Saving Space and Time Using Index Merging for Main-Memory Databases

Managing information stored in computers is an integral part of the society we live in. Efficient access to such data is usually supported through the use of index structures. Although indices can significantly reduce the cost of answering queries, they have two major drawbacks. First, they take additional space, which can more than double the size of the database. Second, their maintenance can be a bottleneck. The reason being that every time a table is updated, all indices defined over the table need to be updated accordingly.

We address these two challenges by using a mechanism that reduces the need for storing redundant data among indices. The experiments that we have conducted show that our approach can reduce by half the amount of space taken by the search structures and can improve performance for workloads with high update ratios. These results are achieved by merging indices that contain redundant data into *extended indices*. The later artifact extends a search tree index in two ways: (1) the ordering condition on the index does not need to be linear (e.g., different ordering can be specified on different disjoint subsets of the indexed objects), and (2) efficient (i.e, logarithmic time) search in predefined subsets of the indexed elements is supported. Our current implementation manages extended indices stored exclusively in main-memory, where evaluating the impact of storing extended indices on secondary storage is an area for future research.

Categories and Subject Descriptors: E2 [**Data**]: Data Storage Representations

1. INTRODUCTION

Keeping a database system tuned and running is a non-trivial task that usually requires the full-time involvement of several database administrators (DBAs). In order to mitigate the problem, most commercial database systems include an auto-tuning module that is designed to simplify the job of the DBAs by suggesting values for the different parameters of the system. In particular, commercial products like IBM DB2 ([Valentin et al. 2000]) and Microsoft SQL Server ([Agrawal et al. 2000]) have an advisor that suggests what indices and materialized views should be created for a particular workload. In this paper we show how an output similar to that produced by a physical design advisor can be compressed in a way that preserves the set of queries that can be answered efficiently. (Throughout the paper the term *efficiently* is used to denote order logarithmic worst-case time for returning each element of the query result.)

Consider a workload of ten queries defined over the same table. An index advisor may 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, then less space will be needed and updates will be faster because fewer copies of the same data will have to be updated.

Index merging is a challenging task and most commercial database management systems (DBMSs) provide only limited support (e.g., [Bruno and Chaudhuri 2005]).

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 executed 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 consider the indices X_2 and X_3 to be redundant because they contains a subset of the data that is already stored in X_1. However, removing the two indices will prevent us from efficiently answering queries such as "`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 will show how the three indices can be merged into a single *extended index*[2] on the elements of R_1.

Different index structures have been known for more than forty years. For example, AVL trees were first introduced in 1962 (see [Adelson-Velskii and Landis 1962]), while B+ trees were introduced in 1972 (see [Bayer and McCreight 1972]). However, few studies have considered the possibility of merging indices in order to eliminate redundant data. Two of the few exceptions are [Chaudhuri and Narasayya 1999] and [Bruno and Chaudhuri 2005], which consider merging indices whenever they have attributes in common. However, the papers' approach does not preserve the worst-case logarithmic time-bound on the input queries. To put it differently, a query that has a worst-case logarithmic time-bound against one of the indices to be merged can have linear worst-case time-bound against the merged index. In this 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. Moreover, our experimental evaluation shows that our approach can work well in practice, resulting in a significant reduction of storage cost and improved performance.

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 II 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 trees 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 for example [Finkelstein et al. 1988]). Alternatively, if the *what-if* query optimizer model presented in [Agrawal et al. 2000] and [Valentin et al. 2000] 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.

Note that the input to our system includes sSQL queries rather than indices because sSQL queries carry more detailed information. For example, the index $\langle R, \langle A \ \texttt{asc}, B \ \texttt{asc} \rangle\rangle$ does not tell us whether the order of the attributes can be

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

swapped without sacrificing the logarithmic time bound for object retrieval. In particular, the order of the attributes is not important if the query that generated the index had the form "`select * from` R `where` $A =$ `:`P_1 `and` $B =$ `:`P_2" (we will use P to denote a parameter), but it is important if it had the form "`select * from` R `order by` A `asc`, B `asc`".

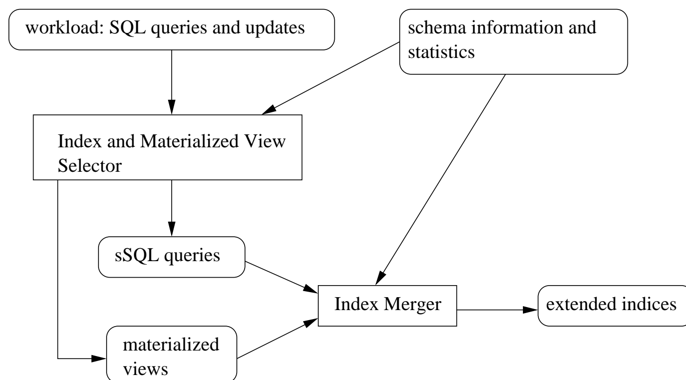

Fig. 1. Relevant DBMS Architecture

The core of this 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. In particular, we will show that the problem of selecting the set of extended indices of the smallest size that can efficiently answer all the input sSQL queries is NP-Complete. Accordingly, we will present an exact exponential-time algorithm and an approximate polynomial-time algorithm for solving the problem.

Both algorithms start by creating a *Parameterized Access Requirement Type* (PART) for each input sSQL query, where a PART represents a potentially infinite set of extended indices. In particular, for an input sSQL query Q, we will create the PART $\mathcal{P}$ that corresponds to exactly the set of indices that can efficiently answer Q. We next merge PARTs that represent indices that have a non-empty intersection. The merging is done in a way that guarantees that the resulting PART represents exactly the intersection of the set of indices represented by the initial PARTs. To paraphrase, a merged PART represents exactly the indices that can efficiently answer the sSQL queries that generated the PARTs that were merged. For example, in Figure 2 the queries $\{Q_i\}_{i=1}^3$ generated the PARTs $\{\mathcal{P}_i\}_{i=1}^3$ and the PART $\mathcal{P}$, which was computed my merging $\{\mathcal{P}_i\}_{i=1}^3$, represents exactly the set of indices that can be used to efficiently answer the three queries.

