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

Young-Kyoon Suh[1], Alain Crolotte[2] and Pekka Kostamaa[3]
[1] School of Computer Science and Engineering,
Kyungpook National University
Daegu, 41566 – Republic of Korea
[e-mail: yksuh@knu.ac.kr]
[2] Teradata Corporation,
El Segundo, CA 90245, USA
[e-mail: {alain.crolotte}@teradata.com]
[3] OpenX
West Los Angeles, CA 90232, USA
[e-mail: {pekkak@yahoo.com}]
*Corresponding author: Young-Kyoon Suh

*Received July 19, 2017; revised October 14, 2017; accepted November 8, 2017;
published April 30, 2018*

Abstract

An important technique used by database administrators (DBAs) is to improve performance in decision-support workloads associated with a Star schema is multi-level partitioning. Queries will then benefit from performance improvements via partition elimination, due to constraints on queries expressed on the dimension tables. As the task of multi-level partitioning can be overwhelming for a DBA we are proposing a wizard that facilitates the task by calculating a partitioning scheme for a particular workload. The system resides completely on a client and interacts with the costing estimation subsystem of the query optimizer via an API over the network, thereby eliminating any need to make changes to the optimizer. In addition, since only cost estimates are needed the wizard overhead is very low. By using a greedy algorithm for search space enumeration over the query predicates in the workload the wizard is efficient with worst-case polynomial complexity. The technology proposed can be applied to any clustering or partitioning scheme in any database management system that provides an interface to the query optimizer. Applied to the Teradata database the technology provides recommendations that outperform a human expert's solution as measured by the total execution time of the workload. We also demonstrate the scalability of our approach when the fact table (and workload) size increases.

Keywords: Data warehousing, information systems, star schema, fact table, OLAP, analytical workloads, multi-level partitioning

A preliminary version of this paper appeared as a short paper in ACM CIKM 2012, October 29-November 12, Hawaii, USA. This extended version includes a substantial amount of new materials and contributions. This work started initially when the first author interned at Teradata. The work was supported by the National Research Foundation of Korea(NRF) grant funded by the Ministry of Science and ICT (No. NRF-2018R1C1B6006409). We thank to Dr. Ahmad Ghazal for his initial contribution and to anonymous reviewers for their insightful comments.

http://doi.org/10.3837/tiis.2018.04.016 **ISSN : 1976-7277**

1. Introduction

Over the past decades much attention has been paid to improve the performance on analytical workloads like OLAP in a data warehousing environment. A relational database management system (DBMS), which underlies information systems, has been exploited to process such analytical queries faster and assist customers to draw a timely business-decision in rapidly-changing enterprise data warehousing. Teradata DBMS has been positioned as leading one in this data warehousing marketplace getting more competitive than ever.

Background: To optimize the performance of analytical processing in the Teradata DBMS, tables and materialized views 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.

Partitioned Primary Index (PPI) [1] is an optional horizontal partitioning scheme applied locally on the data belonging to each AMP. In some other DBMSes, this type of partitioning is termed *clustering*, but hereafter we call the term partitioning only 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 group data with the same values together in contiguous data blocks. This grouping enables "partition elimination" in scans and joins for performance enhancement. PPI can be specified as single or multiple levels. This type of partitioning scheme is known as Multi-Level PPI (MLPPI) [2].

By allowing non-qualified partitions to be eliminated,MLPPI can considerably reduce 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.

Table 1 exemplifies an MLPPI table with a defined partitioning option. The example is a subset of the `LINEORDER` table from the Star Schema Benchmark (SSB) [3]. (Note that the full table with modified data types was actually used in our experiments.) In the definition of `LINEORDER` in **Table 1**, the primary index is on `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`. Specifically, `LO_DISCOUNT` has two ranges: (i) one for values >= 7 and (ii) another for all the other values (or `NO CASE OR UNKNOWN`), and `LO_QUANTITY` has four ranges. Hence, the relation will have a total of 2×4 = 8 partitions as shown in **Table 2**.

Whatever partition is chosen, namely, the mapping of rows to partitions must be an injection. the selection must be semantically correct. 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 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 following query: '`SELECT * FROM LINEORDER WHERE LO_QUANTITY < 25.`'

Table 1. An Example of a LINEORDER Table [3] with Multi-level Partitioned Primary Index

Fact table	Partitioning Candidate

LineOrder	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 AND LO_QUANTITY <= 30,
	        LO_QUANTITY > 30 AND LO_QUANTITY <= 35,
	        NO CASE OR UNKNOWN));

Table 2. All Possible Partitions by LO_DISCOUNT and LO_QUANTITY

Partition Number	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

Challenge*:* Consider the query set *Q* with two queries *q1* and *q2* on the SSB as illustrated in **Table 3**. A DBA may focus on the LINEORDER fact table and the predicates involving the table's columns only. The DBA can then figure out the five constraints, identified by the query number and the sequence number of predicates in each query as shown in **Table 4**.

At this point DBA needs to take into account options based only on two fields and the five constraints. There are many other possibilities from a "fine-grained" partition set to a "coarser-grained" definition.

Given 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 partitioning expression (named Candidate 1) for LO_DISCOUNT as illustrated in the first row of **Table 5**. This 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 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 another candidate (named Candidate 2) as seen in the second row of **Table 5**. Candidate 2 decreases the number of partitions, compared to Candidate 1 but still focuses sharply on the constraints with still relatively small partitions.

Similar considerations apply to the partitioning associated with LO_QUANTITY. The resulting partitioning definition for the table includes both LO_DISCOUNT and LO_QUANTITY, and thus the number of possible combinations is the product of the potential combinations for each field.

Table 3. An Example of a Given Star Schema Query Workload (*Q*)

Query ID	*Query String*

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`

Table 4. The Extracted Query Predicates for Each of the Two Queries in *Q*

Query Number.Sequence Number	*Query Predicate*
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`

Table 5. Possible Partitioning Candidates Based on `LO_DISCOUNT`

Partitioning Candidates	

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

Candidate 2	`CASE_N(LO_DISCOUNT >= 1 AND LO_DISCOUNT <= 5,`
	`LO_DISCOUNT >= 7, NO CASE OR UNKNOWN)`

Furthermore, given two queries one partitioning scheme may be favorable for one query but not for the other, or vice versa. Consequently, the DBA is 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” for the DBA.

