that 1) available operators, 2) fields, and 3) the minimum and maximum values of the fields are already known. For each query, the generator first randomly selects the number of single-table predicates to create. Then, it arbitrarily chooses a field and a specific operator for each predicate. If `IN` operator on a chosen field is picked up, the query generator determines the number of values to add to the `IN` list and then chooses random values within the min-max value range of the field. In this way the generator builds a query involving the generated predicates.

A generated query is based on a template that joins `LINEORDER` and `DDATE` with constraints defined by the predicates. This template is common in customer cases like reports and form templates. Using the query generator we generated two workloads consisting of 10 queries (10Q) and 20 queries (20Q).

For the query workloads we populated the fact table with a scale of 1TByte (1TB) and 3TBytes (3TB).

4.2. Statistics Report

Table 1 summarizes the WIZARD's execution statistics, which were obtained while the recommendations for the workloads were produced. The statistics includes input partitions, range pairs, number of API calls made, and total iterations observed in each phase.

Table 1. The statistics obtained on the used workloads

	Init. Phase		Opt. Phase	
	10Q	20Q	10Q	20Q
# Input Partitions	1,008,000	110,739,200	51,840	59,520
# Input Range Pairs	441	4,180	24	48
# Total Iterations	14	55	1	1
# Total API Calls (Worst Calls by Lemma 2)	1,305 (1,944,810)	31,915 (349,448,000)	78 (5,760)	388 (46,080)
# Average API calls per Range Pair	2.96	7.63	3.25	8.08
# Average number of range pairs compared per iteration	31.5	76.0	24	48
# Maximum number of range pairs compared per an iteration	38	103	24	48
# Minimum number of range pairs compared per an iteration	25	49	N/A	N/A
# Average API calls made per iteration	93.21	580	78	388
# Maximum number of API calls made per an iteration	123	791	78	388
# Minimum number of API calls made per an iteration	73	381	N/A	N/A

About 1 million partitions were initially derived for 10Q whereas roughly 0.11 billion partitions for 20Q. Although the workload size doubled, the total number of partitions was increased exponentially. Much more predicates with different bindings produced two orders of magnitude more ranges in 20Q than those of 10Q. While the huge input partitions were made, the number of collected range pairs per field was relatively small.

The total number of iterations in the initial phases tended to be proportional to the input partitions generated from the workloads. Note that there happened only one iteration in the optimized phase. The reason was that since the partitions produced by the initial phase were customized enough, no more iterations were necessary for further optimization.

As mentioned in Section 3, the total calls made to the server per workload were by far fewer than the theoretical bound shown in parentheses. The worst call counts were calculated regarding the number of queries and range pairs observed in each phase. The average number of calls per range pair was limited within 10.

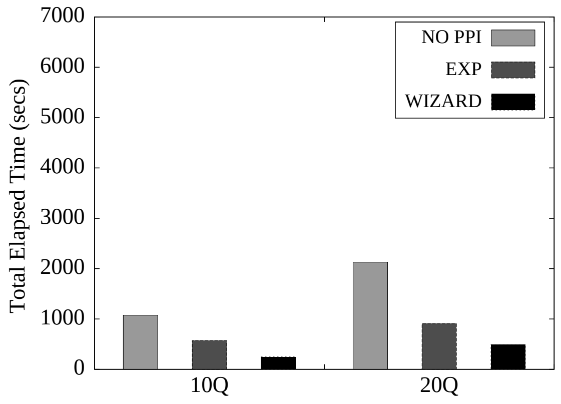
4.3. Performance Evaluation

The main focus of our demonstration is to see the performance (quality) of MLPPI recommendations made by the MLPPI wizard (WIZARD), compared with the performance of no partitioning (NO PPI) and EXP (partitioning by a human expert).