The exact PART merging algorithm considers all possible PART mergings, while the approximate algorithm applies a greedy heuristic. The final step in both algorithms is to create an extended index for each of the resulting PARTs, where both algorithms select the extended indices that are anticipated to be of the smallest size

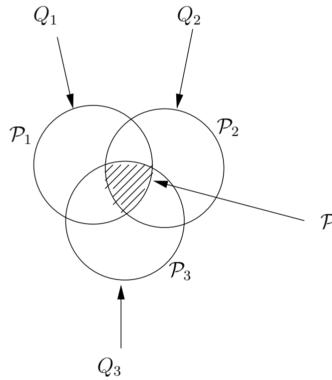

Fig. 2. PART merging example

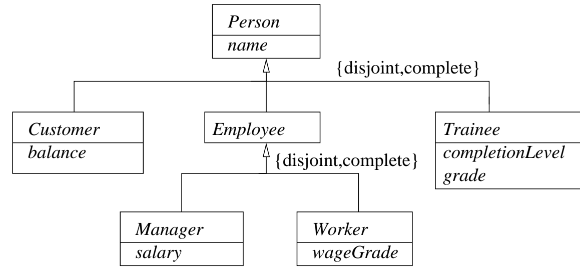

Fig. 3. Example database schema

(*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 I. Three example queries

given the available statistics. In this paper will show that our exact algorithm is not only correct, but also complete in the sense that it produces a set of extended indices of minimal size under certain assumptions.

1.2 Motivating Example

We now demonstrate our approach on the UML class diagram shown in Figure 3. 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. Suppose that

the workload consists of the example queries shown in Table I. 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. Such an index structure is encoded in our implementation using an Arne Andersson search tree (see [Andersson 1993]), where the non-system data at each node consists of a pointer to a *Person* object and every *Person* is pointed to exactly once. (We chose AA search trees because they are easy to implement, but any other balanced search tree will also do the job.) Since the class *Person* contains objects of four different types (i.e., *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 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 of 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. Similarly, 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 tree index. The bit in a node will be set exactly when the node or one of its descendent nodes in the search tree contains a pointer to a *Trainee* object for which $completionLevel = 1$.

The extended index that was created can be used to efficiently answer query Q_1 from Table I by visiting the nodes of the index in order. This will result in producing the correct query answer because the nodes of the index are ordered relative to the attribute *name*.

Query Q_2 from Table I 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 in the search tree the nodes for people with the same name are ordered relative to their type. The query result consists of the objects pointed to by the 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 traversing the marked nodes of the search tree in order. 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 Contributions

The most significant contributions of the paper are listed below.

(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 3.
(2) We show that the problem of merging indices into extended indices in a way that minimizes storage is NP-Complete under certain assumptions and show an exponential time exact algorithm and polynomial time approximate algorithm for solving the problem - see Section 4.
(3) We present experimental results that show how our approach to index merging can decrease storage overhead and speedup updates - see Section 5.

2. DEFINITIONS

We start this section by formally describing the database schema and query language that we will be using. We then use these definitions to formally define the problem at hand.

2.1 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 $\texttt{attr}(R)$ to refer to the attributes of the table R.

We require that every table has the system attributes `ID`. Although this attribute is similar to the global object identifier from the ODMG model (see [Bancilhon and Ferran 1994]), there are two significant differences. First, the `ID` attribute of an object coincides with the physical address where the object is stored in memory and, second, two or more objects can have the same `ID` value when they are stored on top of each other. The second difference applies only when at least one of the objects belongs to a materialized view. That is, two distinct objects that belong to base tables cannot have the same `ID` value because they are never stored at the same location.

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. We require that all reference attributes are not `NULL` and point 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 bellow.

(*type*)	(*query*)
(1)	select $B_1, \ldots, B_b$ from R [where $A_1 = :P_1$ and ... 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 ... 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 II. The three sSQL query types

