Stanchev, Weddell. Published in Data and Knowledge Engineering, 69(10). October, 2010. 1062-1080

Saving Space and Time Using Index Merging

Lubomir Stanchev[a,], Grant Weddell[b]

[a] *Computer Science Department, Indiana University - Purdue University Fort Wayne, USA*

[b] *David R. Cheriton School of Computer Science, University of Waterloo, Canada*

Abstract

Managing digital information is an integral part of our society. Efficient access to data is supported through the use of indices. Although indices can reduce the cost of answering queries, they have two significant drawbacks: they take additional storage space and their maintenance can become a bottleneck. We address these challenges by introducing search data structures that reduce the need for storing redundant data among indices. Our experimental results with the main-memory version of these data structures show that our approach can reduce by half the storage space and can improve performance, where the highest performance improvement is achieved for workloads with high update ratios. Our experimental results with the secondary-storage version of the data structures shows that our approach produces a solution that can outperform both IBM DB2 and Microsoft SQL Server on the popular TPC-C workload.

Email addresses: `stanchel@ipfw.edu` (Lubomir Stanchev), `gweddell@uwaterloo.ca` (Grant Weddell)

1. Introduction

Efficient access to vast amount of data, whether stored on secondary disk or kept in main memory, is possible through the use of cleverly designed auxiliary data structures, such as indexes, hash tables, and materialized views. Although these data structures can improve access time exponentially, they require additional storage space and additional maintenance overhead. This paper shows how index structures can be merged by reducing the redundant data among them. This results not only in savings in storage space, but also improved performance due to reduced maintenance overhead.

Consider a workload of ten queries defined over the same table. An index advisor, such as the one available in IBM DB2 ([16]), Microsoft SQL Server ([2]), and Oracle ([10]), can suggest creating an index for each query in order to improve performance. This will result in substantial increase of storage overhead. Moreover, every update to the table needs to be accompanied with updates to each of the ten indices, which can become a performance bottleneck. Conversely, if some of the suggested indices are merged in a way that reduces redundant data among them, then less storage space will be needed and updates will be faster because fewer copies of the same data will have to be refreshed.

Index merging is a difficult problem and most commercial database management systems (DBMSs) provide only limited support ([5]). The reason is that, except for the most trivial cases, it is impossible to perform index merging in a way that preserves the set of queries that can be answered efficiently without creating instances of data structures that are more evolved than a traditional index. For example, consider the indices $X_1 = \langle R_1, \langle A \texttt{ asc}\rangle\rangle$ [1], $X_2 = \langle R_2, \langle A \texttt{ asc}\rangle\rangle$, and $X_3 = \langle R_3, \langle A \texttt{ asc}\rangle\rangle$. Moreover, suppose that all three tables have the same set of attributes and R_2 and R_3 are disjoint materialized views that contain a subset of the elements of R_1. One may identify the indices X_2 and X_3 as redundant because they contain a subset of the data that is already stored in X_1. However, removing the two indices will prevent us from efficiently answering the query "`select * from` R_2 `where` $A > 5$" because index X_1 cannot be used to efficiently enumerate the elements of R_2. In this paper we show how the three indices can be merged into a single *extended index*[2] on the elements of R_1 that can efficiently answer all

[1] This denotes an index on the table R ordered by the attribute A is ascending order.

[2] An extended index is a tree index that has additional capabilities that allow for data

the queries that the initial tree indices can answer.

Different index structures have been known for more than forty years. For example, AVL trees were first introduced in 1962 (see [1]), while B+ trees were introduced in 1972 (see [4]). However, few studies have considered the possibility of merging indices in order to eliminate redundant data. Two of the few exceptions are [9] and [5], which consider merging indices whenever they have attributes in common. However, unlike our approach, the papers' approach can lead to exponential degradation in query performance. The reason is that a query that has a worst-case logarithmic time-bound against one of the indices to be merged can have linear worst-case complexity against the merged index. This happens when the merged index no longer efficiently supports one or more of the initial queries.

In the paper we examine how indices can be merged in a way that does not affect the space of queries that can be efficiently answered. The goal is to preserve the initial intention of the index advisor about the queries that need to run efficiently. This restriction results in less index merging opportunities compared to the approach taken in [9, 5]. However, our experimental evaluation shows that our approach can work well in practice, resulting in a significant reduction of storage cost and improved performance for workloads with with significant update ratio (e.g., above 10% updates). The main-memory experimental results in the paper are based on Arne Andersson trees (or AA trees - see [3]), while the secondary-storage experiments use the B+ tree implementation of the commercial system that is being compared.

1.1. Our Approach

We adopt the model where the physical design advisor of a DBMS produces a set of parameterized simple SQL queries (or sSQL queries for short - see Table 3 for a formal definition) over existing base tables and newly recommended materialized views. Such an output can be produced, for example, by examining the operation tree of the queries in the workload and identifying leaf subtrees for which the benefit of creating an index and/or a materialized view outweighs the cost of maintenance (see [12] for details). Alternatively, if the *what-if query optimizer* model presented in [2] and [16] is applied, then the what-if query optimizer can easily estimate the cost of a SQL query based on information about the cost of sSQL queries. Finally,

---

compactness, where the precise definition will be presented in Section 3.

(*name*)	(*query*)
Q_1	`select *` `from` *Employee* `where` *SSN* = :*P*
Q_2	`select *` `from` *Department* `where` *depName* = :*P*

Table 1: Breakup sSQL queries

even if the physical design tuning is done online without user input (see [6]), the same set of requirements can be identified.

In order to demonstrate how sSQL queries are created, consider the following query and assume that every employee works for a single department.

> `select` *
> `from` *Employee e*, *Department d*
> `where` $e.depName = d.name$ and $e.SSN = {:}P$

In the query, $:P$ is used to denote a parameter, *SSN* is a key for the table *Employee*, and *depName* is a key for the table *Department*. When constructing a query plan for answering the query, the query engine can determine that the queries shown in Table 1 are needed to efficiently support the initial query. In this case, our system will take the sSQL queries Q_1 and Q_2 as input. Using these sSQL queries, the original query can be answered using the following query plan.

> *Employee e* $= Q_1(:P)$;
> *Department d* $= Q_2(e.depName)$;
> `return join`(e, d);

We assume that the query optimization that selects the sSQL queries happens outside our system and is done by an external tool. Note that the input to our system includes sSQL queries rather than indices because sSQL queries carry more detailed information. For example, the index $\langle R, \langle A$ `asc`$, B$ `asc`$\rangle\rangle$ does not tell us whether the order of the attributes can be swapped without sacrificing the logarithmic time-bound for object retrieval. In particular, the order of the attributes is not important if the sSQL query "`select * from` R `where` $A =\ :P_1$ `and` $B =\ :P_2$" generated the index, but it is important

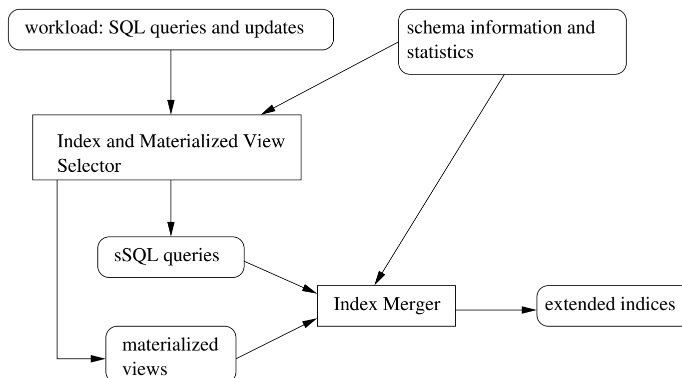

Figure 1: Physical design advisor architecture

if the SQL query "`select * from` R `order by` A `asc`, B `asc`" generated the index.

The core of the paper is the *Index Merger* module shown in Figure 1. It takes as input sSQL queries defined over base tables and newly introduced materialized views, schema information, and statistics on the current state of the database. The result of the module is a set of extended indices and a mapping between each input sSQL query and the extended index that can efficiently answer it. Statistical information is used to estimate the size of an extended index.

The first step of our algorithm is to create a *Parameterized Access Requirement Type* (PART) for each input sSQL query, where a PART represents a set of extended indices. In particular, for an input sSQL query Q, we will create the PART $\mathcal{P}$ that corresponds to a set of extended indices that can efficiently answer Q by doing a single index scan, where we require that the extended indices are minimal in the sense that they cannot be simplified and still effetely answer the query Q. The second step is to merge PARTs that represent indices that have a non-empty intersection. The merging has the property that the initial queries can be efficiently answered by each extended index that is represented by the created PART. The final step in both algorithms is to create an extended index for each of PARTs that is constructed in the previous step, where the extended indices that are anticipated to be of the smallest size given the available statistics are selected.

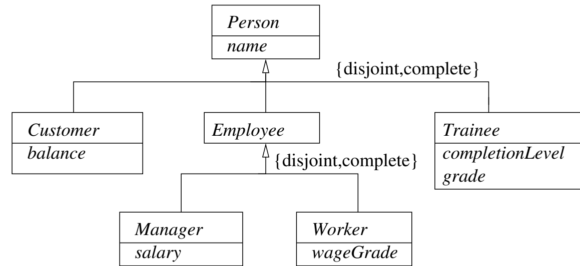

Figure 2: Example database schema

(*name*)	(*query*)

Q_1	`select *`
	`from` *Person*
	`order by` *name* `asc`

Q_2	`select *`
	`from` *Customer*
	`where` *name* = :P_1
	`order by` *balance* `asc`

Q_3	`select *`
	`from` *Trainee*
	`where` *completionLevel* = 1
	`order by` *name* `asc`, *grade* `asc`

Table 2: Three example queries

1.2. Motivating Example

We next demonstrate our approach on the UML class diagram shown in Figure 2. It depicts a typical company that has customers, employees, and trainees that are disjoint and every employee is a manager or a worker, but not both. We have picked a UML class diagram rather than a relational schema as out data schema in order to simplify the presentation. However, the algorithm works on any relational or object-oriented database schema. For example, the relational database that corresponds to the example schema can contain the materialized views *Customer*, *Employee*, and *Trainee* defined over the base table *Person* and the materialized views *Manager* and *Worker* defined over the materialized view *Employee*. If the proper integrity constraints are specified, then our algorithm can be applied on this modified relational database schema. It is also the case that inheritance between

classes (or set containment between tables in the relational case) is not required in order to reap the full benefit of our algorithm. For example, it may be the case that the initially data schema can contains only the table *Person*, where the other entities are generated as the result of the queries in the workload (for example, a query that asks about persons of type trainee could have generated the materialized view *Trainee*).