Solution: To assist the DBA to draw a better partitioning solution, in this article we propose a novel physical database design tool, termed *MLPPI wizard*. The proposed tool recommends an effective partitioning solution customized for a given workload. The proposed tool uses a greedy algorithm for search space enumeration. The initial search space is driven by query predicates extracted from the workload. The tool utilizes the optimizer’s cost model to prune the search space and then reach the final solution. In particular, the tool is based on a “general framework” allowing general expressions, ranges and case expressions for partition definitions. Therefore, the predicate-driven technique used by the tool can be applied to *any* clustering or partitioning scheme based on simple fields and expressions or complex SQL predicates in an “arbitrary” DBMS.

Contribution*:* The article provides the following contributions.

- We propose a novel physical database design tool, called MLPPI wizard, to produce a multi-level partitioning solution for minimizing the execution cost of a given query workload on a star schema of a data warehouse.
- Our proposed tool uses a greedy technique by which we incrementally narrow down the search space, initially driven by the predicates of the given workload.
- The tool's predicate-driven method for search space enumeration can be applied to any clustering or partitioning scheme in a relational DBMS, allowing general expressions for partitioning definitions; note also that any complex query workload, giving many different partitioning options, can benefit from the proposed tool to find almost an optimal partitioning in an automated fashion.
- Our tool is totally outside the database server, thereby incurring no overhead of instrumenting query optimizer's code, which is required by some existing tools [4–6].
- To the best of our knowledge, our tool is the very *first* physical database design tool to address the multi-level partitioning problem on analytical workloads.
- We empirically show that a partitioning recommendation of our tool outperforms those of a human expert and no partitioning over increasing workload size and growing fact table.

This manuscript is a substantial extension of an earlier work [7]. The rest of the manuscript is organized as follows. In the subsequent section, we review some related works and elaborate on how the proposed tool is fundamentally different from existing tools proposed by other DBMSs. We in turn propose a multi-level partitioning algorithm—consisting of three phases—of our MLPPI wizard. Subsequently, we conduct a detailed analysis of the asymptotic complexity of the proposed algorithm. We then provide the performance evaluation results. Finally, we conclude this article by summarizing our discussion.

2. Related Work

Physical database design [4–22] has been discussed in academic research and industrial communities in the past decades. The major DBMS vendors have led much of the work. Their specific interests have been around automating the physical design for table partitioning [4–6, 12–14], indexes/materialized views [8, 15–21], and integration [22].

Some of the tools in IBM DB2 [5], Oracle [12], and MS SQL Server [4] appear to be similar to our MLPPI wizard. With respect to problem scope and approach, however, our wizard is fundamentally different from the existing tools. First, DB2 MDC Advisor [5] actually tackles a different problem of automatically recommending the most well suited MDC keys for a given workload. Agrawal's work [4] discusses another problem of merging single-level range partitionings on objects such as tables and indexes. On the other hand, we address the multi-level partitioning problem. Hence, the existing solutions cannot be directly applied to our MLPPI wizard.

Regarding the approach, DB2 MDC Advisor [5] uses the search space driven by fields. In contrast, our search space is driven by query-predicates, of which the use is superior to that of the fields, as utilizing predicates can be more customized and specific to a given workload.

Agrawal's horizontal partitioning scheme [4] also uses the search space driven by simple range predicates, but his technique has a shortcoming. Specifically, his work produces a solution for each individual query and attempt to merge the solutions. However, this approach cannot reach an optimized solution in a global perspective. We generate the whole search space upfront and in turn merge partitions, leading to a globally optimized solution. While his work considers a single column only, our wizard deals with multiple fields.

Oracle Partitioning Advisor [12] provides no published technical details. Thus, we do not know its similarities and differences, compared with our work.

As addressed in Section 1, implementations of previous tools [4–6] 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. However, our algorithm is based existing APIs supported by an optimizer, which requires no code change in the optimizer.

In addition, Nehme's work [6], deeply integrated with optimizer, reveals a concern about the quality of the recommendations made by some tools, shallowly integrated with optimizer. However, 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.

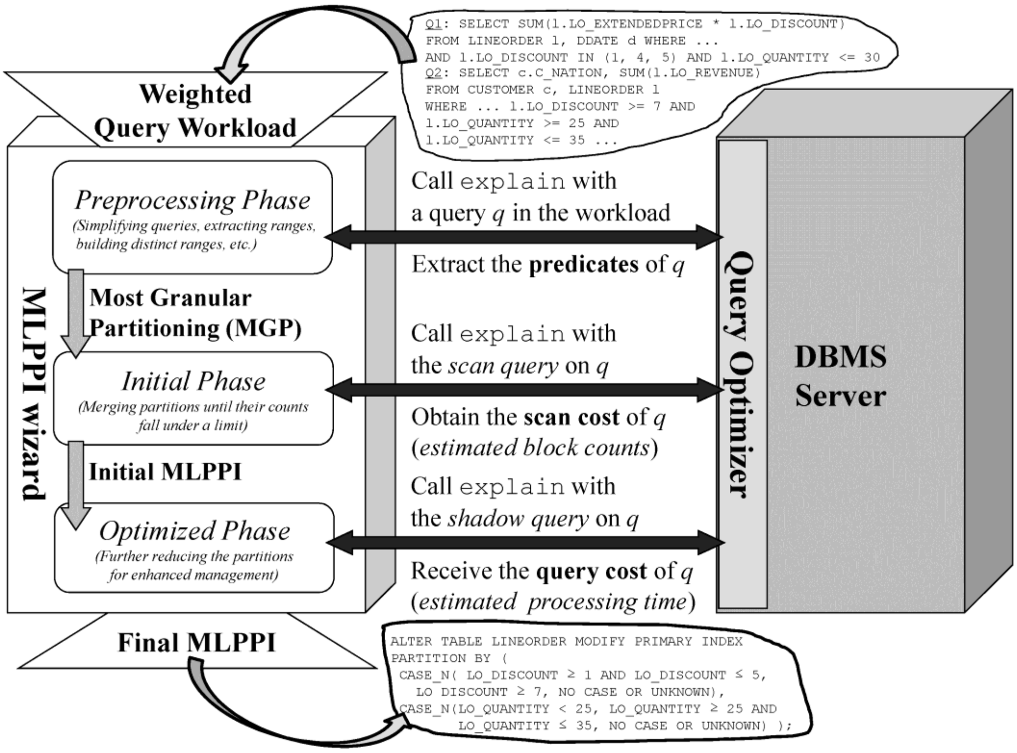
Some previous work [6, 14, 23] tackles table partitioning in multi-node systems, but our problem is discussed in the context of a single node system, as in the existing work [4]. Database cracking [24] assumes a single node environment, but it does not address our multi-level partitioning problem.