Definition 2.1 (*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 $t.salary > 200\ 000$) is an example of an efficient predicate and 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 > 200 000
```

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) without the need to be stored in a separate data structure.

2.2 The Query Language

We assume that the the input queries to the Index Merger module (see Figure 1) are sSQL queries – see Table II. 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 one from defining an ordering on reference attributes. This is required 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.

2.3 The Problem

We next define the characteristics of an efficient query plan.

Definition 2.2 (*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 referenced in Q.

In this paper we will describe the design of the Index Merger module (see Figure 1). The optimization constraint is that each sSQL query should have an efficient plan based on one of the produced extended indices and the optimization criteria is that the size of the produced extended indices should be as small as possible, where the supplied statistical information is used to approximate the size of an extended index.

3. PHYSICAL DESIGN MODEL

We will use an Arne Andersson search tree (see [Andersson 1993]) to implement a search tree. Although an AVL tree ([Adelson-Velskii and Landis 1962]) is also a possibility, we have chosen AA trees because the balancing is more easy to implement (see [Andersson 1993]). We next present the notion of object ordering, followed by the formal syntax and semantics of an extended index.

Definition 3.1 (*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 $dir_1 = \texttt{asc}$ and in descending order if $dir_1 = \texttt{desc}$, 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 $R_1 \subset R_2$, $R_1 \cup R_2 = \emptyset$, and so on, to denote $\pi_{\texttt{ID}} R_1 \subseteq R_2$, $\pi_{\texttt{ID}} R_1 \cup \pi_{\texttt{ID}} R_2 = \emptyset$. The reason is that we will consider two tables to be equivalent (or a subset of each other) when they have the same set (or subset) of object `IDs` even when their attributes differ.

Definition 3.2 (*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 $\bar{\gamma} = \langle \{\gamma_1, \ldots, \gamma_m\}$ 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 (i.e., the children of a parent node are ordered) with node labels of the form $\langle R, \langle A_1, \ldots, A_a \rangle\rangle$, where $\{A_i\}_{i=1}^{a} \subseteq \texttt{attr}(R)$. We will refer to this tree as the *description tree* of the index. We will refer to R as the *table of the node* and write $\texttt{table}(n)$ to $\langle A_1, \ldots, A_a \rangle$ as the *ordering label* of the node and write $\mathcal{L}(n)$. The predicates $\{\gamma_i\}_{i=1}^{m}$ should be efficient predicates over the table R that is the root node of the 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 $i \in [1 \ldots k]$.

(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, complete 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 $\mathcal{L}^{\uparrow}$, that 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 for the labels of 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: $\langle \{\gamma\}, \mathit{Peron}, \langle \mathit{name} \rangle, [\langle \mathit{Customer}, \langle \mathit{balance} \rangle \rangle, \langle \mathit{Employee}, \langle\rangle \rangle, \langle \mathit{Trainee}, \langle \mathit{grade} \rangle \rangle] \rangle$, 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 3.2, where the meaning of this extended index will become clear after we present the semantics of an extended index.)

Definition 3.3 (*semantics of an extended index*). Consider the extended index $X = \langle \{\gamma_1, \ldots, \gamma_m\}, G^t \rangle$. It will represent an AA 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 $\texttt{Or}$, where the elements in the AA search tree will be ordered relative to the order $\texttt{Or}(n^r)$, where 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 ore 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 $t.A = i$ iff $t \in \texttt{table}(n_i)$ for $1 \leq i \leq k$.
(3) Finally, if two ore 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.

We next define the placement of the marker bits in the index. 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 of X 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 ($1 \leq j \leq m$).

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

Definition 3.4 (*extended γ-condition*). Given an extended index $X = \langle \bar{\gamma}, G^t \rangle$, we define it's extended γ-condition as $\{\texttt{TRUE}\} \cup \{\texttt{FALSE}\} \cup \{ \bigcup_{\emptyset \not\equiv \bar{\gamma} \subseteq \gamma(X)} \bigvee_{\gamma \in \bar{\gamma}} \gamma\}$ and write $\gamma^e(X)$ to denote it.

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

Lemma 3.5 (*meaning of an extended γ-condition*). Let X be an extended index. Then $\gamma \in \gamma^e(X)$ iff the value of the marker bit for γ in every node of the search tree for X, assuming such a marker bit existed, can be computed as a function of the values of the other marker bits for that node.

Proof: $\Rightarrow$ Assume that $\gamma \in \gamma^e(X)$. If γ is TRUE or FALSE, then this direction is trivially true. Therefore, suppose that $\gamma \equiv \gamma_1 \vee \ldots \vee \gamma_k$ and $\{\gamma_i\}_{i=1}^k$ correspond to marker bits in the index. Then the marker bit for γ of a node n in the search tree will be set exactly when at least one of the marker bits of $\{\gamma_i\}_{i=1}^k$ for n is set. The reason is that if one of $\{\gamma_i\}_{i=1}^k$ holds for an object pointed to by n or one of n's descendants, then γ will hold for this object. Alternatively, if all of $\{\gamma_i\}_{i=1}^k$ are false for n, then neither n nor one of its descendants can point to an object for which $\gamma_1 \vee \ldots \vee \gamma_k$ is true and therefore the marker bit for γ should not be set for the node n.

$\Leftarrow$ The reverse is also true, that is, if $\gamma \notin \gamma^e(X)$, then the value for the marker bit of γ in a node in the search tree can not be inferred from the values for the other marker bits of that node. Assume the opposite, that is, the value for the marker bit of γ for all nodes in the search tree can be inferred from the values of the other marker bits for that node. The expression for γ cannot be a negation of a predicate on which a marker bit is defined in X. The reason is that the fact that γ_1 is TRUE for an object pointed to by a node n or one of its descendants does not tell us whether $\neg\gamma_1$ is TRUE for an object pointed to by n or one of its descendants. Similarly, the expression for γ cannot be a conjunction of γ_1 and γ_2, where marker bits are defined for the two predicates. The reason is that the fact that the marker bits γ_1 and γ_2 are set for a node n implies that γ_1 is TRUE for the object pointed to by n or for an object pointed to by one of n's descendants and γ_2 is also true for an object with the same characteristics. However, it might be the case that γ_1 and γ_2 are true for objects pointed to by different descendants of n, which implies that the value for the marker bit for $\gamma_1 \wedge \gamma_2$ of n cannot be inferred from the value of the marker bits for γ_1 and γ_2 in n. Similarly, the marker bit for the negation of an expression and for the conjunction of expressions cannot be calculated from the maker bits for that expressions. Therefore γ must be equal to TRUE, FALSE, or a disjunction of predicates for which maker bits are defined, that is $\gamma \in \gamma^e(X)$ – contradiction. ■

Definition 3.6 (*rooted path in a tree*). A rooted path in a tree is any path in the tree that starts from the root of the tree. We will denote by $\mathtt{rp}(G^t)$ the set of all rooted paths in the tree G^t.