Suppose that the workload consists of the example queries shown in Table 2. Note that queries Q_1 and Q_2 follow the sSQL syntax, while query Q_3 can be rewritten as a sSQL query over the materialized view V_T with the following underlying query.

```
select *
from Trainee
where completionLevel = 1
```

We next describe an extended index that can efficiently answer all three queries. The search tree for the index will contain nodes that contain pointers to *Person* objects (that is, one can perceive this index as a secondary index on the *Person* table). Since the class *Person* contains objects of four different types (that is, *Customer*, *Manager*, *Worker*, and *Trainee*), the index will contain pointers to objects of four different types, which demonstrates the *polymorphic* property of an extended index.

The nodes in the search tree of the index are first ordered relative to the attribute *name* in ascending order and next relative to the derived attribute A, where $A = 0$ for *Customer* objects, $A = 1$ for *Employee* objects, and $A = 2$ for *Trainee* objects. Next, the ordering depends on the value for the attribute A. Nodes that point to objects that have the same value for the *name* attribute and for which $A = 0$ are ordered relative to the attribute *balance* in ascending order. Alternatively, nodes that point to objects that have the same value for the *name* attribute and for which $A = 2$ are ordered relative to the attribute *grade* in ascending order. The presented ordering demonstrates the *branching order* property of an extended index.

Lastly, we are going to add a *marker bit* to each node of the search tree of the index. The bit of a node will be set exactly when the node or one of its descendent nodes contains a pointer to a *Trainee* object for which *completionLevel* $= 1$.

The extended index that was described can be used to efficiently answer query Q_1 from Table 2 by performing an in-order traversal of the search tree.

This will result in computing the correct query result because the nodes of the index are ordered relative to the attribute *name*.

Query Q_2 from Table 2 can be efficiently answered by first finding the left-most node in the search tree that points to a *Customer* object with the given name. This can be done efficiently because nodes for people with the same *name* are ordered relative to their type. The query result consists of the objects pointed to by sequential nodes in the search tree starting with the found node, where the terminating condition is reaching a node that points to an object that is not a *Customer* or that has a *name* that is different from the specified name.

Query Q_3 can be efficiently answered by performing an in-order traversal of the marked nodes of the search tree. In particular, any subtree with a root node that is not marked can be pruned-out because such a subtree cannot contain a pointer to a *Trainee* object for which $completionLevel = 1$. Conversely, any subtree that has a root node that is marked will contain a pointer to an object from the query result and therefore needs to be examined. Note that the resulting objects will be in the correct order because the nodes in the search tree that point to *Trainee* objects are first ordered relative to *name* in ascending order and then relative to *grade* in ascending order.

1.3. Paper Outline and Contributions

Chapter 2 presented related research, while Chapter 3 outlines relevant definitions. The most significant contributions of the paper are presented in the next three chapters.

1. We present the notion of an extended index, which is a novel data structure that can contain data from several indices in a way that reduces redundancies - see Section 4.

2. We show how extended index can be merged - see Section 5.

3. We present experimental results that show how our approach to index merging can decrease storage overhead and speedup updates - see Section 6.

Chapter 7 summarizes our results and outlines directions for future research.

2. Related Research

Two papers that address the problem of index merging are [9] and [5]. Given the indices $\langle R, \langle A \rangle\rangle$ and $\langle R, \langle B \rangle\rangle$, the algorithms in both papers can decide to merge them into the index $\langle R, \langle A, B \rangle\rangle$. This differs from our approach in two ways: (1) indices rather than sSQL queries are part of the input and (2) the exponential time capabilities of the initial indices are not preserved. For example, if all the objects in R have different values for A, then the new index can take linear time to answer the query "`select * from` R `where` $B = :P$". However, this query can be efficiently answered by the second input index. The reason the papers' approach cannot overcome this shortcoming is because they do not explore index merging techniques that generates index structures that are different from traditional indices. We, on the other hand, explore extended indices, which have the polymorphic, branching order, and marker bit properties. To summarize, our approach is orthogonal to the approach presented in the two papers and can be applied in combination with them.

Merging artifacts and the order of performing the merging procedure has been considered for different applications. For example, [8] examines how image regions can be merged and how the order of merging can affect the final result. However, their approach is not applicable here because indices are simpler artifacts than images and more precise merging techniques can be developed.

Other ways to reduce index maintenance cost have been explored. For example, [11] explains how to reduce index maintenance cost by using adaptive indexing. Adaptive indexing allows updates to be propagated fast, but index retrieval produces "candidate" and not "certain" query results. Another proposal is [13]. It is one of a sequence of papers on *cracked databases*, where updates are applied just before the required data is queried. We believe that this work is orthogonal to our proposal. An interesting approach is described in [6]. It shows how physical design selection can be done on the fly while the database is queried and updated rather than being invoked manually. Again, we believe that this work complements this paper.

It is important to note that this paper does not address the problem of automatic physical design (see [2, 5, 7]). Our approach assumes that access requirements (described as sSQL queries and materialized views) have already been determined before our algorithm is applied.

3. Definitions

In this section we describe the database schema type, the query language, and the problem that we are solving. Note that we fix the database schema type in order to increase the readability of the paper, where our algorithm can be applied to both a relational and object-oriented database schema.

3.1. Database Schema

Throughout the paper we will use the unqualified term *table* to refer to both base tables and materialized views, where a base table defines the set of objects that are instances of a particular class. We will use T to denote a base table, V to denote a materialized view, R to denote a table, and Σ to denote a database schema. We use the letters A and B to refer to table attributes and `attr`(R) to refer to the attributes of the table R.

We require that every table has the system attribute `ID` that uniquely identifies a database object. The non-system attributes of a table are either *non-reference* and are of one of the predefined types (e.g., integer, string, etc.), or are *reference* and store the `ID` of an object in the database. We require that all reference attributes are not null and refer to an existing object, that is, we impose a foreign key constraint on reference attributes.

We will refer to a materialized view that is defined using a query of the following type as a *simple materialized view*.

`select` $A_1, \ldots, A_a$
`from` T as t
`where` $\gamma(t)$

In the above query γ is used to denote an *efficient predicate*, where the precise definition follows.

Definition 3.1 (efficient predicate). *An efficient predicate γ over a table R has the property that it can be decided in $O(|def(\gamma)| \cdot |t|)$ time whether the predicate holds for an object $t \in R$.*

Note that throughout the paper we use $|\cdot|$ to denote the size of the enclosed component and $|def(\cdot)|$ to denote the size of its definition. The predicate ($t.name$ =“`John`” `and` $t.salary > 200000$) is an example of an efficient predicate. It can be used to define a simple materialized view with the following underlying query.

```
select *
from Manager as t
where t.name = "John" and t.salary > 200000
```

Simple materialized views are important because they reuse the ID attribute of the underlying tables over which they are defined. For example, in the above materialized view managers named John that make more than two hundred thousand dollars can be identified as such (e.g, by connecting them in a linked list or creating an index on them) without the need to create additional records for them.

3.2. The Query Language

We assume that the the input queries to the Index Merger module (see Figure 1) are sSQL queries – see Table 3. In the table we have used $[\cdot]$ to denote an optional component and *dir* to denote `asc` or `desc`. The restrictions for sSQL queries prevents ordering on reference attributes. This is reasonable because the value of a reference attribute depends on the the internal implementation of the system and should not be relied on by external users. The restriction also enforces partial-match attributes to be non-`ID`. A query in which one of the partial-match attributes is an `ID` attribute can be answered by executing a query of the third type followed by a predicate check on the resulting object. We have chosen this sSQL syntax because it restricts input queries to single table queries that can be efficiently answered using a single index. Adhering to the SQL standard, we will use "`select` *" to denote selecting all the attributes of a table.

3.3. The Problem

We next define the characteristics of an efficient query plan.

Definition 3.2 (efficient plan for a query). *Consider a SQL query Q and the corresponding access plan Q_P. Assume that the size to encode a value for each of the attributes of the database schema is constant. Then the query plan Q_P is* efficient *exactly when it returns each object of the query result in $O(|def(Q)| \cdot (\sum_{i=1}^{m} log(|R_i|)))$ time, where $\{R_i\}_{i=1}^{m}$ are the the tables that are referenced in Q.*

(*type*)	(*query*)
(1)	select $B_1, \ldots, B_b$ from R [where $A_1 = {:}P_1$ and $\ldots$ and $A_l = {:}P_l$] [order by $A_{l+1}\ dir_{l+1}, \ldots, A_a\ dir_a$]
(2)	select $B_1, \ldots, B_b$ from R where $A_1 = {:}P_1$ and $\ldots$ and $A_l = {:}P_l$ and A_{l+1} between ${:}P_{l+1}$ and ${:}P_{l+2}$ [order by $A_{l+1}\ dir_{l+1}, \ldots, A_a\ dir_a$]
(3)	select $B_1, \ldots, B_b$ from R where ID $= {:}P_1$

$\{A_i\}_{i=1}^{l}$ are non-ID attributes and $\{A_i\}_{i=l+1}^{a}$ are non-reference attributes.