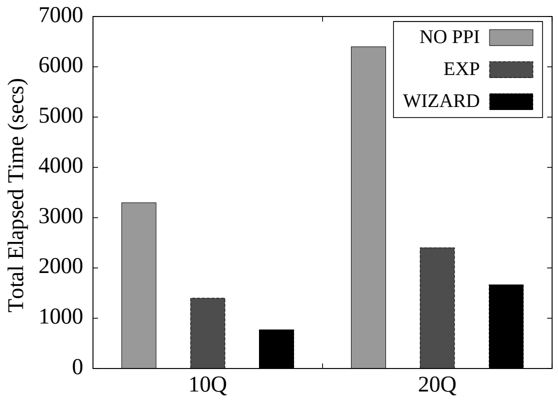
Figures 4 and 5 exhibit our evaluation results. Figure 4 shows the times (in seconds) taken to run the 10Q and 20Q workloads over the 1TB and 3TB fact tables. Based on the run results, Figure 5 shows how much improvement was gained by the EXP and WIZARD solutions, compared with the NO PPI solution.

Overall, the WIZARD recommendations were very effective at partitioning the fact tables by leveraging the predicates in the workloads. On average, the total execution time of the workloads on the WIZARD solutions was about 3.5 and 2 times faster than those of NO PPI and EXP, respectively. Also, the quality of the WIZARD solutions outperformed that of the EXP solutions; the WIZARD solutions yielded an average of about 77% improvement on the NO PPI solutions.

Furthermore, the WIZARD recommendations scaled very well with growing workload size and table size, as illustrated in Figure 5. Although we doubled the workload scale from 10Q to 20Q, the performance improvement of the WIZARD solution stayed the same in Figure 5(a) and got almost negligibly decreased in Figure 5(b). This result showed that our WIZARD solution was scalable over increasing workload size.



(a) 1TB



(b) 3TB

Fig. 4. Total Elapsed Time

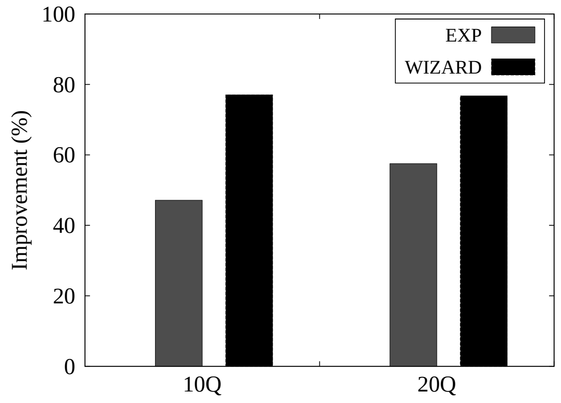
We also increased the table scale from 1TB to 3TB. For 10Q the performance improvement (about 77%) was almost the same over the increasing scales, and for 20Q it was decreased from 76% to 74%, which was almost negligible. The WIZARD partitioning recommendation scaled well with increasing table size.

In sum, our experiment results attest the effectiveness of the proposed algorithms for WIZARD and show the superiority of the recommendations of WIZARD.

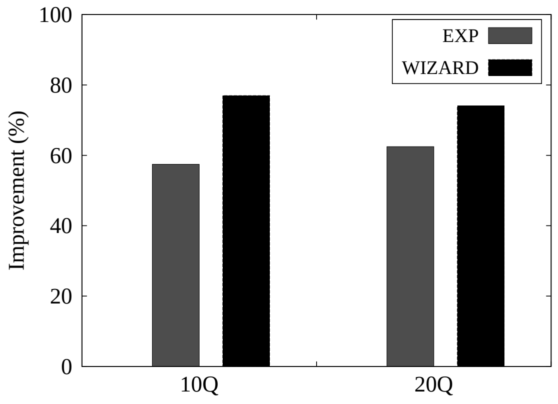
5. Related Work

Physical database design [6, 10, 18, 25] has been discussed in academic research and industrial sectors in the past years. The major DBMS vendors (e.g., IBM, MS, and Oracle) have driven much of the work. Their specific interests have been in automating the physical design for table partitioning [3, 11, 13, 15–17], indexes/materialized views [2, 1, 5, 8, 12, 22, 24], and integration [23].