There exists also some other recent work proposing a multi-level algorithm or a recommendation system. Wang et al. [25] aim at improving online service under hybrid deployment, using an effective multi-level scheduling algorithm. Yuan's work [26] proposes a top-k recommendation system for OLAP sessions for better user assistance. Even though these works and our work seem to have something in common in terms of multi-level and recommendation, the existing works were discussed in a different domain from ours, mainly focusing on "partitioning" a fact table for better performance of a star schema workload.

Finally yet importantly, there are some recent works [27, 28] addressing the problems of exploiting analytical workloads from the real scenarios. Those works have something in common with our work, in terms of utilizing analytical workloads. Each of these two camps, however, focuses on fundamentally different problems. The authors' works seem interested in exploiting analytical workloads from the real studies in order to propose algorithms for efficient job and task scheduling in grid computing, but we exploit a given star schema analytical workload for producing with no human intervention a possible partitioning solution on a huge fact table for the DBA.

3. The MLPPI wizard

Fig. 1 shows the overall architecture of the wizard. As shown in **Fig. 1**, our wizard recommends the final MLPPI customized for a given workload consisting of a set of queries and corresponding weights while passing through a series of phases—*Preprocessing*, *Initial*, and *Optimized* Phases. Throughout this section, we elaborate in detail on each of the three phases, using the running query set *Q* comprised of *q1* and *q2* provided in Section 1, for better exposition. As seen in **Fig. 1**, the wizard resides completely on the client side, as opposed to several previous tools [4–6] that require optimizer code instrumentation. This loosely coupled