Table 3: The three sSQL query types

In the paper we will describe the design of the Index Merger module (see Figure 1). One possible optimization criteria is that the size of the produced extended indices should be as small as possible subject to the constraint that each input sSQL query should have an efficient plan based on one of the produced extended indices. (The supplied statistical information can be used to approximate the size of an extended index.) Alternatively, the problem can be formulated as finding the physical design that fits in the available storage space and that can efficiently supports as many of the input queries as possible. For conciseness, in the paper we skip the details of the algorithms for solving the two optimization problems and concentrate on the index merging procedure.

4. Physical Design Model

We next present the notion of object ordering, followed by the formal syntax and semantics of an extended index.

Definition 4.1 (object ordering). *For a table R, an object ordering is defined using the syntax $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle\rangle$, where $A_1, \ldots, A_a$ are distinct attributes of the table R. It denotes an ordering of the objects in the table R, where the objects are first ordered relative to the value of A_1 in ascending order if $dir_1 =$* asc *and in descending order if $dir_1 =$* desc*, next relative to the value of the attribute A_2 in direction dir_2 and so on.*

Sometimes, when the table on which an ordering is applied is clear from the context, we will skip the table name from the syntax of an object ordering. Also, note that we will use R_1 `op` R_2 to denote $\pi_{\texttt{ID}}(R_1)$ `op` $\pi_{\texttt{ID}}(R_2)$, where `op` $\in \{\subset, \subseteq, \equiv, \cup, \cap\}$ and $\equiv$ is used to denote the set equivalence operator.

Definition 4.2 (syntax of an extended index). *An extended index X is represented by a pair $\langle\{\gamma_1, \ldots, \gamma_m\}, G^t\rangle$. We will refer to $\bar{\gamma} = \{\gamma_1, \ldots, \gamma_m\}$ as the γ-condition of X and write $\gamma(X)$. When the γ-condition is missing, the trivial γ condition that consists of the empty set is assumed. The second argument G^t is a rooted tree with sibling ordering (that is, the children of a parent node are ordered) and node labels that are of the form $\langle R, \langle A_1, \ldots, A_a\rangle\rangle$. We will refer to this tree as the* description tree *of the index. For a label $\langle R, \langle A_1, \ldots, A_a\rangle\rangle$, we will refer to R as the* table of the node *and write* $\texttt{table}(n)$ *and to $\langle A_1, \ldots, A_a\rangle$ as the* ordering label *of the node and write $\mathcal{L}(n)$. We require that the predicates $\{\gamma_i\}_{i=1}^m$ are efficient predicates over the table R that is the table of the root node of the description tree. We impose the following additional restrictions on G^t.*

1. *Let the node n with table R be the parent of the nodes $\langle n_1, \ldots, n_k\rangle$ with tables $\langle R_1, \ldots, R_k\rangle$, respectively. Then the following rules should hold for any instance of the tables:*

   (1) $R_i \subseteq R$ *for* $1 \leq i \leq k$,

   (2) $R_i \cap R_j = \emptyset$ *for* $1 \leq i \neq j \leq k$,

   (3) $\bigcup_{i=1}^{k} R_i = R$, *and*

   (4) $\texttt{attr}(R) \subseteq \texttt{attr}(R_i)$ *for* $i = 1$ *to* k.

2. *If the node n_1 is an ancestor of the node n_2, then the ordering labels of n_1 and n_2 do not share attributes in common.*

For convenience, we introduce several node labeling functions. We will use $\texttt{label}(n)$ to denote the label of a node n. We also define $\mathcal{L}^{\downarrow}$ recursively as follows: for a leaf node n: $\mathcal{L}^{\downarrow}(n) = \texttt{label}(n)$ and for a non-leaf node n with label $\langle L\rangle$ and ordered children $n_1, \ldots, n_k$ we defined $\mathcal{L}^{\downarrow}(n) = \langle L, [\mathcal{L}^{\downarrow}(n_1), \ldots, \mathcal{L}^{\downarrow}(n_k)]\rangle$. Since the string $\mathcal{L}^{\downarrow}(n^r)$, where n^r is the root node of tree, completely describes a tree, we will refer to it as the tree's *string description* and we will sometimes represent a tree by its string description.

We next recursively define the function $\mathcal{L}^{\uparrow}$, which returns the *extended label* of a node. For the root node of the tree n^r, we define $\mathcal{L}^{\uparrow}(n^r) = \texttt{label}(n^r)$. For a non-root node n with label $\langle L \rangle$ and parent node n' with extended label $\langle L' \rangle$ we defined $\mathcal{L}^{\uparrow}(n) = \langle L', L \rangle$. Informally, the extended label of a node is a listing of the labels for the nodes in the path that starts at the root node of the tree and ends at the node.

Consider the extended index from our running example created in Section 1.2. It will have the syntax: $\langle \{\gamma\}, \mathit{Person}, \langle \mathit{name} \rangle, [\langle \mathit{Customer}, \langle \mathit{balance} \rangle \rangle, \langle \mathit{Employee}, \langle \rangle \rangle, \langle \mathit{Trainee}, \langle \mathit{grade} \rangle \rangle] \rangle$, where $\gamma(t)$ is true exactly when t is a *Trainee* object with $\mathit{completionLevel} = 1$. (For now, it should be clear that this extended index satisfies Definition 4.2, where the meaning of this extended index will become clear after we present the semantics of an extended index.)

Definition 4.3 (semantics of an extended index). *The extended index $\langle \{\gamma_1, \ldots, \gamma_m\}, G^t \rangle$ is implemented by a search tree. If $n_1, \ldots, n_k$ are the leaf nodes in G^t and they have tables $\langle R_1, \ldots, R_k \rangle$, respectively, then the search tree contains data pointers to the objects of the tables $\{R_i\}_{i=1}^k$. For each node n of G^t, we next define an ordering function* `Or`*, where the elements in the search tree will be ordered relative to the order $\texttt{Or}(n^r)$ and n^r is the root of G^t.*

If n is a leaf node and $\mathcal{L}(n) = \langle A_1, \ldots, A_a \rangle$, then we define $\texttt{Or}(n) = \langle A_1 \texttt{ asc}, \ldots, A_a \texttt{ asc} \rangle$. If n is a non-leaf node with children $\langle n_1, \ldots, n_k \rangle$, $\texttt{table}(n) = R$, and $\mathcal{L}(n) = \langle A_1, \ldots, A_a \rangle$, then we define $\texttt{Or}(n)$ to be the following ordering. (Nodes that are indistinguishable relative to this order will be ordered relative the attribute `ID` *of the objects they point to.)*

1. *The objects are first ordered relative to the object ordering $\langle A_1 \texttt{ asc}, \ldots, A_a \texttt{ asc} \rangle$.*

2. *Next, if two or more objects have the same value for the attributes $\{A_i\}_{i=1}^a$, then they are ordered relative to the attribute A in ascending order, where $t.A = i$ if and only if $t \in \texttt{table}(n_i)$ for $1 \leq i \leq k$.*

3. *Finally, if two or more objects have the same value for the attributes $\{A_i\}_{i=1}^a$ and for the attribute A, then they are ordered relative to $\texttt{Or}(n_i)$, where i is the common value for the attribute A.*

If the index has a non-trivial γ-condition of the form $\{\gamma_1, \ldots, \gamma_m\}$, then we will associate with each node in the search tree m marker bits. The jth

marker bit of a node is set exactly when the node or one of its descendants in the search tree contains a data pointer to an object for which γ_j holds ($1 \leq j \leq m$).

Figure 3 shows the extended index for our running example. The γ-condition of an extended index contains one predicate for each marker bit that needs to be created. In the example case, the marker bit allows us to efficiently find *Trainee* objects for which *completionLevel* $= 1$. The description tree of an extended index defines the branching order of the index. In our example, people with the same name are ordered relative to their type and further different ordering is defined for customers, employees, and trainees. The tables of the leaf nodes in the description tree define the objects that will be pointed-to by the index. In our example, the extended index will point to the *Customer*, *Employee*, and *Trainee* objects. The tables for the non-leaf nodes are simply defined as the union of the tables of the child nodes. In our example, in order for the extended index to be valid, the table *Person* must be the union of the tables *Customer*, *Employee*, and *Trainee*.

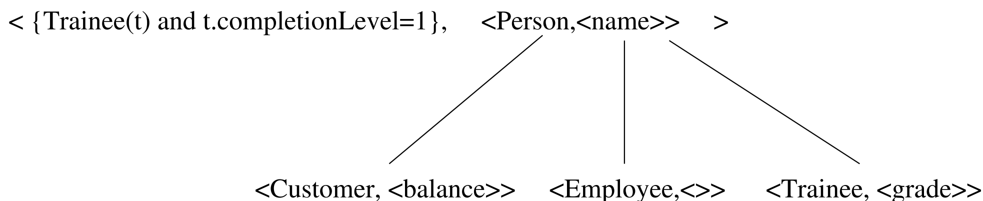

Figure 3: Example extended index

Before defining the interface of an extending index (that is, the methods that it can efficiently supports), we introduce several intermediate definitions and lemmas.

Definition 4.4 (extended γ-condition). *The extended γ-condition of an extended index X is* $\{\texttt{TRUE}\} \cup \{\texttt{FALSE}\} \cup \{\bigcup_{\emptyset \not\equiv \bar{\gamma} \subseteq \gamma(X)} \bigvee_{\gamma \in \bar{\gamma}} \gamma\}$. *We will write $\gamma^e(X)$ to denote it.*

We next present a lemma that explains the meaning of an extended γ-condition.

Lemma 4.5 (meaning of an extended γ-condition). *Let X be an extended index. Then $\gamma \in \gamma^e(X)$ if and only if the value of the marker bit for*

γ in every node of the search tree for X, if such a marker bit hypothetically existed, can be computed as a function of the values of the other marker bits for that node.