Some of the tools in IBM DB2 [11], Oracle [15], and MS SQL Server [3] appear to be similar to our MLPPI wizard. However, in light of problem scope and approach, the



(a) 1TB



(b) 3TB

Fig. 5. Recommendation Quality Comparison

wizard is fundamentally different from the existing tools excluding Oracle Partitioning Advisor [15] providing no published technical details.

DB2 MDC Advisor [11] actually tackles a different problem of automatically recommending the most well-suited MDC keys for a given workload. Agrawal's work [3] discusses another problem of merging single-level range partitionings on objects such as tables and indexes. This article addresses the multi-level partitioning problem. Hence, the existing solutions cannot be directly applied to our MLPPI wizard.

Regarding the approach, DB2 MDC Advisor [11] uses the search space driven by *fields*. In contrast, our search space is driven by query-predicates, which is superior because predicates are more customized and specific to a workload than fields. Agrawal's horizontal partitioning scheme [3] also uses the search space driven by simple range predicates, but his technique has a shortcoming as follows. His work produces a solution for each individual query and attempt to merge the solutions. But this approach cannot reach an optimized solution in a global perspective. Our wizard generates the whole search space upfront and in turn merges partitions, leading to a globally-optimized solution. Moreover, only a single column is considered in his work [3], whereas our tool deals with multiple fields.

Implementations of previous tools [3, 11, 13] 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 that 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. The APIs are sufficient to avoid the costly optimizer code change.

Also, Nehme's work [13], deeply-integrated with optimizer, reveals a concern about the quality of the recommendations made by some tools, shallowly integrated with optimizer. But we observed in our experiments that the quality of the wizard's solutions was much superior to that of the base solutions. In addition, some might argue that in our loosely-coupled approach the cost to invoke the optimizer might be significant. But the measured call counts were much fewer than the theoretical bound, since most calls were made only when queries were affected by a merge.

Lastly, there has been some previous work [13, 17, 21] regarding table partitioning in multi-node systems, but our problem is discussed in the context of a single node system, as in the existing work [3]. Database cracking [7] assumes a single node environment, but it does not address our multi-level partitioning problem.

6. Summary

Given workloads, it is difficult for DBAs to select appropriate fields in partitioning the fact table due to large search space. DBAs cannot easily determine how granular partitions should be made for the workloads.

To address this concern, we presented the MLPPI wizard to recommend a fact-table partitioning for a given star schema workload. Since query-predicates are exploited to capture necessary ranges for fields and define partitions, the MLPPI recommendation can be very optimized, customized to the workload.

We proposed the wizard's algorithms consisting of the three phases. The wizard incrementally reduced its search space by merging the range pair with the least scan or query cost, and eventually reached an MLPPI recommendation. In addition, we analyzed the running complexity for the initial and optimized phases. Despite the theoretically high bound, in practice, the wizard made much fewer calls in the phases.

We measured the performance of the recommendation by the wizard using our workloads. We demonstrated that the produced MLPPI solutions by the wizard could reduce the total elapsed time by more than a factor of two, compared with those of no partitioning approach and partitioning done by a human expert.

Acknowledgments

This work was carried out while the first author visited Teradata. The first author is now at Korea Institute of Science and Technology Information (yksuh@kisti.re.kr).