Weighted Query Workload
MLPPI wizard
Preprocessing Phase
(Simplifying queries, extracting ranges, building distinct ranges, etc.)
Most Granular Partitioning (MGP)
Initial Phase
(Merging partitions until their counts fall under a limit)
Initial MLPPI
Optimized Phase
(Further reducing the partitions for enhanced management)
Final MLPPI
Call explain with a query q in the workload
Extract the predicates of q
Call explain with the scan query on q
Obtain the scan cost of q (estimated block counts)
Call explain with the shadow query on q
Receive the query cost of q (estimated processing time)
Query Optimizer
DBMS Server

```
Q1: SELECT SUM(l.LO_EXTENDEDPRICE * l.LO_DISCOUNT)
FROM LINEORDER l, DDATE d WHERE ...
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_DISCOUNT >= 7 AND
l.LO_QUANTITY >= 25 AND
l.LO_QUANTITY <= 35 ...
```

```
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. 1. The MLPPI Wizard Architecture (with an Input Workload and the Final Solution)

approach makes the wizard extensible and portable to different releases of the DBMS server. Our tool simply invokes the server's existing APIs used to simplify the queries, capture fact table predicates, and estimate processing costs.

3.1 The Preprocessing Phase

The first phase extracts query predicates and checks any constraint in partitioning. This phase consists of six steps, as shown in **Fig. 2**.

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*)

Fig. 2. The Preprocessing Phase

Query Simplification: The first step is to simplify the predicates of queries in a given workload. In other words, we remove redundant conditions among the queries. 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 simplification can be performed via an API call to the database server. This simplification is not required for the running example.

Table 6. *Collected Predicates from Q*

Predicate ID	*Predicate*
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

Range Extraction: In the subsequent step, we gather the simplified predicates and then extract ranges from the collected predicates. This task can also be finished through an API call to the server. The query predicate is of the form,

⟨*variable*⟩ ⟨*op*⟩ ⟨*constant*⟩,

where ⟨*variable*⟩ is a field from LINEORDER, ⟨*op*⟩ is in {= , <, <=, >=, >, IN}, and ⟨constant⟩ represents a constant value(s). All ⟨*op*⟩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* as shown in **Table 6**.

Now we build a set of distinct ranges for each field referenced by the predicates. Specifically, we gather fields referenced by predicates in *P*, extract ranges from the predicates, and group the ranges of each of the referenced fields. Suppose *L* to be a list of fact table fields referenced by *P*. In the workload *Q*, we add in *L* the following two fields: LO_DISCOUNT and LO_QUANTITY used by *P*. The whole range of the gathered predicates can be represented by a kind of two-dimensional array, called *R*.

Each entry under a field in *R* represents a pair of (start and end) values forming the range. When it comes to range representation for an entry, '[' and ']' are used for closed ranges while '(' and ')' for open ranges. Also, infinity (∞) and -infinity (-∞) can be used for unbounded ranges. In particular, IN predicate is represented by multiple ranges. The above *p1*, for example, 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 per-field distinct range set (*R*) is comprised of the entries below.

R:

LO_DISCOUNT = {[1, 1], [4, 5], [7, ∞)}
LO_QUANTITY = {(∞, 30], [25, 35]}.

In turn, we build another map, called *M*2, between *P* and *R*. That is, we associate each predicate in *P* with a corresponding range in *R*. Let *R*[*i*, *j*] denote the interval value for the *i*-th field and *j*-th range in *R*. For instance, *R*[1, 3] indicates range [7, ∞) under LO_DISCOUNT. In the running example, *M2* is constructed as follows:

M2:

⟨*p1*, {*R*[1, 1], *R*[1, 2]}⟩
⟨*p2*, {*R*[1 ,3]}⟩
⟨*p3*, {*R*[2, 1]}⟩
⟨*p4*, {*R*[2, 2]}⟩.

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

Fig. 3 describes the way of breaking the overlap of ranges. For each field, the algorithm gets associated ranges of each field and sorts them by start value. We then check whether adjacent ranges overlap each other or not. If that is the case, we make a split between the ranges. In the example, the split is applied on the overlapping ranges, (∞, 30] and [25, 35] in `LO_QUANTITY`. Subsequently, we then create a common (or intermediate) range ([25, 30]) and insert into the range set belonging to the `LO_QUANTITY` field, in order to fill the gap. *R* includes the following ranges:

R:

`LO_DISCOUNT` = {[1, 1], [4, 5], [7, ∞)}
`LO_QUANTITY` = {(∞, 25), [25, 30], [31, 35]}.

The update of *R* affects the existing map *M2*. In the running example, *p3* and *p4*, influenced by the split, are mapped to new range sets, {*R*[2, 1], *R*[2, 2]} and {*R*[2, 2], *R*[2, 3]}, respectively. Of course, the range sets corresponding to *p1* and *p2* referencing `LO_DISCOUNT` remains unchanged. As a result, in the running example we obtain the updated *M2* in the following.

M2:

⟨*p1*, {*R*[1, 1], *R*[1, 2]}⟩
⟨*p2*, {*R*[1, 3]}⟩
⟨*p3*, {*R*[2, 1], *R*[2, 2]}⟩
⟨*p4*, {*R*[2, 2], *R*[2, 3]}⟩.

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

Fig. 3. Splitting Overlapping Ranges

Field Count Limit Check: One question that occurs is what if a DBMS has a limit of fields that can be used for a partitioning definition. (In the case of Teradata DBMS, up to 64 fields can be allowed for 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 fact table we use has much fewer fields than the limit, and thus, this step will not be executed.) To choose victim fields, our tool computes the weighted sum of *query cost*, which is discussed in Section 3.3, 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 incrementally discard a field with the largest sum of query cost until reaching the limit. Accordingly, we can update the existing *M*2. One might say that such an MLPPI definition may not be exploited if too many ranges over a partition count limit (65,536 in the case of Teradata DBMS) are found in one field. However, we assume that such an extreme case is not expected. Even if such a case happens, the merges of consecutive ranges can make the MLPPI feasible.

Query-to-Range-Set Map Construction: Using the existing maps *M1* and *M2*, the tool can create the final bi-directional query-to-range-set map, say *M*, between *Q* and *R*. To construct *M*, we leverage a transitive property from *M1* to *M2*. Once we compute *M*, there is no need to keep the intermediate maps (*M1* to *M2*) for the subsequent phases. In the running example, we can build *M* in the following.

M:
⟨*q1*, {*R*[1, 1], *R*[1, 2], *R*[2, 1], *R*[2, 2]}⟩
⟨*q2*, {*R*[1, 3], *R*[2, 2], *R*[2, 3]}⟩.

In the subsequent phases, we use for cost computation and then update *M* in accordance with a merge of ranges; *R* is refined for an MLPPI recommendation.

Partition Count Limit Check: 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 aforementioned partition count limit, then it is not possible to make a feasible MLPPI based on the 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 obtain a total of (number of ranges in LO_DISCOUNT) × (number of ranges in LO_QUANTITY) = (3+1) × (3+1) = 16 partitions. Note here that one reserved range covering the rest but the identified is added by default to each field in the calculation, and the default range is equivalent to the 'NO CASE OR UNKNOWN' case as shown in **Table 1**. Since the total partitions surpass the limit, the tool should go through the initial phase in which we make fewer partitions than the limit. (Otherwise, the wizard can bypass the initial phase and then immediately proceed to the optimized phase.)

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

3.2 The Initial Phase

This phase incrementally merges a range pair in *R* to reduce partitions. The merge continues until the number of ongoing partitions drops below the partition count limit. After finishing the merge, the tool can derive a feasible MLPPI with the remaining partitions.

Overall, the merge may increase I/O cost, in that the full scan on a partition should be carried out to retrieve all the rows potentially 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 a range pair for a merge, we calculate the scan cost of a query on the merged range pair.

Fig. 4 represents how the scan cost can be computed on the range pair that influences a given query. The scan cost represents the I/O cost to answer the query when the range pair is merged; it is defined as the number of blocks that are to be read for the given query. To compute the scan cost of a query *q*, we write a corresponding scan cost query, denoted by s, which can be constructed as follows:

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

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

Fig. 4. Computing Scan Cost

where *F* is a fact table such as LINE_ORDER, 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 range. The altered range set is remapped to *q* in *M* and then translated to the equivalent predicates. Then *CP* for s can be built based on the predicates. Unless the merge range pair influences *q*, we can simply derive *CP* from the existing ranges mapped to *q*.

To see in Fig. 4 how a scan cost query is built, 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 correspondingly altered to {(∞, 25), [25, 35]} and {[25, 35]}, respectively. Certainly, no change is made to the existing range sets of the queries on LO_DISCOUNT. Therefore, we can build the corresponding scan cost queries for *q1* and *q2*, as illustrated in Table 7. Note that because the altered ranges {(∞, 25), [25, 35]} of *q1* are not discontinuous, the merged, single predicate, which is 'l.LO_QUANTITY <= 35,' are built for *s1* as in the first row of Table 7.

In turn, the wizard sends each scan cost query to the server through an API call. The tool extracts from the server's response the *spool size* (in bytes), which is the result size of the sent scan cost query, and then calculates as scan cost the block counts (denoted by *blc*) using the spool size. If the merge of a range pair does not influence any query in the workload, the existing (previously computed) scan cost of queries is reused for the range pair. In other words, the wizard only re-computes the scan cost of a query only if the query is affected by the merge of two consecutive ranges. The weighted sum of scan cost of queries, T_s can be computed for each range pair. T_s can be defined in the following:

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

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

Table 7. The Constructed Scan Cost Queries

Scan Cost Query ID	*Query Text*

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)`

Once examining all range pairs, 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 the heuristic of favoring the range pair that produces the 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` = {[1, 1], [4, 5], [7, ∞)}
`LO_QUANTITY` = {(∞, 25), [25, 35]}

M:

⟨*q*1, {*R*[1, 1], *R*[1, 2], *R*[2, 1], *R*[2, 2]}⟩
⟨*q*2, {*R*[1, 3], *R*[2, 2]}⟩

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] ∩ *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. As the partitions formed by ((*R*[1, 1]∪*R*[1, 2])∩(*R*[2, 2])) contain the non-qualifying rows for q1, unfortunately, *R* cannot provide q1 with as much benefit as *q2*. *R*, however, 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*.

Fig. 5 describes the initial phase. Given a query-to-range set map (*M*) and an input range set (*R*), the algorithm determines which range pair to yield the least total execution cost (T_s) and updates *M* and *R* using the chosen range pair, until the number of the partitions by *R* falls below a pre-defined partition limit count. Finally, the algorithm produces the final recommendation based on *R*.

In the running example the total partition counts (or 4 × 3=12) by *R* is under the assumed limit (or 15), which meets the partitioning limit. Hence, the wizard can enter the optimized phase with *R*, without repeating the initial phase.

3.3 The Optimized Phase

After passing the initial phase, we now have an initial MLPPI recommendation based on *R*, for the `LINEORDER` fact table. Certainly, we can sufficiently use the initial recommendation. However, our emphasis is on the fact that additional merges may make fewer partitions, which actually can enhance the overall workload performance for the following reasons. First, multiple file contexts by many partitions can incur huge overhead that affects the query optimizer. There also exists an operational threshold that the optimizer can handle the