Proof: See [14].

As an example, if an extended index contains the γ-conditions *Trainee*(t) and $t.completionLevel = 1$, then the γ-condition *Trainee*$(t) \vee t.completionLevel = 1$ will be in the extended γ-condition of the index. The reason is that if a node in the search tree has one of the marker bits for the first two γ-conditions set, then the marker bit for the third γ-condition will also be set and therefore the third marker bit does not need to be stored explicitly.

Definition 4.6 (rooted path in a tree). *A rooted path in a tree is any path in the tree that starts at the root of the tree. We will denote by* $\texttt{rp}(G^t)$ *the set of all rooted paths in the tree* G^t.

Definition 4.7 (clustering property). *Let* $K = \langle n_1, \ldots, n_k \rangle$ *be a rooted path in the description tree of the extended index* X *and let the attributes in* $\mathcal{L}^{\uparrow}(n_k)$ *be* $\langle A_1, \ldots, A_b \rangle$ *in this order. Given an integer* a, $1 \leq a \leq b$, *we will say that* K is clustered *relative to the integer* a *if and only if at least one of the following conditions holds.*

1. $k = 1$.

2. *If* n_r *is the node in* K *with the biggest subscript for which* $\mathcal{L}^{\uparrow}(n_r)$ *contains exclusively attributes from the set* $\{A_i\}_{i=1}^{a}$ *in its ordering label, then either* $r = k$ *or* $r = k - 1$.

The following lemma explains why the above property is called the clustering property.

Lemma 4.8 (meaning of the clustering property). *Let* X *be an extended index and* $K = \langle n_1, \ldots, n_k \rangle$ *be a rooted path in description tree of the extended index. Let* $\texttt{table}(n_k) = R_k$ *and* $\langle A_1, \ldots, A_b \rangle$ *be the attributes in* $\mathcal{L}^{\uparrow}(n_k)$ *in order of appearance. Then the* R_k *objects that have the same value for the attributes* $\{A_i\}_{i=1}^{a}$ $(a \leq b)$ *are sequential in an in-order traversal of the index (that is, clustered together) if and only if* K *is clustered relative to the integer* a.

Proof: See [14].

In the description tree of Figure 3, consider the left most rooted path that goes through the *Person* and *Customer* tables. This path is clustered relative to the integers 1 and 2 because the *Customers* that have the same value for the attribute *name* are clustered together, as are the *Customers* that have the same value for the attributes *name* and *balance*. Conversely, if the node for the *Customer* table had two children nodes with tables *wealthyCustomers* and *averageCustomers*, then the rooted path that reaches the *wealthyCustomers* node will not be clustered relative to the integer 1 because wealthy customers with the same name are going to be separated in groups relative to their *balance* and therefore not clustered together.

Definition 4.9 (interface of an extended index). *An extended index X with description tree G^t supports the operations that are listed bellow. For the last three operations a common pre-condition is the existence of a node n_r in G^t with table R_r. We will refer to the path in G^t that starts at the root and ends at n_r as $K = \langle n_1, \ldots, n_r \rangle$. We also assume that the attribute in $\mathcal{L}^{\uparrow}(n_r)$ are $\langle A_1, \ldots, A_b \rangle$ in this order and a is an integer between 1 and b. Note that the* `next` *and the two* `search` *methods return* `NULL` *when the object we are searching for does not exist.*

- `insert(reference` p`):`
  - **pre-condition:** *The object with* `ID` *p belong to the table of one of the leaf nodes in G^t.*
  - **action:** *A node that points to this object is inserted in X.*
- `delete(reference` p`):`
  - **pre-condition:** *There exists a node that points to the object with* `ID` *p in X.*
  - **action:** *Deletes this node from X.*
- `reference exact_search(table` R_r, `param` $P_1, \ldots,$ `param` P_a, `direction` $dir_{a+1}, \ldots,$ `direction` dir_b, `efficient predicate` γ`):`
  - **pre-condition:** *Either K is clustered relative to the integer a and $\gamma \in \gamma^e(X)$ or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$* `TRUE` *if and only if $t \in \gamma(R_r)$ for any object t.*

- **return value:** *Let O be the object ordering $\langle A_{a+1}$ $\texttt{dir}_{a+1}, \ldots, A_b$ $dir_b\rangle$. This method returns the* ID *of the first object t in X, relative to the order O, for which the following hold.*
  (a) $t \in R_r$.
  (b) $\gamma(t)$ holds.
  (c) $\bigwedge_{i=1}^{a}(t.A_i = P_i)$.

- `reference closest_search(table` R_r, `param` $P_1, \ldots,$ `param` P_a, `direction` $dir_a, \ldots,$ `direction` dir_b, `efficient predicate` γ`):`
  - **pre-condition:** *Either K is clustered relative to the integer $a-1$ and $\gamma \in \gamma^e(X)$ or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$* TRUE *if and only if $t \in \gamma(R_r)$ for any object t.*
  - **return value:** *Let O be the ordering $\langle A_a$ $\texttt{dir}_a, \ldots, A_b$ $dir_b\rangle$. This method returns the* ID *of the first object t in X, relative to the order O, for which the following hold.*
    (a) $t \in R_r$.
    (b) $\gamma(t)$ holds.
    (c) $\bigwedge_{i=1}^{a-1}(t.A_i = P_i)$.
    (d) $t.A_a > P_a$ when $dir_a =$ `asc` and $t.A_a < P_a$ when $dir_a =$ `desc`.

- `reference next(table` R_r, `param` $P_1, \ldots,$ `param` P_a, `direction` $dir_{a+1}, \ldots,$ `direction` dir_b, `reference` p, `efficient predicate`γ`):`
  - **pre-condition:** *There exists an object $t \in R_r$ with* ID *p pointed to by a node in X. Also, either K is clustered relative to the integer a and $\gamma \in \gamma^e(X)$ or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$* TRUE *if and only if $t \in \gamma(R_r)$ for any object t.*
  - **return value:** *Let O be the ordering $\langle A_{a+1}$ $dir_{a+1}, \ldots, A_b$ $dir_b\rangle$. The method returns the* ID *of the first object $t' \in R_r$ after the object t, relative to the order O, for which $\gamma(t')$ and $\bigwedge_{i=1}^{a} t'.A_i = P_i$ both hold.*

The preconditions in the last three methods guarantee that either the objects in the query result are clustered together in the index or they can be

retrieved efficiently by accessing only the marked nodes. As a consequence, the following theorem holds.

Theorem 4.10 (interface of an extended index). *If the sizes of the objects that are indexed in the extended index X with description tree G^t are limited by some constant, then each method of the interface of X takes $O(log(|X|) \cdot |def(X)|)$ worst-case time.*

Proof: See [14].

We next present several definitions and a theorem that describe the set of sSQL that can be efficiently answered using an extended index under certain assumptions.

(*type*)	(*query*)
(1)	`select` $B_1, \ldots, B_b$ `from` R [`where` $A_1 = :P_1$ and $\ldots$ and $A_l = :P_l$] [`order by` A_{l+1} $dir_{l+1}, \ldots, A_s$ dir_s]
(2)	`select` $B_1, \ldots, B_b$ `from` R `where` $A_1 =: P_1$ `and` $\ldots$ `and` $A_l = :P_l$ `and` A_{l+1} `between` $:P_{l+1}$ `and` $:P_{l+2}$ [`order by` A_{l+1} $dir_{l+1}, \ldots, A_s$ dir_s]
(3)	`select` $B_1, \ldots, B_b$ `from` R `where ID` $= :P_1$

1. $\langle R_1, \ldots, R_k \rangle$ are the tables of $\langle n_1, \ldots, n_k \rangle$, respectively.

2. R is the base table R_k (in which case $\gamma =$ `TRUE`) or a materialized view with query "`select * from` R_k `where` $\gamma(R_k)$".

3. $\langle A_1, \ldots, A_s \rangle$ are the first s attributes in this order from $\mathcal{L}^{\uparrow}(n_k)$.

4. Either K is clustered relative to l and $\gamma \in \gamma^e(X)$ or there exists a $\gamma' \in \gamma^e(X)$ that has the property that for an object t, $\gamma'(t) =$ `TRUE` if and only if $t \in \gamma(R_k)$.

Table 4: Critical queries for a rooted path $K = \langle n_1, \ldots, n_k \rangle$ of X

Definition 4.11 (queries of a rooted path of an extended index). *Let X be an extended index. Table 4 shows the set of sSQL queries that we will associate with a rooted path K of the index. We will denote this set as $Q_K(X)$.*

The above definition describes the set of sSQL queries that can be efficiently answered by the part of an extended index that is associated with a rooted path of its description tree. The third condition of Table 4 requires that the query can refer only the attributes along the path, while the fourth condition guarantees that the query can be answered efficiently.

Definition 4.12 (critical queries supported by an extended index). *Let X be an extended index. We define the set of critical queries efficiently supported by X to be all queries that have the following properties.*

1. *They are sSQL queries that reference a table that has one of the following properties*
   a) *the table appears in the string description of X,*
   b) *the table is a materialized view that has an underlying query of the form "`select * from` R `where` $\gamma(R)$", where $\gamma \in \gamma^e(X)$ and R appears in the string description of X,*
   c) *the table is a* `union all` *of disjoint tables (that is, no overlapping objects with the same* `ID` *value) described in (a) and/or (b).*
2. *They can be efficiently answered using X.*

We will refer to this query set as $Q(X)$.

Note that we define R_1 `union all` R_2 as all the objects in R_1 followed by the objects in R_2, where we require that R_1 and R_2 do not share objects in common and no constraint is specified on the ordering of the result.

We next define the operator `cl`$(\bar{Q})$ that describes the queries that can be created from the queries $\bar{Q}$ by merge sorting the results.