References

1. Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., Syamala, M.: Database Tuning Advisor for Microsoft SQL Server 2005: Demo. In: SIGMOD. pp. 930–932 (2005)

2. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated Selection of Materialized Views and Indexes in SQL Databases. In: VLDB. pp. 496–505 (2000)
3. Agrawal, S., Narasayya, V., Yang, B.: Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In: SIGMOD. pp. 359–370 (2004)
4. Chaudhuri, S., Narasayya, V.: AutoAdmin 'What-If' Index Analysis Utility. SIGMOD Record 27(2), 367–378 (1998)
5. Dash, D., Polyzotis, N., Ailamaki, A.: Cophy: A scalable, portable, and interactive index advisor for large workloads. PVLDB 4(6), 362–372 (2011)
6. Finkelstein, S., Schkolnick, M., Tiberio, P.: Physical Database Design for Relational Databases. ACM Trans. on Databas. Syst. 13(1), 91–128 (1988)
7. Idreos, S., Kersten, M.L., Manegold, S.: Database Cracking. In: CIDR. pp. 68–78 (2007)
8. Kimura, H., Narasayya, V., Syamala, M.: Compression Aware Physical Database Design. PVLDB 4(10), 657–668 (2011)
9. Klindt, J.: Single-level and Multilevel Partitioned Primary Indexes, [Online]. Available: http://www.teradata.com/white-papers/Single-level-and-Multilevel-Partitioned-Primary-Indexes-eb1889/ (current July 2015)
10. Labio, W., Quass, D., Adelberg, B.: Physical Database Design for Data Warehouses. In: ICDE. pp. 277–288 (1997)
11. Lightstone, S.S., Bhattacharjee, B.: Automated Design of Multi-dimensional Clustering Tables for Relational Databases. In: VLDB. pp. 1170–1181 (2004)
12. Microsoft SQL Server 2000: Index Tuning Wizard SQL Server 2000, [Online]. Available: http://technet.microsoft.com/en-us/library/cc966541.aspx (current March 2012)
13. Nehme, R., Bruno, N.: Automated Partitioning Design in Parallel Database Systems. In: SIGMOD. pp. 1137–1148 (2011)
14. O'Neil, P., O'Neil, B., Chen, X.: The Star Schema Benchmark (SSB), [Online]. Available: http://www.cs.umb.edu/ poneil/StarSchemaB.PDF (current February 2017)
15. Oracle: Partitioning Advisor, [Online]. Available: http://www.oracle.com/technetwork/database/options/partitioning/twp-partitioning-11gr2-2009-09-130569.pdf (current February 2017)
16. Oracle: SQL Access Advisor, [Online]. Available: http://docs.oracle.com/cd/B19306_01/server.102/b14211/advisor.htm (current March 2012)
17. Rao, J., Zhang, C., Megiddo, N., Lohman, G.: Automating Physical Database Design in A Parallel Database. In: SIGMOD. pp. 558–569 (2002)
18. Rozen, S., Shasha, D.: A Framework for Automating Physical Database Design. In: VLDB. pp. 401–411 (1991)
19. Sinclair, P.: Using PPIs to Improve Performance (2008), [Online]. Available: http://www.teradata.com/tdmo/v08n03/pdf/AR5731.pdf (current August 2015)
20. Suh, Y.K., Ghazal, A., Crolotte, A., Kostamaa, P.: A New Tool for Multi-level Partitioning in Teradata. In: CIKM. pp. 2214–2218 (2012)
21. Tatarowicz, A. et. al.: Lookup Tables: Fine-grained Partitioning for Distributed Databases. In: ICDE (2012)
22. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G., Skelley, A.: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In: ICDE. pp. 101–110 (2000)
23. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., Fadden, S.: DB2 Design Advisor: Integrated Automatic Physical Database. In: VLDB. pp. 1087–1097 (2004)
24. Zilio, D.C., Zuzarte, C., Lohman, G.M., Pirahesh, H., Gryz, J., Alton, E., Liang, D., Valentin, G.: Recommending Materialized Views and Indexes with the IBM DB2 Design Advisor. In: Proc. of the Int'l Conf. on Autonomic Computing. pp. 180–188 (2004)
25. Zilio, D.C.: Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems. Ph.D. thesis, Department of Computer Science, University of Toronto (1998)