```
input: $M$ (query-to-range-set map) and $R$ (input range set)
output: $R$ with # partitions, fewer than or equal to a specified partition limit
$W \leftarrow$ Query weights
while $\leftarrow$ (# partitions by $R >$ partition limit) do
  foreach range pair, $rp \in R$ do
    $T_s \leftarrow 0$;
    foreach $q \in M$ do
      $w \leftarrow$ Get $q$'s weight from $W$
      $L \leftarrow$ Get the range set mapped to $q$ in $M$
      if (($rp \cap L) \neq \emptyset$) then
          $T_s$ += $w \cdot (getSC(q, rp, M))$; // Influenced
      else $T_s$ += $w \cdot$($q$'s existing scan cost) // See Alg. 3 regarding getSC()
    end
    Associate $T_s$ with $rp$.
  end
  Find $rp$(s) with the least $T_s$ .
  If a tie occurs, then choose the $rp$ to make fewer partitions when being consolidated.
  Update $M$ and $R$ by the chosen $rp$.
end
return $R$;
```

Fig. 5. The Initial Phase

maximum number of the partitions at a time. Thus, further reducing partitions greatly helps the optimizer to manage these partitions.

A similar algorithm, as suggested in the initial phase, is applied. Namely, this phase uses query cost, which is analogous to scan cost; we can obtain the query cost via a feasible MLPPI. The query cost is defined as the estimated processing time of a query on a so-called “faked” fact table applying the MLPPI definition that reflects the merge of a range pair. This cost estimation technique is similar to the “*what-if*” approach [8].

Fig. 6 illustrates the steps for computing the query cost of a query q. Given a range pair (rp) affecting q when merged, we first 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 on H. After that, we alter H by an MLPPI with the updated R (now say R’), applying the merge of rp. We then build a shadow query h that is equivalent to the input query q and replaces F in q with H. In turn, we make an API call to the `EXPLAIN` tool with h. Eventually, the algorithm extracts from the result and returns the estimated elapsed time of h as the query cost associated with q. As in the initial phase, the query cost of a query (like q) is recomputed only if the merge affects the query.

Fig. 7 proposes the algorithm of the optimized phase. W indicates a set of weights assigned to individual queries in a given workload. T_p keeps track of the existing least query cost sum, and initially is computed on an initial MLPPI. For each range pair, the algorithm computes T_q, the respective weighted sum of query costs of the queries in a given workload. (T_q can be similarly defined as T_s, described in Section 2.2, and thus the definition of T_q is omitted here.) Let the current least T_q be T. If $T <= T_p$ (or the existing least), then for the next iteration T_p is set to T. We perform the merge on the range pair that produces that T and update M and R along

```
input: $q$ (query), $rp$ (range pair), $M$ (query-to-range-set map), and $R$ (input range set)
output: Query cost to answer $q$ when a merge of $rp$ is considered

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).
Propagate all (field and index) statistics of $F$ to $H$.
if $F$ has materialized views (MVs) then
    Create the equivalent MVs on $H$.
    Propagate the statistics of the MVs on $F$ to those of $H$.
end
$r_m \leftarrow$ Merge $rp$;
$L \leftarrow$ Copy ranges associated with $q$ from $M$;
Remove ranges in $rp$ from and add $r_m$ to $L$.
$R' \leftarrow$ Update $R$ with $L$
Alter $H$ using a fictitious MLPPI with $R'$.
Construct and send to the server an $H$-based shadow query $h$ equivalent to $q$.
$ept \leftarrow$ Estimated processing time of $h$ extracted from the server response;
Update $q$'s query cost to $ept$.
return $ept$
```

Fig. 6. Computing Query Cost

```
input: $M$ (a query-to-range-set map) and $R$ (an input range set)
output: an MLPPI solution to produce the minimum total query cost