Definition 4.13 ($cl(\bar{Q})$). $cl(\bar{Q}) \equiv \{Q | \exists \{Q_1, \ldots, Q_k\} \subseteq \bar{Q}, Q_i(D) \cap Q_j(D) = \emptyset$ *for any database instance D and $1 \leq i \neq j \leq k$ and $Q \equiv$* `merge_sort`$(Q_1, \ldots, Q_k, O)\}$*, where the method* `merge_sort` *returns the objects "Q_1* `union all` *$\ldots$* `union all` *Q_k" in the order O assuming the results of $\{Q_i\}_{i=1}^k$ are sorted in this order.*

Theorem 4.14 (efficiently supported queries by an extended index). *Let* X *be an extended index. Then* $cl(\bigcup_{K \in \texttt{rp}(X)} Q_K(X)) \subseteq Q(X)$. *Moreover, if* $Q \in Q(X)$*, then there exists a query* Q' *equivalent to* Q *and* $Q' \in cl(\bigcup_{K \in \texttt{rp}(X)} Q_K(X))$.

Proof: See [14].

The above theorem states that the queries that are efficiently supported by an extended index are the queries that are associated with the different rooted paths of the description tree of the index. The `cl` operator is introduced to describe that a query that asks for the union of the results of several queries that can be efficiently answered can also be efficiently answered by the extended index by merging the already sorted results.

5. Index Merging

5.1. Syntax and Semantics of a PART

Our strategy for index merging is to first calculate the set of extended indices that can be efficiently answered by each input sSQL query. In order to do so, we need the definition of a *Parameterized Access Requirement Type* (PART), which describes a set of extended indices.

Definition 5.1 (PART syntax). *A PART* $\mathcal{P}$ *is defined by the pair* $\langle \bar{\gamma}, G^T \rangle$*, where* $\bar{\gamma}$ *is a set of efficient predicates and* G^t *is a description tree with no order defined on node siblings, where every node has a label with the following syntax.*

$$\textit{node label} ::= \langle R, \langle L \rangle \rangle \mid \langle R, \langle \rangle \rangle$$
$$L ::= E \mid F \mid E,\, L \mid F,\, L$$
$$E ::= \{A_1, \ldots, A_a\}$$
$$F ::= A \mid A, F$$

In the above grammar, we require that $a > 0$.

Given a node label $\langle R, \langle L \rangle \rangle$*, we will refer to the* E *parts of* L *as* E-components *and to the* F *parts – as* F-components*. We will refer to* R *as the node's* table *and write* $\texttt{table}(n)$ *to denote it, to* L *as the node's* ordering label *and write* $\mathcal{L}(n)$ *to denote it. Similarly, we will refer to* $\bar{\gamma}$ *as the PART's* γ-condition *and write* $\gamma(\mathcal{P})$ *to denote it. We will use* $|L|$ *to denote the number of attributes referenced in an ordering label.*

The meaning of an E-component is that the order of the attributes is not important. For example, in the query : "`select * from` R `where` $A = :P_1$ `and` $B = :P_2$, the attributes in the resulting index can be ordered relative to A, B or relative to B, A. Conversely, the order of the attributes that define the ordering of the index that efficiently support the query "`select * from` R `order by` A, B" is important.

Informally, a PART represents a set of extended indices that can be created from it by fixing the order of the attributes in the E components, removing the curly brackets in the ordering labels of the nodes, fixing the sibling order in the tree, and adding additional nodes when needed to satisfy condition 1.3 of Definition 4.2. Note that the curly brackets are used to denote that the ordering of the attributes inside them is not fixed. In order to formally define the semantics of a PART, we present the following intermediate definitions.

Definition 5.2 (permutations for a PART ordering label). *Let L be an ordering label of a PART node. We will use $\Pi(L)$ to denote the result of the nondeterministic procedure of permutating the attributes in some of the E-components and then splitting the e-components. We will call Π a complete permutation if it converts L into an expression in which all E-components are of size at most 1. We will use Π^c to denote a complete permutation.*

For example, if "$L = A, B, \{C, D, E\}$", then "$A, B, \{C, E\}, \{D\}$" is one possible value for $\Pi(L)$, while "$A, B, \{E\}, \{C\}, \{D\}$" is an example result of a complete permutation.

Definition 5.3 (permutation for a PART). *A* permutation Π *for a PART $\mathcal{P}$ permutates the E-components of some of the ordering labels of the PART. A* complete permutation Π^c *applies complete permutations to all ordering labels.*

Definition 5.4 (fixed PART). *A fixed PART has no E-components of size greater than one in its ordering labels.*

Definition 5.5 (fixed PART ⇒ extended index). *Let $\mathcal{P}$ be a fixed PART. This PART can be used to create an extended index with the same γ-condition. In order to construct the description tree of the index, apply the following nondeterministic procedure.*

1. *Fix the order of the node siblings.*
2. *Convert all ordering labels to F-components by removing the curly brackets around the E-components.*
3. *If* $\{n_i\}_{i=1}^k$ *are the children of the node* n *and* $\bigcup_{i=1}^{k} \texttt{table}(n_i) \subset \texttt{table}(n)$, *then add a new child node to* n *with table* R *that includes the objects that are in* $\texttt{table}(n)$ *but not in* $\texttt{table}(n_i)$ *for* $i = 1$ *to* k. *Include empty ordering label for the new node.*

Definition 5.6 (valid PART). *A PART* $\mathcal{P}$ *is valid if and only if for every complete permutation* Π^c *of* $\mathcal{P}$, $\Pi^c(\mathcal{P})$ *is converted into a valid extended index by any application of the nondeterministic procedure described in Definition 5.5.*

Definition 5.7 (cover of a set of indices). *Let* $\bar{X}$ *be a set of extended indices. Then* $\texttt{cover}(\bar{X}) = \{X | \exists X' \in \bar{X}, Q(X') \subseteq Q(X)\}$.

Informally, a cover of a set of indices is the set of all indices that can be used to efficiently answer the sSQL queries that the initial extended can efficiently answer.

Definition 5.8 (semantics of a fixed PART). *Let* $\mathcal{P}$ *be a fixed PART. Then we will use* $X(\mathcal{P})$ *to denote the set of extended indices that* $\mathcal{P}$ *can be converted into using the nondeterministic procedure from Definition 5.5. We will also use* $X^c(\mathcal{P})$ *to denote the set* $\texttt{cover}(X(\mathcal{P}))$.

Definition 5.9 (semantics of a PART). *Let* $\mathcal{P}$ *be a PART. Then we will use* $X(\mathcal{P})$ *to denote the set of extended indices* $\bigcup_{\Pi^c} X(\Pi^c(\mathcal{P}))$, *where* Π^c *varies over all valid complete permutations for* $\mathcal{P}$. *We will also use* $X^c(\mathcal{P})$ *to denote the set* $\texttt{cover}(X(\mathcal{P}))$ *and refer to is the set of extended indices represented by the PART.*

Going back to our motivating example from Section 1.2, the queries from Table 2 will generate the PARTs shown in Table 5. Note that all PARTs in Table 5 are fixed PARTs, where $\mathcal{P} = \langle \mathit{Customer}, \langle \{\mathit{name}, \mathit{balance}\} \rangle \rangle$ is an example of a PART that is not fixed and that will be generated by a query that is equivalent to the following query.

(*name*)	(*query*)	(*PART*)
Q_1	`select *` `from` *Person* `order by` *name*	$\mathcal{P}_1 = \langle Person, \langle name \rangle\rangle$
Q_2	`select *` `from` *Customer* `where` $name = {:}P_1$ `order by` *balance*	$\mathcal{P}_2 = \langle Customer, \langle \{name\}, balance \rangle\rangle$

Q_3	`select *`
	`from` *Trainee*
	`where` $completionLevel = 1$
	`order by` *name* `asc`, *grade* `asc`
		$\mathcal{P}_3 = \langle V_T, \langle name, grade \rangle\rangle$

Table 5: Three example critical queries and the corresponding PARTs

select $*$
from $Customer$
where $name = {:}P_1$ and $balance = {:}P_2$

As Definition 5.3 suggests, a complete permutation will convert $\mathcal{P}$ in either the fixed PART $\langle Customer, \langle name, balance\rangle\rangle$ or the fixed PART $\langle Customer, \langle balance, name\rangle\rangle$.

The merging step will merge the PARTs from Table 5 into the PART $\mathcal{P} = \langle\{\gamma\}, Person, \langle name\rangle, [\langle Customer, \langle balance\rangle\rangle, \langle V_T, \langle grade\rangle\rangle]\rangle$, where $\gamma(t)$ holds for a *Person* object if and only if t is also an object in V_T (that is, t is a *Trainee* object for which $completionLevel = 1$).

The final step of our algorithm is converting the created PART into an extended index. The extended index $X = \langle\{\gamma\}, Person, \langle name\rangle, [\langle Customer, \langle balance\rangle\rangle, \langle V_T, \langle grade\rangle\rangle, \langle V_R, \langle\rangle\rangle]\rangle$ is one possible index, where V_R contains the *Person* objects that are not in the tables *Customer* and V_T.

We next formally describe the PART creation and merging steps of our algorithm.

5.2. Step 1 - Converting Extended Indices into PARTs

Table 6 shows the PARTs that will be produced for each type of sSQL query (see Table 3). Note that for a sSQL query of Type 3 we do not need to create a PART because we assume the capability of efficiently retrieving an object from its `ID` (that is, the `ID` can be the address of the object in the

(*query type*)	(*PART*)
(1)	$\langle R, \langle \{A_1, \ldots, A_l\}, \langle A_{l+1}, \ldots, A_a \rangle \rangle \rangle$
(2)	$\langle R, \langle \{A_1, \ldots, A_l\}, \langle A_{l+1}, \ldots, A_a \rangle \rangle \rangle$
(3)	

Table 6: The PARTs for the three sSQL query types

main-memory case or a mapping hash table can be created in the secondary-storage case).