Definition 3.7 (*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, $0 \leq a \leq b$, we will say that K *is clustered* relative to the integer a iff 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 3.8 (*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 $\mathtt{table}(n_k) = R_k$ and $\langle A_1, \ldots, A_b \rangle$ be the attributes in $\mathcal{L}^{\uparrow}(n_k)$ in order of appearance. Then the objects in R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ $(a \leq b)$ are clustered together relative to the order defined by X (i.e., if relative to this order an object is between two objects in R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$, then the object must also belong to R_k and have these values for the attributes $\{A_i\}_{i=1}^{a}$) iff K is clustered relative to a.

Proof: $\Leftarrow$ First, assume that K is clustered relative to a. This implies that one of the two condition from Definition 3.7 must hold. If the first condition holds, that is $k = 1$, then the the index ordering is $\langle A_1\ \mathtt{asc}, \ldots, A_b\ \mathtt{asc} \rangle$ and all nodes in the index will point to objects of R_k. Next, note that $a \leq b$ implies that the objects will also be ordered relative to the order $\langle A_1\ \mathtt{asc}, \ldots, A_a\ \mathtt{asc} \rangle$ and therefore the nodes in the search that point to objects that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ will be clustered together relative to the search tree order.

Next, consider the case when Condition 2 from Definition 3.7 holds. If $r = k$, then $\mathcal{L}^{\uparrow}(n_k)$ contains exclusively attributes from the set $\{A_i\}_{i=1}^{a}$ in its ordering label and therefore $a = b$. Let c be the largest index of an attribute in $\mathcal{L}^{\uparrow}(n_{k-1})$. By the definition of the semantics of an extended index (Definition 3.3), it follows that the nodes in the search tree that point to object of R_k that have the same value for the attributes $\{A_i\}_{i=1}^{c}$ will be indexed together. Because of the presence of the node n_k, these objects will be ordered relative to the order $\langle A_{c+1}\ \mathtt{asc}, \ldots, A_a\ \mathtt{asc} \rangle$ and therefore the nodes that point to objects of R_k that have the same values for the attributes $\{A_i\}_{i=1}^{a}$ will be sequential relative to the search tree order.

Finally, consider the case when $r = k - 1$. Definition 3.7 implies that $\mathcal{L}^{\uparrow}(n_{k-1})$ contains exactly the attributes $\{A_i\}_{i=1}^{a}$. Again, by the definition of the semantics of an extended index (Definition 3.3), it follows that the nodes in the search tree that point to object of R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ will be sequential in the search tree order.

$\Rightarrow$ Now, assume that K is not clustered relative to a. This implies that $k > 1$ and that the ordering label of n_{k-1} must contain attributes that are outside the

set $\{A_i\}_{i=1}^a$. This implies that it may be the case that an object in the table for the node n_{k-1} that does not belong to R_k is between two objects of R_k (relative to the search tree order) that have the same value for the attributes $\{A_i\}_{i=1}^a$. This implies that the objects in R_k that have the same value for the attributes $\{A_i\}_{i=1}^a$ are not clustered together relative to the search tree order. ■

Definition 3.9 (*interface of an extended index*). An extended index X with description tree G^t supports the following operations.

(1) `insert(reference` p`):`
   pre-conditions: An object with `ID` p is in the table of one of the leaf nodes in G^t.
   action: A pointer to the object is inserted in X.

(2) `delete(reference` p`):`
   pre-conditions: There exists a node with data pointer equal to p in X.
   action: Deletes this node from X.

(3) `reference exact_search(table` R_r, `attr` A_1, `param` P_1, ..., `attr` A_a, `param` P_a, `direction` $dir_{a+1}, \ldots,$ `direction` dir_b, `efficient predicate` γ `):`
   pre-conditions:
   (a) $\gamma \in \gamma^e(X)$.
   (b) There exists a node n_r in G^t with table R_r. Let $\langle n_1, \ldots, n_r \rangle$ be the path from the root of G^t to the node n_r. We will refer to this path as K.
   (c) The attributes in $\mathcal{L}^{\uparrow}(n_r)$ are exactly $\langle A_1, \ldots, A_b \rangle$ in this order, where $1 \leq a \leq b$.
   (d) Either K is clustered relative to a or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$ `TRUE` iff $t \in \gamma(R_r)$ for any object t.
   return value: Let O be the ordering $\langle A_{a+1}\ \mathtt{dir}_{a+1}, \ldots, A_b\ dir_b \rangle$. This method returns the `ID` of the first object t in X, relative to the order O, for which:
   (a) $t \in R_r$,
   (b) $\gamma(t)$ holds,
   (c) $\bigwedge_{i=1}^{a} (t.A_i = P_i)$.
   If such an object does not exist, then the method returns `NULL`.

(4) `reference closest_search(table` R_r, `attr` A_1, `param` P_1, ..., `attr` A_a, `param` P_a, `direction` $dir_a, \ldots,$ `direction` dir_b, `efficient predicate` γ `):`
   pre-conditions:
   (a) $\gamma \in \gamma^e(X)$.
   (b) There exists a node n_r in G^t with table R_r. Let $\langle n_1, \ldots, n_r \rangle$ be the path from the root of G^t to the node n_r. We will refer to this path as K.
   (c) The attributes in $\mathcal{L}^{\uparrow}(n_r)$ are exactly $\langle A_1, \ldots, A_b \rangle$ in this order, where $1 \leq a \leq b$.
   (d) Either K is clustered relative to $a-1$ or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$ `TRUE` iff $t \in \gamma(R_r)$ for any object t.

return value: Let O be the ordering $\langle A_a\ \mathtt{dir}_a, \ldots, A_b\ dir_b \rangle$. This method returns the ID of the first object t in X, relative to the order O, for which:
(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
If such an object does not exist, then the method returns NULL.

(5) `reference next(table` R_r, `attr` A_1, `param` P_1, ..., `attr` A_a, `param` P_a, `direction` dir_{a+1}, ..., `direction` dir_b, `reference` p, `efficient predicate` γ`):`

pre-conditions:
(a) $\gamma \in \gamma^e(X)$.
(b) There exists a node n_r in G^t with table R_r. Let $\langle n_1, \ldots, n_r \rangle$ be the path from the root of G^t to the node n_r. We will refer to this path as K.
(c) $\mathcal{L}^{\uparrow}(n_r)$ contains exactly the attributes $A_1, \ldots, A_b$ in this order and $a \leq b$.
(d) There exists an object $t \in R_r$ with ID p pointed to by a node in X.
(e) Either K is clustered relative to a or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$ TRUE iff $t \in \gamma(R_r)$ 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. When the described object does not exist, the method returns NULL.

Theorem 3.10 (interface of an extended index). *Consider an extended index X with description tree G^t. Assuming that the sizes of the objects pointed to by the index are constant, then each method of the interface of X takes $O(log(|X|) \cdot |def(X)|)$ worst-case time.*

Proof: First, consider the insert(p) method. The method is executed by first retrieving the object stored at location p that belongs to a table of one of the leaf nodes of G^t (Definition 3.2 implies that these tables are disjoint). We will refer to this object as t. Next, we need to find where the node for t should be inserted in the search tree. This is done in the expected way by starting at the root node of the search tree. At every node, the attributes and the type of t are compared with the attributes and types of the objects pointed to by each visited node in the search tree and then, based on the result of the comparison, branching to the left or right child node occurs. $O(log(|X|)$ nodes need to be visited and $O(|def(X)|)$ work needs to be performed at each node to decide whether the insertion point is in the left or in right subtree (total work is $O(log(|X|) \cdot |def(X)|)$). After the new node is inserted in the correct place, its marker bits need to be calculated and the marker bits of its ancestor nodes may need to be recalculated. In particular, if the γ-condition of X is not trivial, then the insertion will also involve checking whether $\gamma(t)$ is true for each predicate γ in $\gamma(X)$. If this is the case for the jth predicate in $\gamma(X)$, then the jth marker bit of the inserted node and all its ancestors should be set. Of course, once