$W \leftarrow$ Query weights
$T_q \leftarrow$ Total query cost sum on an initial MLPPI by $R$;
while $\leftarrow$ (Pre-defined iterations || $R \neq \emptyset$) do
    foreach range pair, $rp \in R$ do
        $T_q \leftarrow 0$;
        foreach $q \in M$ do
            $w \leftarrow$ Get $q$'s weight from W
            $L \leftarrow$ Get the range set mapped to $q$ in $M$
            if (($rp \cap L) \neq \emptyset$) then
                $T_q$ += $w \cdot (getQC(q, rp, M, R))$; // Influenced
            else $T_q$ += $w \cdot$($q$'s existing query cost) // See Fig. 6 on getQC()
        end
        Map $T_q$ to $rp$.
    end
    $T \leftarrow$ The least $T_q$ that has been so far seen.
    if $T > T_p$ then break;
    else
        $T_p \leftarrow T$;
        Find $rp$(s) with $T$.
        If a tie occurs, then choose the $rp$ to make fewer partitions when being consolidated.
        Update M and R by the chosen $rp$.
    end
end
return an MLPPI by $R$;
```

Fig. 7. The Optimized Phase

with the merge. The algorithm repeats until (i) there are no more ranges in R left, or (ii) the number of specified iterations is reached. If $T > T_p$, then this optimization process is over.

To break a tie between range pairs, we apply the same heuristic to favor one of the range pairs with fewer partitions. The order of the different partitioning levels may influence 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 the number of partitions in descending order before making the final recommendation. The solution in **Table 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 of the output in **Fig. 1** can go either way since both levels have the same number of partitions.

In the running example, suppose that in the first round, for instance, a range pair of $R[1, 1]$ and $R[1, 2]$ are chosen for a merge. If so, then the wizard produces a merged range, or [1, 5] under `LO_DISCOUNT`; it 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. The output of the tool is represented on the bottom of **Fig. 1**, and it corresponds to the final MLPPI recommendation for the given workload Q.

4. Complexity Analysis

We now conduct complexity analysis of the proposed algorithms: preprocessing, initial, and optimized phases. We use as the metric on each phase the logical running time of the wizard, equivalent to the number of API calls made to the database server during that phase.

4.1 The Preprocessing Phase

The following Lemma is the analysis on the running time of 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. Consider the 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 one value per field can be mapped to each query. Thus, the running time complexity for the map construction is is $O(N \cdot C \cdot V)$.

But our experiments demonstrate that the bound of $O(N \cdot C \cdot V)$ is overly pessimistic.

4.2 The Initial and Optimized Phases

The following is the running time complexity on each of the initial and optimized phases.

Lemma 2. *The running time complexity for the initial (or optimized) phase is $O(M^2 \cdot N)$, where M is the number of range pairs, and N is the number of queries in a 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, we pay the re-computation cost of $M \cdot N$ and merge a victim range pair. In the subsequent round, (M-1) range pairs remain, so the cost amounts to $(M\text{-}1) \cdot N$. In the worst case, we end up consolidating all range pairs, thus having no partition on the target table. The number of total calls made to the server can be added up as follows:

$$M{\cdot}N + (M\text{-}1){\cdot}N + \cdots + N = (\sum_{i=1}^{M} i){\cdot}N = (M\,(M - 1) / 2)\ {\cdot}N$$

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

Our experiments also showed that the above running time complexity was overly pessimistic. Specifically, the number of calls made in our experiments was by far fewer than that bound. That was because it was very rare for a range pair to involve all queries in a given workload (as will be seen in **Table 8**).

5. Experiments

In this section we evaluate the performance of our wizard using the proposed algorithms. We first describe our environment settings. We then report the statistics measured dur ing our experiments. Finally, we compare the performance of the recommendation by the MLPPI wizard (WIZARD) with (1) that of no partitioning (NO PPI) and (2) the partitioning of a human expert (EXP) under different setups.

5.1 Environment Settings

Development: We implemented our MLPPI wizard as a prototype on top of the Teradata DBMS server. We developed our wizard in Java. We evaluated the performance of the wizard on the Teradata DBMS server machine running UNIX.

Workload Generation & Deployment: For evaluation, we used synthetic query workloads on the SSB mentioned in Section 1. Specifically, to generate the query workloads we built a simple *star schema query-generator*, which assumes that it already knows (i) available operators, (ii) fields of the LINEORDER fact table, and (iii) the minimum and maximum values of the fields. To build each query, the generator selects a random number to specify how many single-table predicates should be created. In turn, the generator randomly chooses a field and a specific operator for each predicate. If the IN operator on a chosen field is picked up, the query generator determines how many values should be added in the IN predicate and then put as many random values in the range of the field as the determined value count. In this way, the generator produces a random query involving the generated predicates.

An individual query is generated based on a template that joins LINEORDER and DDATE with constraints defined by the predicates. This template is common, realistic, in real customer cases like reports and form templates. We generated two kinds of workloads consisting of 10 queries (10Q) and 20 queries (20Q). Each query in the workload is similar to *q1* in the **Table 3**.

Note that even though the generated 10Q and 20Q workloads are synthetic, they are issued on the SSB. As specified in the article [3], the SSB is a modified version of the TPC-H benchmark [29], which is most popular in measuring the performance of analytical workloads on a database server in a practical environment. Therefore, the used query workloads do not lose their generality in a real scenario.

Again, our algorithm is applicable on any other analytical workload as long as the workload includes query predicates in general form on a fact table. The “predicates” extracted from that workload populates the search space, with which the wizard can start. Those who are interested in reproducing our results can try a workload consisting of queries similar to ones in **Table 3** or an analytical workload extracted from a real environment [27, 28].