We next present a theorem that states that the created PARTs indeed represent the set of extended indices that can be used to efficiently answer the original queries.

Theorem 5.10 (correctness of PART creation). *If the mapping from Table 6 is applied on the sSQL query Q to generate the PART $\mathcal{P}$, then $X^c(\mathcal{P})$ contains exactly the set of extended indices that can be used to efficiently answer Q.*

Proof: See [14].

5.3. Step 2 – PART Merging

In this section, we present a procedure for merging PARTs (that is, sets of extended indices). In order for a set of PARTs to be mergeable, it must be the case that the intersection of the extended indices that the PARTs represent is not empty, that is, the PART merging will be beneficial.

5.3.1. Merging Ordering Labels

A procedure for merging PARTs requires a way for merging the ordering labels in their description trees. We next present an algorithm that does this and we use $\oplus$ to refer to this operation. Informally, the operation succeeds when one of the ordering labels is a prefix of the other under some permutation. The result of the merge is an ordering label that is a permutation of both input ordering labels. A formal definition of the $\oplus$ operation follows, where $\circ$ is the string concatenation operation and $\lfloor \cdot \rfloor$ is the operation that removes the curly brackets from empty E-components.

$$L_1 \oplus L_2 = \begin{cases} L_2 & \text{if } L_1 = \epsilon; \\ L_1 & \text{if } L_2 = \epsilon; \\ E_1 \circ (L'_1 \oplus (\lfloor E_2 - E_1 \rfloor \circ L'_2)) & \text{if } L_1 = E_1 \circ L'_1,\ L_2 = E_2 \circ L'_2,\ E_1 \subseteq E_2; \\ E_2 \circ ((\lfloor E_1 - E_2 \rfloor \circ L'_1) \oplus L'_2) & \text{if } L_1 = E_1 \circ L'_1,\ L_2 = E_2 \circ L'_2,\ E_2 \subseteq E_1; \\ A \circ ((\lfloor E_1 - \{A\} \rfloor \circ L'_1) \oplus L'_2) & \text{if } L_1 = E_1 \circ L'_1,\ L_2 = A \circ L'_2,\ A \in E_1; \\ A \circ (L'_1 \oplus (\lfloor E_2 - \{A\} \rfloor \circ L'_2)) & \text{if } L_1 = A \circ L'_1,\ L_2 = E_2 \circ L'_2,\ A \in E_2; \\ A \circ (L'_1 \oplus L'_2) & \text{if } L_1 = A \circ L'_1 \text{ and } L_2 = A \circ L'_2; \\ \texttt{UNDEFINED} & \text{otherwise.} \end{cases}$$

Note that in the above pseudo-code we have used E to denote a non-empty set, that is, a string of type $\{\ldots\}$ and A to denote a single attribute. Also, we have used $E_1 - E_2$ to denote the string that corresponds to the set difference of the two sets of attributes. Note as well that the "$\oplus$" function is partial, that is, not every two ordering labels are mergeable. Since "$\oplus$" is commutative and associative, we will use $\bigoplus_{i=1}^{k} L_i$ to denote $L_1 \oplus (L_2 \oplus (\ldots (L_{k-1} \oplus L_k) \ldots))$. We next present an intermediate definition and a theorem that describes the properties of the $\oplus$ operation.

Definition 5.11 (compatible attribute orderings). *Let $\{L_i\}_{i=1}^k$ be k ordering labels that are F-components. We will say that the* attribute orderings *defined by $\{L_i\}_{i=1}^k$ are* compatible *if and only if for all $1 \leq i, j \leq k$ either L_i is a prefix of L_j or L_j is a prefix of L_i or L_i and L_j are the same.*

Theorem 5.12 (correctness of the $\oplus$ operation). *1. Let $\{L_i\}_{i=1}^k$ be k ordering labels. If $\bigoplus_{i=1}^{k} L_i$ returns $L_{k+1} \neq$ `UNDEFINED`, then:*

a) Let $\{\Pi_i^c\}_{i=1}^k$ be complete permutations that convert $\{L_i\}_{i=1}^k$ into the F-components $\{L'_i\}_{i=1}^k$. If the attribute orderings defined by $\{L'_i\}_{i=1}^k$ are compatible, then there exists a complete permutation Π_{k+1}^c that converts L_{k+1} into L'_{k+1} and the attribute orderings defined by $\{L'_i\}_{i=1}^{k+1}$ are compatible.

b) Let Π_{k+1}^c be a complete permutation for L_{k+1} and $L'_{k+1} = \Pi_{k+1}^c(L_{k+1})$. Then there exist complete permutations $\{\Pi_i^c\}_{i=1}^k$ that convert $\{L_i\}_{i=1}^k$ into the F-components $\{L'_i\}_{i=1}^k$, where the attribute orderings defined by $\{L'_i\}_{i=1}^{k+1}$ are compatible.

2. *If* $\bigoplus_{i=1}^{k} L_i$ *returns* `UNDEFINED`*, then there do not exist complete permutations* $\{\Pi_i^c\}_{i=1}^k$ *that convert* $\{L_i\}_{i=1}^k$ *into* $\{L'_i\}_{i=1}^k$*, respectively, such that the attribute orderings defined by* $\{L'_i\}_{i=1}^k$ *are compatible.*

Proof: See [14].

For example, the result of merging the ordering labels "{*name*, *balance*}" and "*balance*" will be "*balance*, *name*". In the first ordering label the order of the attributes is not fixed, while the second ordering label forces *balance* to come first.

5.3.2. Merging PARTs

We define a *simple* PART as follows.

Definition 5.13 (simple PART). *A PART is* simple *if it has the format shown in Table 6.*

From the definition it directly follows that Step 1 of our algorithm produces only simple PARTs. We next formally define when two PARTs are mergeable.

Definition 5.14 (PART merging). *We will say that the PARTs* $\{\mathcal{P}_i\}_{i=1}^k$ *are mergeable into the PART* $\mathcal{P}$ *if and only if the following conditions hold.*

1. $\bigcap_{i=1}^{k} X^c(\mathcal{P}_i) \neq \emptyset$ *and* $X^c(\mathcal{P}) = \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$.

2. *If* $\{R_i\}_{i=1}^k$ *are the tables of the root nodes of the description trees of* $\{\mathcal{P}_i\}_{i=1}^k$*, then for every* i*,* $1 \leq i \leq k$ *there exists* j*,* $1 \leq j \leq k$ *(*$j \neq i$*) such that* $R_i \cap R_j \neq \emptyset$*.*

The first rule in the above definition guarantees that the new PART will represent exactly the indices that are common to all of the original PARTs. As expected, PARTs that do not share indices in common are not mergeable. The second rule guarantees that only PARTs that represent indices with common data will be merged.

We next present two methods: `table_PART_merge` and `gamma_PART_merge` that perform two different kind of PART mergings. Both method merge an arbitrary PART with a simple PART. The first method adds a new node to the first PART, while the second method does not. [14] shows that the two PART merging methods are sound and complete.

The method `table_PART_merge`$(\mathcal{P}_1, \mathcal{P}_2)$ merges an arbitrary PART $\mathcal{P}_1$ with a simple PART $\mathcal{P}_2$ - see Algorithm 1. The method adds a new node with table R to $\mathcal{P}_1$ when possible in order to create the resulting PART and returns `UNDEFINED` otherwise. If the node insertion can be done in such a way so that the label of the inserted node is empty, then the method also returns `UNDEFINED` (this is the case at Line 9 of the pseudo-code). The reason is that the created PART will be equivalent to the PART without the inserted node in which a γ-condition is added to the PART, and therefore the `gamma_PART_merge` method can be applied.

Algorithm 1 `table_PART_merge`(PART $\mathcal{P}_1$, PART $\mathcal{P}_2$)

Require: the path $\langle n_1 = \langle R_1, \langle L_1 \rangle, \dots n_k = \langle R_k, \langle L_k \rangle \rangle$ (denoted as K) is a rooted path in the description tree of $\mathcal{P}_1$,$\bar{\gamma}_1$ is the γ-condition of $\mathcal{P}_1$ $\mathcal{P}_2 = \langle R, \langle L \rangle \rangle$, $R \subset R_k$, and $R' \cap R = \emptyset$ for every table R' that is a table of a child node of n_k in the description tree of $\mathcal{P}_1$

1: $L' \leftarrow L_1 \circ \cdots \circ L_k$
2: **if** $L \oplus L' =$ `UNDEFINED` **then**
3:     **return** `UNDEFINED`
4: **end if**
5: $L'' \leftarrow L \oplus L'$
6: $\gamma' \leftarrow t \in R$
7: compute $\{L'_i\}_{i=1}^k$ and L_{k+1} subject to $L'_1 \circ \cdots \circ L'_k \circ L_{k+1} = L''$ and $|L'_i| = |L_i|$
8: **if** $|L| \leq |L'|$ **then**
9:     **return** `UNDEFINED`
10: **end if**
11: compute L^1 and L^2 subject to $L^1 \circ L^2 = L$ and $|L^1| = |L'|$
12: **if** is_E_component(L^1) **or** $|L^1| = 0$ **then**
13:     $\gamma \leftarrow$ `TRUE`
14: **else**
15:     $\gamma = \gamma'$
16: **end if**
17: **return** substitute$(\mathcal{P}_1, \bar{\gamma}_1 \cup \gamma, n_1 = \langle R_1, \langle L'_1 \rangle \rangle, n_2 = \langle R_2, \langle L'_2 \rangle \rangle, \dots,$ $n_k = \langle R_k, \langle L'_k \rangle, [\langle R, \langle L_{k+1} \rangle \rangle] \rangle)$;

The method `substitute`$(\mathcal{P}, \bar{\gamma}, n_1 = \dots, n_k = \dots)$ returns the result of substituting the γ-condition in $\mathcal{P}$ with $\bar{\gamma}$ and the node n_i in $\mathcal{P}$ with the value that is specified ($i = 1$ to k). In order for this method to be well defined, it

must be the case that for $i = 1$ to k either n_i is a leaf node in the description tree of $\mathcal{P}$ or the tree for $\mathcal{P}_i$ is in the form of a directed path. The method `is_E_component(L)` returns `TRUE` exactly when L is an E-component.