an ancestor that has its jth marker bit set is reached, the process can stop because this guarantees that all its ancestors of this node will have their j^{th} marker bit set. Since $O(log(|X|))$ nodes are visited and $O(|def(X)|)$ work is done at each node, the time to recalculate the marker bits will be also $O(log(|X|) \cdot |def(X)|)$. Finally, a rebalancing of the search tree may be needed. This takes $O(log(|X|))$ worst-case time because $O(log(|X|))$ nodes are visited and the rotation at each visited note takes constant time. Node that updating the marker bits for each rotation also takes constant time because the marker bits for the ancestor nodes do not need to be modified because their ancestors do not change.

Next, consider the `delete(`p`)` method. It starts by retrieving the object stored at location p, which we will refer to as t. Next, we need to find where the node for t is in the search tree and remove it. Since a total ordering is defined of the nodes in the index, this can be done in $O(log(|X|) \cdot |def(X)|)$ time as described in the previous paragraph. The algorithm for AA trees deletes a node by moving at most $O(log(|X|))$ nodes, where each node move takes constant time. The rebalancing algorithm is similar to the one for insertion and has the same time complexity.

Next, consider the call `exact_search(`$R_r, A_1, P_1, \ldots, A_a, P_a, dir_{a+1}, \ldots, dir_b, \gamma$`)`. The algorithm will search for the first object t in R_r, relative to O, that passes the efficient predicate γ, and $\bigwedge_{i=1}^{a} (t.A_i = P_i)$. If K is clustered relative to a, then this can be done efficiently because the objects in R_r that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ will be clustered together relative to the tree order. The marker bits that are used to compute γ can be used to determine whether the subtree rooted at the current node will contain a node that points to an object for which γ is true and therefore whether the subtree should be examined or pruned out. In the second case, the objects will not be clustered, but the marker bits that are used to compute γ' will be used to prune out objects that do not belong to R_r or do not pass the predicate γ.

The only non-trivial part is finding the first object in relative to the order O in the identified set. This can be done by finding the first node, relative to the tree order, in the set when dir_{a+1} is `asc` and the last otherwise. Then, within this value of A_{a+1} the first object is found when $dir_{a+1} =$ `asc` and the last when $dir_{a+1} =$ `desc` and so on. The computations will take $O(|def(X)| \cdot log(|X|))$ worst-case time.

Now, consider the call `closest_search(`$R_r, A_1, P_1, \ldots, A_a, P_a, dir_a, \ldots, dir_b, \gamma$`)`. The algorithm is similar to the algorithm for `exact_search`. The difference is that now nodes that point to objects for which γ is true and for which $\bigwedge_{i=1}^{a-1} (t.A_i = P_i)$ will be searched. Next, depending on the value of dir_a, the algorithm will search among those nodes for the first node that points to an object for which $t.A_a > P_a$ when $dir_a =$ `asc` and for the last object for which $t.A.a < P_a$ when $dir_a =$ `desc`. The asymptotic complexity will not change.

Lastly, consider the call `next(`$R_r, A_1, P_1, \ldots, A_a, P_a, dir_{a+1}, \ldots, dir_b, p, \gamma$`)`. We will examine two cases: when K is clustered relative to a and when it is not.

In the first case, the algorithm will look for the first object t' in R_r after the object t (where t is the object with `ID` p), relative to the order O that passes the efficient predicate γ and for which $\bigwedge_{i=1}^{a} (t'.A_i = P_i)$ holds. However, in this case

(*type*)	(*query*)
(1)	select $B_1, \ldots, B_b$ from R [where $A_1 = :P_1$ and ... 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 ... 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 a materialized view with query "select * from R_k where $\gamma(R_k)$", $\gamma \in \gamma^e(X)$.
(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 or there exists a $\gamma' \in \gamma^e(X)$ that has the property that for an object t, $\gamma'(t) =$ TRUE iff $t \in \gamma(R_k)$.

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

the objects in R_r with the same value for the attributes $\{A_i\}_{i=1}^{a}$ are clustered together relative to the order defined by G^t and therefore the search can be done in $O(|def(X)| \cdot log(|X|))$ time. Again, the only non-trivial part is following the order O, which can be done the same way as it was done for the previous two methods.

In the second subcase, the algorithm will look for the first object t' after the object t, relative to the order O, that passes the predicate γ' and for which $\bigwedge_{i=1}^{a}(t'.A_i = P_i)$ holds. This can be done in $O(|def(X)| \cdot log(|X|))$ time because $\gamma' \in \gamma^e(X)$. ■

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.

Definition 3.11 (*queries of a rooted path of an extended index*). Let X be an extended index. Table III shows the set of sSQL queries that we will associate with a rooted path K of the index. We will denote this set as $Q_K(X)$.

Definition 3.12 (*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 described in points 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 $\texttt{cl}(\bar{Q})$, that describes the queries that can be created from the queries $\texttt{cl}(\bar{Q})$ by merging sorting the results.

Definition 3.13 ($cl(\bar{Q})$). Let $\bar{Q}$ be a set of queries that always produce disjoint set of objects, where each resulting set is ordered relative to the order O. We define the expression $cl(\bar{Q}) \equiv \{Q | \exists \{Q_1, \ldots, Q_k\} \subseteq \bar{Q},\ Q \equiv \texttt{merge_sort}(Q_1, \ldots, Q_k, O)\}$, where the method `merge_sort` returns the objects "Q_1 `union all` ... `union all` Q_k" in the order O.

THEOREM 3.14 (EFFICIENTLY SUPPORTED QUERIES BY AN EXTENDED INDEX). *Let X be an extended index. Then $cl(\bigcup_{K \in rp(X)} Q_K(X)) \subseteq Q(X)$. Moreover, if $Q \in Q(X)$, then there exists a query Q' equivalent to Q and $Q' \in cl(\bigcup_{K \in rp(X)} Q_K(X))$.*

Proof:

$\Leftarrow$ We will first prove that $cl(\bigcup_{K \in rp(X)} Q_K(X)) \subseteq Q(X)$. Indeed, suppose that $Q \in cl(\bigcup_{K \in cr(X)} Q_K(X))$. Then there must exist queries $\{Q_i\}_{i=1}^{q}$ such that $Q \equiv \texttt{merge_sort}(Q_1, \ldots, Q_q, O)$, $Q_i \in \bigcup_{K \in rp(X)} Q_K(X)$ for $i = 1$ to q, $\{Q_i\}_{i=1}^{q}$ are sSQL queries, and the results of the q queries are ordered according to O. Next, note that $Q_i \in Q(X)$ for $i = 1$ to q implies that $Q \in Q(X)$ because the access plan for Q that returns the results of $\texttt{merge_sort}(Q_1, \ldots, Q_q, O)$ can be constructed by applying a merge sort on the results of the queries $\{Q_i\}_{i=1}^{q}$, which can be done in order size of the result time. We therefore need to show that if $Q \in Q_K(X)$ for some $K \in rp(X)$, then $Q \in Q(X)$.

The fact Property 1 from Definition 3.12 holds is a direct consequence of Definition 3.11. Therefore, we only need to show that Property 2 from Definition 3.12 holds for Q, this is, that Q can be efficiently answered using X.

Suppose the nodes of K starting with the root node of the description tree of X are $\langle n_1, \ldots, n_k \rangle$ and the table of n_k is R_k. Let the attributes in $\mathcal{L}^{\uparrow}(n_k)$ be $\langle A_1, \ldots, A_b \rangle$ in this order.

First, consider the case when Q is a query of Type 1 (see Table III). Such a query can be efficiently answered by executing the following pseudo-code.