Finally, we ran the generated 10Q and 20Q workloads on a fact table populated with a scale factor of 1TByte (1TB) and 3TBytes (3TB).

5.2 Statistics Report

Table 8 is a summary of the execution statistics measured while the wizard produced partitioning solutions for the workloads. The statistics includes (i) input partitions, (ii) range pairs, (iii) the number of API calls made, and (iv) total iterations observed in each phase.

Roughly 1 million partitions were initially derived for 10Q and roughly 0.11 billion partitions for 20Q (Item #1). The workload size doubled from 10Q to 20Q, but surprisingly the total number of partitions was in an exponential growth. Much more predicates with different bindings produced about 110x more ranges in 20Q than those of 10Q. While a great number of as shown in the third row (Item #2) in **Table 8**.

The total number of iterations in the initial phase tended to be proportional to the input partitions generated from the 10Q and 20Q workloads. Note that the optimized phase ran only once for those two workloads. Since the partitions produced by the initial phase were customized enough, there was no need to iterate for further optimization.

As mentioned in Section 4, for each workload the number of the total calls to the server was extremely small, compared to that of the theoretical worst calls shown in the parentheses (Item #3). The worst call counts were calculated regarding the number of queries and range pairs observed in each phase. Furthermore, the average number of calls per range pair was limited within 10 across the workloads and phases (Item #4).

Finally, the per-iteration numbers of the average, minimum, and maximum range pairs proportionally increased along with the growing workload size (Items #5−#7), while the per-iteration numbers of the average, minimum, and maximum API calls increased by more than a scale factor of 2x (Items #8−#10). Nevertheless, empirically the number of invoked API calls was much fewer than that of the theoretical bound.

5.3 Performance Evaluation

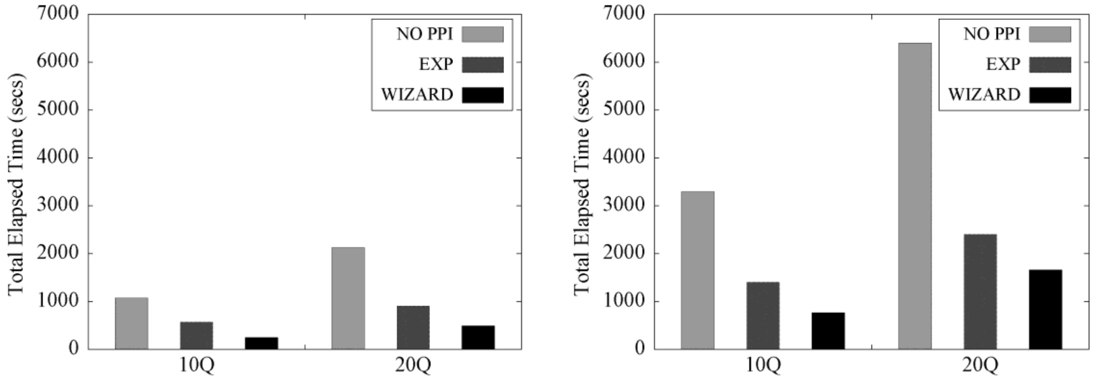
The focus of our evaluation is to see the performance (or quality) of MLPPI recommendations made by the MLPPI wizard (WIZARD), compared with those of no partitioning (NO PPI) and partitioning by a human expert (EXP).

Fig. 8 exhibits our evaluation results. Specifically, it shows the time taken to run the 10Q and 20Q workloads over the 1TB and 3TB fact tables that we populated into the database server. Overall, we found that the WIZARD recommendations were "very effective" at partitioning the fact tables via the predicates in the given workloads. First, regarding the total execution time for the 10Q workload on the 1TB table, WIZARD significantly outperformed NO PPI and EXP by about a factor of four and two, respectively, as illustrated in **Fig. 8(a)**. More specifically, it took 1076 seconds with NO PPI (the leftmost bar) and 569 seconds with EXP (the middle bar), but only 247 seconds with WIZARD (the rightmost bar) for 10Q. Such outperformance persisted even when we doubled the workload size (to 20Q). As shown in in **Fig. 8(b)**, WIZARD reduced the total execution time for 10Q on the 3TB table by almost four times and twice, compared with NO PPI and EXP. The result was almost same; that is, WIZARD spent only 1,663 seconds (the rightmost bar) to execute 20Q on the 3TB table while 6,399 seconds with NO PPI (the leftmost bar) and 2,402 seconds with EXP (the middle bar).

We further compared the performance of WIZARD and EXP by computing their partitioning efficiency, relative to that of NO PPI. Given the time taken for NO PPI set to be 1, we calculated the efficiency as reduction factor of the WIZARD and EXP solutions.

Table 8. The Runtime Statistics Obtained on Used Workloads

Item	The Initial Phase		The Optimized Phase	
	10Q	**20Q**	**10Q**	**20Q**
1. The number of input partitions	1,008,000	110,739,200	51,840	59,520
2. The number of input range pairs	441	4,180	24	48
3. The number of total API calls (The number of worst calls by Lemma 2)	1,305 (1,944,810)	31,915 (349,448,000)	78 (5,760)	388 (46,080)
4. The number of average API calls per range pair	3	8	3	8
5. The number of average range pairs compared per iteration	32	76	24	48
6. The maximum # of range pairs compared per an iteration	38	103	24	48
7. The minimum # of range pairs compared per an iteration	25	49	24	48
8. The number of average API calls made per iteration	93	580	78	388
9. The maximum # of API calls made per an iteration	123	791	78	388
10. The minimum # of API calls made per an iteration	73	381	78	388



(a) 1TB (b) 3TB

Fig. 8. Comparison of the Total Elapsed Time

Table 9 shows the reduction factor for each solution on a certain configuration. For instance, the value, '0.47,' at the column in the third row, indicates that about 47% of the time taken for NO PPI was reduced by EXP for 10Q on the 1TB table. We observed that WIZARD consistently outperformed EXP in any combination of the configurations. As exhibited in **Table 9**, the WIZARD solution had a higher factor of reduction than that of EXP on the 1TB table. For 10Q, 77% of the total time on NO PPI was reduced, and almost the same efficiency (76%) persisted when the workload size increased to 20Q.

Moreover, the partitioning efficiency by WIZARD was higher than that of EXP even for the fact table increased by three times (3TB). That is, the WIZARD's factor was still unchanged (around 0.77 for 10Q and 0.74 for 20Q) while still greater than that of EXP, as illustrated in the last column at the three and fourth rows in **Table 9**. We observe from these results that WIZARD solution betters the solution of EXP. In addition, we witness that the performance of the proposed tool scales very well with a combination of growing workload and table size.

That said, it is left as future work to apply the proposed tool on a more realistic, complex, workload over a further growing fact table. We anticipate that the automated multi-level partitioning provided by our tool will still outperform any other candidate in such a real scenario.

In sum, we confirm from our experiment results the effectiveness and efficiency of the partitioning recommendation produced by the proposed tool.

Table 9. Reduction factor (partitioning efficiency) of EXP and WIZARD on NO PPI; the greater, the more efficient

Fact Table Size	1TB		3TB	
Partitioning Workload Size	EXP	WIZARD	EXP	WIZARD
10Q	0.47	0.77	0.57	0.77
20Q	0.58	0.76	0.62	0.74

6. Conclusion