The method `table_PART_merge` adds a new node to $\mathcal{P}_1$ with table R. When the conditions in Line 12 of the pseudo-code is true, then the rooted path in the new PART that ends at the inserted node, which we will denote as K', is clustered relative to the integer l, where l is the position of the last attribute in L that is part of L^1. In this case, a γ-condition does not need to be added to the new PART because the extended indices that it represents will be able to efficiently answer the query that generated $\mathcal{P}_2$ without the γ-condition (see Theorems 4.14 and 5.9). In the other case, we will add the appropriate γ-condition. The labels along the path K' are also modified to reflect the result of applying the "$\oplus$" operation between the ordering labels of K and the ordering label L.

The pseudo-code for the method `gamma_PART_merge` is shown in Algorithms 2 and 3. It merges an arbitrary PART $\mathcal{P}_1$ with a simple PART $\mathcal{P}_2$. The method tries to do so without introducing new nodes in $\mathcal{P}_1$ and without changing the tables of the nodes in $\mathcal{P}_1$. When this cannot be done, the method returns `UNDEFINED`. Note that $\mathcal{P}$ is a PART, $\bar{R}$ is a set of tables (initially the empty set), and γ' is an efficient predicate, where all three variables are global variables for both methods.

The presented pseudo-code covers two cases when the merging of the two PARTs will be successful.

1. The table for $\mathcal{P}_2$, which we refer to as R, is a non-strict subset of a table of a leaf node n' in $\mathcal{P}_1$ and the ordering label of the single node in $\mathcal{P}_2$, which we refer to as L, is compatible with $\mathcal{L}^{\uparrow}(n')$. In the resulting PART the ordering labels along the complete path that ends at n', which we will refer to as K, will be changed to $L \oplus \mathcal{L}^{\uparrow}(n')$. The predicate γ' will be added to the γ-condition of the resulting PART exactly when either the table for n' is different than the table R or the two tables are the same but K is not clustered relative to the size of the single E-component in $\mathcal{P}_2$.

2. R is a non-strict subset of the tables of several leaf nodes of $\mathcal{P}_2$. Analogous to the previous case, the predicate γ' will be added to the γ-condition of the resulting PART when R is a strict subset of the tables $\bar{R}$ or when the clustering property is not satisfied for one of the rele-

Algorithm 2 `gamma_PART_merge`(PART $\mathcal{P}_1$, PART $\mathcal{P}_2$)

Require: $n_1 = \langle R_1, \langle L_1 \rangle \rangle$ is the root node of the description tree of $\mathcal{P}_1$, $\bar{\gamma}_1$ is the γ-condition of $\mathcal{P}_1$, and $\mathcal{P}_2$ is the PART $\langle R, \langle L \rangle \rangle$, where $L = \{A_1, \dots, A_l\}, A_{l+1}, \dots, A_a$

1: *mergeable* $\leftarrow$ **true**
2: $\gamma' \leftarrow (t \in R)$
3: **if** $R \not\subseteq R_1$ **then**
4:   **return** UNDEFINED
5: **end if**
6: **if** $L \oplus L_1 =$ `UNDEFINED` **then**
7:   **return** UNDEFINED
8: **end if**
9: $L' \leftarrow L \oplus L_1$
10: **if** ((`is_leaf`(n_1)) **or** ($|L| \leq |L_1|$)) **then**
11:   **if** $R = R_1$ **then**
12:     $\gamma \leftarrow$**true**
13:   **else**
14:     $\gamma \leftarrow \gamma'$
15:   **end if**
16:   **return** `substitute`($\mathcal{P}_1, \bar{\gamma}_1 \cup \{\gamma\}, n_1 = \langle R_1, \langle L' \rangle \rangle$)
17: **end if**
18: compute L^1 and L^2 subject to $L^1 \circ L^2 = L'$ and $|L^1| = |L_1|$
19: $\mathcal{P} \leftarrow$ `substitute`($\mathcal{P}_1, \bar{\gamma}_1, n_1 = \langle R_1, \langle L^1 \rangle \rangle$)
20: **for** $n' \in$ `children`($n_1, \mathcal{P}_1$) **do**
21:   *mergable* $\leftarrow$ *mergable* $\wedge$ `recursive_PART_merge`(n', R, L^2, l)
22: **end for**
23: **if** *mergable* **and** $R \subseteq \bigcup_{R' \in \bar{R}} R'$ **then**
24:   **if** $R \subset \bigcup_{R' \in \bar{R}} R'$ **then**
25:     $\mathcal{P} \leftarrow$ `substitute`($\mathcal{P}_1, \bar{\gamma}_1 \cup \{\gamma'\}, n_1 = \langle$`table`$(n_1), \mathcal{L}(n_1) \rangle$)
26:   **end if**
27:   **return** $\mathcal{P}$
28: **end if**
29: **return** `UNDEFINED`

vant paths. The ordering labels in $\mathcal{P}_2$ will be recalculated in analogous fashion to the first case.

Algorithm 3 `recursive_PART_merge`(node n, table R, label L, int l)

Require: the node n has the syntax $\langle R_1, \langle L_1 \rangle\rangle$
1: **if** $R \cap R_1 = \emptyset$ **then**
2: **return** **true**
3: **end if**
4: $mergable \leftarrow$**true**
5: **if** $L \oplus L_1 =$ `UNDEFINED` **then**
6: **return** **false**
7: **end if**
8: $L' \leftarrow L \oplus L_1$
9: **if** `is_leaf`(n) **or** $|L| \leq |L_1|$ **then**
10: $\mathcal{P} \leftarrow$ `substitute`$(\mathcal{P}, n = \langle R_1, \langle L' \rangle\rangle)$
11: $\bar{R} \leftarrow \bar{R} \cup R_1$
12: $K \leftarrow$ the rooted path in $\mathcal{P}$ that ends at n
13: $n_1 \leftarrow$ the root node of $\mathcal{P}$
14: **if** K is not clustered relative to l **then**
15: $\mathcal{P} \leftarrow$ `substitute`$(\mathcal{P}, \gamma(n_1) \cup \{\gamma'\}, n_1 = \langle$`table`$(n_1), \mathcal{L}(n_1)\rangle)$;
16: **end if**
17: **else**
18: compute L^1 and L^2 subject to $L' = L^1 \circ L^2$ and $|L^1| = |L_1|$
19: $\mathcal{P} \leftarrow$ `substitute`$(\mathcal{P}_1, n = \langle R_1, \langle L^1 \rangle\rangle)$
20: **for** $n' \in$ `children`(n) **do**
21: $mergable = mergable \wedge$ `recursive_PART_merge`(n', R, L^2, l)
22: **end for**
23: **end if**
24: **return** $mergable$

The following theorem holds.

Theorem 5.15 (correctness of the PART merging algorithm). *Suppose that the sSQL queries $\{Q_i\}_{i=1}^k$ generate the simple PARTs $\{\mathcal{P}\}_{i=1}^k$. Let $\mathcal{P}$ be the PART produced by applying a combination of the two merge methods. Then $X^c(\mathcal{P}) \equiv \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$.*

Proof: See [14].

6. Experimental Evaluation

6.1. Main Memory

We conducted experimental results using the example database schema shown in Figure 2. The code was written in Java and executed on a Sony VAIO VGN-NW150J laptop running the Windows OS.

First, the database was populated with 130650 customer, 3146 managers, 6500 workers, and 10400 trainees. The data was uniform. For example, there were 676 distinct customer names and 201 different balances (from -100 to +100) for each customer. We next created a workload that consisted of queries and updates, where the update ratio (that is, the ratio of the number of single object updates to the number of single object updates plus the number of retrieved objects in retrieval queries) was set. We examined two cases: the *naive* approach, where an index was created for each query, and the *merge* approach, where indices were merged into extended indices. In the naive approach, the created index was an AA tree. The results of the experiment are shown in Figure 4. The horizontal axis is the update ratio, while the vertical access is the time in milliseconds. The workload consisted of 2000 operations, where the type of operation to execute was chosen randomly based on the update ratio. For example, when the update ratio was 0.1, 181 updates and 1819 retrieve operations were performed on average. All the input queries were sSQL queries. The workload was ran 1000 times for each distinct update ratio and the average over all runs was recorded. As the update ratio increases, update time will increase and query time will decrease. Therefore, we included only the total performance time, which shows how the update ratio influences the performance.

As the figure shows, index merging is beneficial for workloads with high update ratios. The reason is that less duplicate information needs to be refreshed after every update. However, index merging is only slightly beneficial for workloads with low update ratios. The reason is that index merging contributes to the creation of bigger indices and searching in them can be slower. At the same time, index merging can still be beneficial for workloads with low update ratios when storage is scarce. For example, as Figure 5 shows, index merging reduced storage by a factor of two. In the figure, the vertical line represents the total size of the extended indices in the two cases in MBs. The saving in space is the result of avoiding unnecessary data replication.