```
(1) for (i = s+1; i <= b; i++) dir_i = asc;
(2) p = X.exact_search(R_k, A_1, P_1, ..., A_l, P_l, dir_{l+1}, ..., dir_b, γ);
(3) while (p ≠ NULL){
(4)     t = R_k.retrieve(p);
(5)     send construct_result(B_1 = t.B_1, ..., B_b = t.B_b);
(6)     p = X.next(R_k, A_1, P_1, ..., A_l, P_l, dir_{l+1}, ..., dir_b, p, γ);
(7) }
```

Note that the primitive `construct_result` is used to construct a resulting object out of the specified values for the attributes, while the `retrieve` primitive is used to retrieve the object with the specified `ID`. The primitive `send` is used to denote

the sending of the calculated object to the query result. Note as well that when Q is of the described type, Condition 4 from Table III must hold and therefore the `next` and `search` methods can be invoked on X.

The access plan first returns the first object t in the index that is in $\gamma(R_k)$ and for which $\bigwedge_{i=1}^{l}(t.A_i = P_i)$ holds. Then, it scans forward for the other objects in the index for which the described condition holds. Theorem 3.10 guarantees the resulting tuples are returned in the desired time bound for efficient query plans. As well, the directions in the calls to the methods `search` and `next` guarantee that the resulting tuples are returned in the order $\langle A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s, A_{s+1}$ `asc`, $\ldots$, A_b `asc`$\rangle$, that is, the presented access plan is correct and efficient.

Next, consider the case when Q is a query of Type 2 (see Table III). If the query does not contain an `order by` condition, then we will add `order by` A_{l+1} `asc` to the query. (Note that this will not change the semantics of the query.) The query can be efficiently answered using the following pseudo-code.