In this article, we presented the novel physical database design tool, called MLPPI wizard, to recommend a partitioning solution for a given analytical query workload on a huge fact table. Since the proposed tool exploits query-predicates to capture necessary ranges for fields of the fact table, the algorithms of the tool are easily applicable to any clustering and partitioning scheme in a relational DBMS. In addition, the final partitioning solution by the proposed tool can be customized for the given workload, as the definition includes fine-grained partitions concerning the extracted predicates. We analyzed the running complexity for the proposed algorithms. We found that in practice the tool makes much fewer API calls than the high bound in theory. Finally, we demonstrated in our experiments that the produced MLPPI solutions by the wizard reduced the total elapsed time up to by more than a factor of four, compared with existing solutions. The performance of the solutions also scaled very well with a combination of growing workload and fact table size. To conclude, the proposed wizard is a profitable option for a DBA to be able to draw with little overhead a best partitioning to minimize the execution cost of the given workload on the fact table.

References

[1] Sinclair, P., “Using PPIs to Improve Performance,” August 31, 2015. Article (CrossRef Link)

[2] Klindt, J., “Single-level and Multilevel Partitioned Primary Indexes,” July 15, 2015. Article (CrossRef Link).

[3] O’Neil, P., O’Neil, B., and Chen, X., “The Star Schema Benchmark (SSB),” July 15, 2017. Article (CrossRef Link).

[4] Agrawal, S., Narasayya, V., and Yang, B., “Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design,” in *Proc. of SIGMOD*, pp. 359–370, 2004. Article (CrossRef Link)

[5] Lightstone, S.S. and Bhattacharjee, B., “Automated Design of Multi-dimensional Clustering Tables for Relational Databases,” in *Proc. of VLDB*, pp. 1170–1181, 2004.

[6] Nehme, R. and Bruno, N., “Automated Partitioning Design in Parallel Database Systems,” in *Proc. of SIGMOD*, pp. 1137–1148, 2011. Article (CrossRef Link)

[7] Suh, Y.K., Ghazal, A., Crolotte, A., and Kostamaa, P., “A New Tool for Multi-level Partitioning in Teradata,” in *Proc. of CIKM*, pp. 2214–2218, 2012. Article (CrossRef Link)

[8] Chaudhuri, S. and Narasayya, V., “AutoAdmin ‘What-If’ Index Analysis Utility,” *SIGMOD Record*, vol. 27, no. 2, pp. 367–378, 1998. Article (CrossRef Link)

[9] Finkelstein, S., Schkolnick, M., and Tiberio, P., “Physical Database Design for Relational Databases,” *ACM Trans. on Databas. Syst.* vol. 13, no. 1, pp. 91–128, 1988. Article (CrossRef Link)

[10] Labio,W., Quass, D., and Adelberg, B., “Physical Database Design for Data Warehouses,” in *Proc. of ICDE*, pp. 277–288, 1997. Article (CrossRef Link)

[11] Rozen, S. and Shasha, D., “A Framework for Automating Physical Database Design,” in *Proc. of VLDB*, pp. 401–411, 1991.

[12] Oracle Corp., “Partitioning Advisor,” February 2, 2017. Article (CrossRef Link).

[13] Oracle Corp., “SQL Access Advisor,” March 2012. Article (CrossRef Link).

[14] Rao, J., Zhang, C., Megiddo, N., and Lohman, G., “Automating Physical Database Design in a Parallel Database,” in *Proc. of SIGMOD*, pp. 558–569, 2002. Article (CrossRef Link)

[15] Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, and V., Syamala, M., “Database Tuning Advisor for Microsoft SQL Server 2005,” in *Proc. of SIGMOD*, pp. 930–932, 2005. Article (CrossRef Link)

[16] Agrawal, S., Chaudhuri, S., and Narasayya, V.R., “Automated Selection of Materialized Views and Indexes in SQL Databases,” in *Proc. of VLDB*, pp. 496–505, 2000.

[17] Dash, D., Polyzotis, N., and Ailamaki, A., “CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads,” in *Proc. of PVLDB*, vol. 4, no. 6, pp. 362–372, 2011. Article (CrossRef Link)

[18] Kimura, H., Narasayya, and V., Syamala, M., “Compression Aware Physical Database Design,” in *Proc. of PVLDB*, vol. 4, no. 10, pp. 657–668, 2011. Article (CrossRef Link)

[19] Microsoft Corp. “Microsoft SQL Server 2000: Index Tuning Wizard SQL Server 2000,” Article (CrossRef Link) (viewed on March 1, 2012).

[20] Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G., and Skelley, A., “DB2 Advisor: An Optimizer

[21] Smart Enough to Recommend Its Own Indexes,” in *ICDE*, pp. 101–110, 2000. Article (CrossRef Link)

[22] Zilio, D.C., Zuzarte, C., Lohman, G.M., Pirahesh, H., Gryz, J., Alton, E., Liang, D., and 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. Article (CrossRef Link)

[23] Zilio, D.C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., and Fadden, S. “DB2 Design Advisor: Integrated Automatic Physical Database,” in *Proc. of VLDB*, pp. 1087–1097, 2004. Article (CrossRef Link)

[24] Tatarowicz, A.L., Curino, C., Jones, E.P.C., and Madden, S., “Lookup Tables: Fine-grained Partitioning for Distributed Databases,” in *Proc. of ICDE*, pp. 102–113, 2012. Article (CrossRef Link)

[25] Idreos, S., Kersten, M.L., and Manegold, S., “Database Cracking,” in *Proc. of CIDR*, pp. 68–78, 2007.

[26] Wang, J., Hang, S., Liu, J., Chen, W., and Hou, G., “Multi-level Scheduling Algorithm Based on Storm,” in *Proc. of TIIS*, vol. 10, no.3, 2016. Article (CrossRef Link)

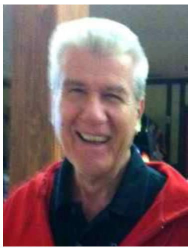
[27] Yuan, Y., Chen, W., Han, G., Jia, G., “OLAP4R: A Top-K Recommendation System for OLAP Sessions,” in *Proc. of TIIS*, vol. 11, no.6, 2017. Article (CrossRef Link)

[28] Zahra Pooranian, Mohammad Shojafar, Bahman Javadi, Ajith Abraham, “Using Imperialist Competition algorithm for Independent Task Scheduling in Grid Computing,” *Journal of Intelligent and Fuzzy Systems*, vol. 27, no. 1, pp. 187-199, 2014. Article (CrossRef Link)

[29] Zahra Pooranian, Mohammad Shojafar, Reza Tavoli, Mukesh Singhal, Ajith Abraham, “A Hybrid Metaheuristic Algorithm for Job Scheduling on Computational Grids,” *Informatica* (Slovenia) vol. 37, no. 2, pp.157-164, 2013.

[30] TPC-H, September 29, 2017. Article (CrossRef Link).

Young-Kyoon Suh is an Assistant Professor in the School of Computer Science and Engineering (CSE) at Kyungpook National University (KNU), Daegu, Republic of Korea. He received MS and PhD degrees in computer science from KAIST, Korea, and the University of Arizona, USA, in 2005 and 2015, respectively. Prior to joining the CSE faculty at KNU, he was a senior researcher at Korea Institute of Science and Technology Information (KISTI), Daejeon, Korea. His research interests include database design and systems, science of computing, big data analytics, and HPC.



Alain Crolotte received a PhD degree from University of California at Los Angeles (UCLA). He was a technical consultant at Abacus Programming Corporation. He then moved to Teradata Corporation, where he worked as senior benchmark consultant. He is now a performance specialist at Teradata. His research focuses on big data systems, query optimization, and physical database design.

Pekka Kostamaa is a big data architect at OpenX. He was a Vice President (VP) of Engineering and Big Data Lab at Teradata Aster before joining OpenX. Prior to VP, he was a director of Product Management at Teradata Corporation. He also worked as a director in the Department of Advanced Development at the same company. He is interested in query optimization and physical database design.