We also ran experiments without creating any indices for workloads with high update ratios. In particular, we stored the tables as linked lists. As

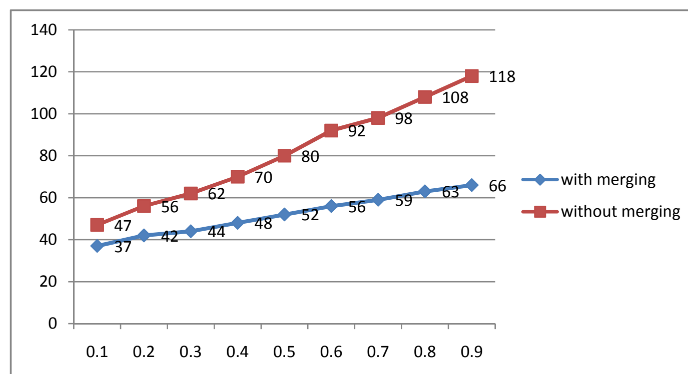

Figure 4: Execution time in milliseconds for different update ratios

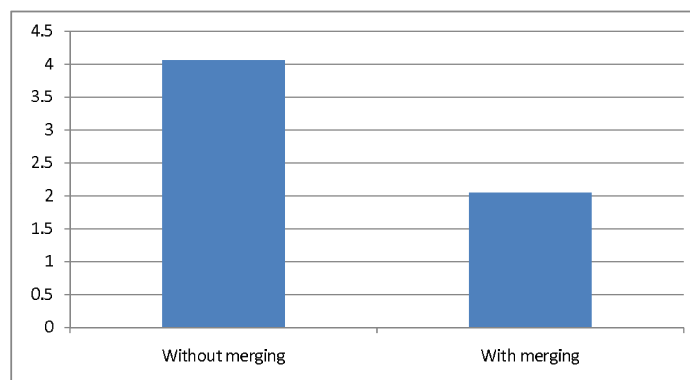

Figure 5: Comparison in MBs of the total sizes of the two sets of extended indices

expected, the transactions ran much slower. We executed the same workload 1000 times and the average run time was 12.6 seconds for update ratio of 0.9. This is about 100 times slower than using an index. This shows that, even for workloads with high update ratios, creating an index is beneficial because an index scan significantly outperforms sequential search.

6.2. Secondary Storage

Figure 6 shows out experimental results with secondary storage. In the experiments, the indices and materialized views that were suggested by IBM DB2 and Microsoft SQL Server for the TPC-C workload benchmark ([15]) with ten warehouses were compared to performing the queries over merged extended indices. The merged indices were manually calculated. Specifically, the sSQL queries and materialized views that are input to our algorithm were

manually created based on the SQL queries in the workload. That is, we manually broke TPC-C queries that are not sSQL queries into sSQL queries. Fortunately 92% of the TPC-C workload can be directly expressed using sSQL queries. The index merging algorithm was also applied manually (implementing it and evaluating its performance is a topic for future research). We applied the approximate algorithm from [14] to selected the PARTs to be merged. The algorithm runs in quadratic time relative to the number of input queries. For the TPC-C workload, the approximate algorithm found the indices of the smallest size to create.

The left side of Figure 6 shows the size of the different auxiliary data structures in MBs, while the right side of the figure shows performance evaluation of running the TPC-C workload for two hours under four different scenarios. In particular, both the Microsoft SQL Server and DB2 were run with the physical design suggested by their own physical design advisors and with the merged extended index that were created by applying the paper's algorithm. All four experiments used the indices on key attributes, which are automatically created by both commercial DBMS systems. Since IBM DB2 and Microsoft SQL Server cannot directly use extended indices, the displayed experimental results were approximated. For example, the overhead of storing and processing marking bits was calculated by adding extra bit attributes for the marking bits. Similarly, since both commercial systems cannot process indices with branching order, the performance was evaluated on equivalent indices of the same size without the branching order. Lastly, the polymorphic property of an extended index was evaluated by evaluating the performance of indices on the different object types. The improvement in performance is due to the fact that TPC-C is an update intensive OLTP workload. Therefore, when redundant data is eliminated through index merging, less work needs to be done to refresh the indices after every update.

7. Conclusion

We presented a novel data structure called extended index. It can used to save space and speed up updates for workloads with significant update ratios. This is done by identifying certain redundant data among indices and eliminating the redundancy. We showed theoretically and validated experimentally the benefits of our approach. Areas for future research include implementing our index merging algorithm and evaluating its performance.

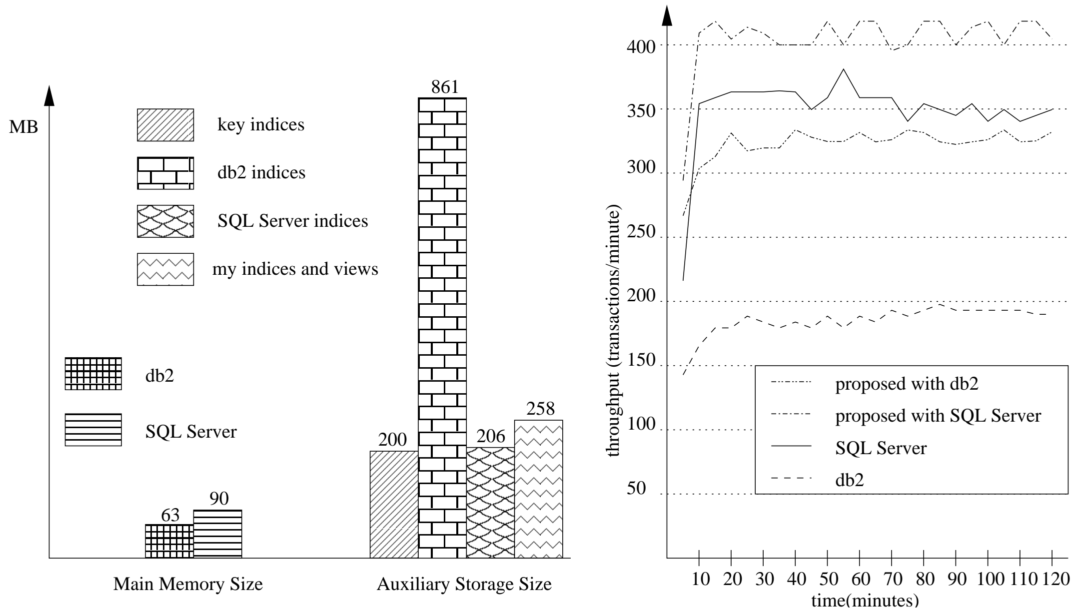

Figure 6: Throughput with different indices

8. Summary of Notation

$cl(\cdot)$ merge sorting function - see Definition 4.13

$cover(\cdot)$ finds the cover of a set of indices - see Definition 5.7

dir constant in the set $\{\texttt{asc}, \texttt{desc}\}$

E an expression of the form $\{A_1, \ldots, A_a\}$, where $a \geq 1$

F an expression of the form $A_1, \ldots, A_a$, where $a \geq 1$

G^t tree

K path in a tree

L node's label or a substring of a node's label

P query parameter

$Q(\cdot)$ set of queries represented by the enclosed component

Q_P query plan

R relation

V materialized view

X extended index

$\mathcal{P}$ PART

$\mathcal{L}(n)$ ordering label of the node n

$\mathcal{L}^{\uparrow}(\cdot)$ node labeling function – see page 13

$\mathcal{L}^{\downarrow}(\cdot)$ node labeling function – see page 13

γ efficient predicate

$\gamma(\cdot)$ result of applying the predicate γ over the enclosed component; result is `TRUE` of `FALSE`

$\Pi(\cdot)$ permutation, where the actual semantics depends on the type of the enclosed component (see Definitions 5.2 and 5.3)

$|\cdot|$ size of the enclosed component

$|def(\cdot)|$ size of the definition of the enclosed component

`ID` an object identifier

References

[1] Adelson-Velskii, G. M. and Landis, E. M. 1962. An Algorithm for the Organization of Information. *Soviet Math. Doklady 3*, 1259–1263.

[2] Agrawal, S., Chaudhuri, S., and Narasayya, V. 2000. Automated Selection of Materialized Views and Indexes for SQL Databases. *VLDB*, 496–505.

[3] Andersson, A. 1993. Balanced search trees made simple. *Workshop on Algorithms and Data Structures*, 60–71.

[4] Bayer and McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica 1,* 3.

[5] Bruno, N. and Chaudhuri, S. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. *SIGMOD 2005*, 227–238.

[6] Bruno, N. and Chaudhuri, S. 2007. An online approach to physical design tuning. *ICDE 2007*, 826–835.

[7] Bruno, N. and Chaudhuri, S. 2010. Constrained Physical Design Tuning. *The VLDB Journal 19,* 1, 21–44.

[8] Calderero, F. and Marques, F. 2010. Region Merging Techniques Using Information Theory Statistical Measures. *Transactions on Image Processing 19,* 6, 1567–1586.

[9] Chaudhuri and Narasayya. 1999. Index Merging. *ICDE*, 296–303.

[10] Dageville, B. 2004. Automatic SQL Tuning in Oracle 10g. *VLDB*, 826–835.

[11] Dittrich, J.-P., Fisher, P. M., and Kossmann, D. 2005. AGILE: Adaptive indexing for context-aware information filters. *ACM SIGMOD*, 215–226.

[12] Finkelstein, S., Schkolnick, M., and Tiberio, P. 1988. Physical Database Design for Relational Databases. *ACM Transaction on Database Systems 13,* 1 (March), 91–128.

[13] Idreos, S., Kersten, M., and Manegold, S. 2007. Updating a Cracked Database. *ACM SIGMOD*, 413–424.

[14] Stanchev, L. and Weddell, G. 2009. Saving Space and Time Using Index Merging for Main-Memory Databases. *IPFW Computer Science Department Technical Report 2008-2,http://www.cs.ipfw.edu/reports/2008/report2.pdf*.

[15] Transaction Processing Performance Council. *TPC-C OLTP*. Transaction Processing Performance Council, http://www.tpc.org.

[16] Valentin, G., Zulian, M., Zilio, D. C., Lohman, G., and Skelley, A. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes. *ICDE*, 101–110.