```
(01) for (i = s+1; i ≤ b; i++) dir_i = asc;
(02) if (dir_{l+1} == asc) P = P_{l+1} else P = P_{l+2};
(03) p = X.exact_search(R_k, A_1, P_1, ..., A_l, P_l, A_{l+1}, P, dir_{l+1}, ..., dir_b, γ);
(04) if (p ≠ NULL) t = X.retrieve(p);
(05) if (p == NULL) {
(06)     p = X.closest_search(R_k, A_1, P_1, ..., A_l, P_l, A_{l+1}, P, dir_{l+1}, ..., dir_b, γ);
(07)     if (p == NULL) return;
(08)     t = X.retrieve(p);
(09) }
(10) while ((t.A_{l+1} ≥ P_{l+1}) && (t.A_{l+1} ≤ P_{l+2})) {
(11)     send construct_result(B_1 = t.B_1, ..., B_b = t.B_b);
(12)     p = X.next(R_k, A_1, P_1, ..., A_l, P_l, dir_{l+1}, ..., dir_b, p, γ);
(13)     if (p == NULL) return;
(14)     t = X.retrieve(p);
(15) }
```

The query plans assumes that $P_{l+1} \leq P_{l+2}$. Note as well that Condition 4 from Table III must hold for Q and therefore the `next` and `search` methods can be invoked.

When the scan on A_{l+1} is forward, that is, $dir_{l+1} =$ `asc`, the pseudo-code sets $P = P_{l+1}$ (i.e., the scan will be from $A_{l+1} = P_{l+1}$ to $A_{l+1} = P_{l+2}$). When the scan on A_{l+1} is backwards, that is, $dir_{l+1} =$ `desc`, the pseudo-code sets $P = P_{l+2}$ (i.e., the scan will be from $A_{l+1} = P_{l+2}$ to $A_{l+1} = P_{l+1}$) (see Line 2 of the pseudo-code). Line 4 of the code covers the case when there exists an object $t \in R_k$ in X that has the property that $\bigwedge_{i=1}^{l}(t.A_i = P_i)$, $t.A_{l+1} = P$, and $\gamma(t)$ all hold. When such an object exists, the method `exact_search` retrieves the first such object relative to the defined order. If such an object does not exist (Lines 6-8), then the pseudo-code finds the first object $t \in R_r$, relative to the desired order, that passes the condition γ, for which $\bigwedge_{i=1}^{l}(t.A_i = P_i)$ holds, and for which $t.A_{l+1} > P$ when $dir_{l+1} =$ `asc` and

$t.A_{l+1} < P$ when $dir_{l+1} =$ `desc`. When such an object exists, it is a candidate to be the first object in the query result.

Lines 10-15 of the code first check if the found object has value for A_{l+1} within the $[P_{l+1}, P_{l+2}]$ interval. Then the `next` method is repeatedly called to return the rest of the result. Theorem 3.10 guarantees that the objects are returned in the desired time bound for efficient query plans. As well, the resulting objects are returned in the order $\langle A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s, A_{s+1}$ `asc`, …, A_b `asc`$\rangle$, that is, the presented access plan is correct and efficient.

Finally, consider the case when Q is a query of third type (see Table III). Such a query can be answered in constant time using the following access plan.

(1) `tuple` $t = R_k$.`retrieve`(P_1);
(2) `if` $\gamma(t)$ `send` construct_result$(B_1 = t.B_1, \ldots, B_d = t.B_d)$;

$\Rightarrow$ We will next show that if $Q \in Q(X)$, then there exists a query Q' that is equivalent to Q and for which $Q' \subseteq cl(\bigcup_{K \in rp(X)} Q_K(X))$. Indeed, let $Q \in Q(X)$.

Note that from Definition 3.12 it follows that Q must reference a single table R such that $R \equiv (\gamma_1(R_1)$ `union all` … `union all` $\gamma_q(R_q))$, where $\{\gamma_i\}_{i=1}^q$ are in the extended γ-condition of X and $\{R_i\}_{i=1}^q$ are tables in the description tree of X. Note that if Q contains an `order by` condition, then there must be an efficient way to produce $\gamma_i(R_i)$ according to that order. The reason is that sorting N elements takes $O(N \cdot log(N))$ without using additional data structures.

Let Q_i be the query Q in which R is substituted with $\gamma_i(R_i)$ for $i \in [1..q]$. Then, since there exists an efficient plan for answering Q, there should exist efficient plans for answering $\{Q_i\}_{i=1}^q$ (Q can be evaluated using $\{Q_i\}_{i=1}^q$ by performing a merge sort). It remains to prove that for every query Q_i there exists equivalent query Q'_i such that $Q'_i \in \bigcup_{K \in rp(X)} Q_K(X)$ for $i \in [1..q]$. Indeed, let K_i be a rooted path in the description tree of X that passes through the node with table R_i in description tree of X for $i \in [1..q]$. We will show that $Q'_i \in Q_{K_i}(X)$ for $i \in [1..q]$.

First, we will consider the case when Q_i is a sSQL query of Type 1 or 2 - see Table II. Let $K_i = \langle n_1, \ldots, n_k \rangle$ be the rooted path in the description tree that ends at the node with table R_i. Since Q_i is a sSQL query of Type 1 or 2, it will have the following general form.

select $D_1, \ldots, D_d$
from $\gamma_i(R_i)$
where $A_1 = :P_1$ and … $A_l = :P_l$ and A_{l+1} between $:P_{l+1}$ and $:P_{l+2}$
order by $A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s$

Since Q_i can be efficiently answered using X, then the first s attributes in $\mathcal{L}^{\uparrow}(n_k)$ must be $A_1, \ldots, A_s$, where only the order of the first l attributes may be changed. Let Q'_i be a rewrite of the query Q_i in which the order of appearance of the attributes $\{A_i\}_{i=1}^l$ is changed to reflect the order in $\mathcal{L}^{\uparrow}(n_k)$. Then $Q'_i \in Q_{K_i}(X)$ - see Table III. In particular, Condition 1-3 in Table III are true because of the way Q'_i was constructed. Condition 4 is true as a consequence of the fact that Q_i can be efficiently answered using X and Lema 3.8.

Next, consider the case when Q_i is a sSQL query of Type 3. Then Q_i has the following general syntax (see Table II).

```
select D_1, ..., D_d
from γ_i(R_i)
where ID = :P_1
```

Therefore, Q_i is a query of Type 3 for $Q_{K_i}(X)$ (see Table III), which implies that $Q_i \in Q_{K_i}(X)$. ■

4. INDEX MERGING

In this section we present a solution to the problem of creating the smallest possible set of extended indices that can efficiently answer the specified set of sSQL queries.

Note that for every sSQL there exist a number of extended indices that can be used to efficiently answer it. For example, consider the following query over our example company schema presented in Figure 3.

```
select *
from Trainee
where completionLevel > :P
```

The extended index $\langle \mathit{Trainee}, \langle \mathit{completionLevel}, A \rangle\rangle$ can be used to efficiently answer the query regardless of the value for the attribute A. Note as well that every extended index can be used to efficiently answer a number of critical queries – see Theorem 3.14.

Suppose that we are given the sSQL queries $\{Q_i\}_{i=1}^k$. Our strategy for solving the problem is to first construct the sets of extended indices $\{\bar{X}_i\}_{i=1}^k$, where $\bar{X}_i$ consists of exactly the set of extended indices that can be used to answer Q_i efficiently. Let $\bar{X}$ be the set of all extended indices that are valid over the defined database schema. Then the sets $\{\bar{X}_i\}_{i=1}^k$ divide $\bar{X}$ into up to 2^k distinct subsets (e.g., the set of extended indices that are in $\bar{X}$ but not in $\{\bar{X}_i\}_{i=1}^k$, the set of extended indices that are in $\bar{X}_1$ but not in $\{\bar{X}_i\}_{i=1}^{k-1}$ and so on). Our algorithm will then pick m of these subsets that cover $\{\bar{X}_i\}_{i=1}^k$, that is, for every $i \in [1..k]$ there exists a picked subset that contains at least one element of $\bar{X}_i$. The solution can then be constructed by picking the smallest extended index from each of the subsets, where the size of an extended index is approximated using the defined statistical information. To find the optimal solution, the algorithm will consider all possible ways of generating the subsets that cover $\{\bar{X}_i\}_{i=1}^k$.

In the above algorithm, we will use a *parameterized access requirement type* (PART) to describe a set of indices. We continue by presenting a formal definition of a PART and its properties followed by our algorithm for extended index merging.

4.1 Syntax and Semantics of a PART

The definition of the syntax of a PART follows.

Definition 4.1 (*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 the siblings, where each node has a label of the following form.

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 $a > 0$ and $m > 0$.

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 `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 γ 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.

We next present several intermediate definitions that are needed in defining the semantics of a PART.

Definition 4.2 (*concatenating ordering labels*). If L_1 and L_2 are ordering labels, then we define their concatenation as follows.

$$L_1 \circ L_2 = \begin{cases} \text{“}L_1\text{”} & \text{, if } L_2 \text{ is empty;} \\ \text{“}L_2\text{”} & \text{, if } L_1 \text{ is empty;} \\ \text{“}L_1, L_2\text{”} & \text{, otherwise.} \end{cases}$$

Definition 4.3 (*the $\lfloor \cdot \rfloor$ operation*). The following nondeterministic operation converts a set of attributes into either the empty string or an E-component, where ϵ is used to denote the empty string.

$$\lfloor \{A_1, \ldots, A_a\} \rfloor = \begin{cases} \epsilon & \text{, if } a = 0; \\ \text{the string: “}\{A_1, \ldots, A_a\}\text{”} & \text{, otherwise.} \end{cases}$$

Definition 4.4 (*permutations for a PART ordering label*). Let L be an ordering label. We will use $\Pi(L)$ to denote the result of applying the κ operation to one or more of the E-components of L one or more times. Let E be the E-component $\{A_1, \ldots, A_a\}$ and let $\langle A'_1, \ldots, A'_a \rangle$ be a permutation of $\langle A_1, \ldots, A_a \rangle$. Then the κ operation converts E into a nondeterministic way into one of the following expressions:

(1) $\{A'_1, \ldots, A'_a\}$,
(2) $A \circ \lfloor E - \{A\} \rfloor$,
(3) $\lfloor E - \{A\} \rfloor \circ A$, or
(4) $\{A'_1, \ldots, A'_k\}, \{A_{k+1}, \ldots, A_a\}$

where $1 \leq k < a$. 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 complete permutations.

Definition 4.5 (*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 4.6 (*fixed PART*). A fixed PART has no E-components of size greater than one in its ordering labels.

(*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 IV. Three example critical queries and the corresponding PARTs

Note that PARTs are closed under the permutation operation, that is, if $\mathcal{P}$ is a PART, then so is $\Pi(\mathcal{P})$ for any valid permutation Π over $\mathcal{P}$. Note as well that the result of applying a complete permutation to a PART is a fixed PART.

Definition 4.7 (*fixed PART* $\Rightarrow$ *extended index*). Let $\mathcal{P}$ be a fixed PART. 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 E-components.

(3) If $\{n_i\}_{i=1}^{k}$ are the children of the node n and $\bigcup_{i=1}^{k} \mathtt{table}(n_i) \subset \mathtt{table}(n)$, then add a new child node to n with table R that includes the objects that are in $\mathtt{table}(n)$ but not in $\mathtt{table}(n_i)$ for $i = 1$ to k.

Definition 4.8 (*valid PART*). A PART $\mathcal{P}$ is valid iff for every complete permutation Π^c of $\mathcal{P}$, $\Pi^c(\mathcal{P})$ is converted into a valid extended index by the procedure described in Definition 4.7.

Definition 4.9 (*cover of a set of indices*). Let $\bar{X}$ be a set of extended indices. Then $\mathtt{cover}(\bar{X}) = \{X | \exists X' \in \bar{X}, Q(X') \subseteq Q(X)\}$.

Definition 4.10 (*extended indices 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 4.7. We will also use $X^c(\mathcal{P})$ to denote the set $\mathtt{cover}(X(\mathcal{P}))$.

Definition 4.11 (*extended indices 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 $\Pi^c(\mathcal{P})$. We will also use $X^c(\mathcal{P})$ to denote the set $\mathtt{cover}(X(\mathcal{P}))$. We will refer to $X(\mathcal{P})$ as the *underlying* set of extended indices for $\mathcal{P}$ and to $X^c(\mathcal{P})$ as the set of extended indices for $\mathcal{P}$.

Going back to our motivating example from Section 1.2, the queries from Table I will generate the PARTs shown in Table IV. We will later show that each PART

(*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 V. The PARTs for the three sSQL query types

represents (i.e., see Definition 4.11) exactly the set of indices that can be used to efficiently answer the respective query. Note that all PARTs in the table are fixed PARTs. $\mathcal{P} = \langle Customer, \langle\{name, balance\}\rangle\rangle$ is an example of a PART that is not fixed and that can be generated from the following query.

```
select * from Customer
where name = :P1 and balance = :P2
```

As definition 4.5 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 PART merging step will merge the PARTs from Table IV 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 iff t is also an object in V_T (i.e., t is a *Trainee* object for which $completionLevel = 1$). Note that we do not need to add to the PART a γ-condition that selects the objects in the table *Customer* because the query Q_2 contains a partial match predicate on the attribute *name*. Note that $X^c(\mathcal{P}) = \bigcap_{i=1}^{3} X^c(\mathcal{P}_i)$, that is, the set of indices that the created PART represents is exactly the intersection of the extended indices of the initial PARTs.

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. Note that $X \in X(\mathcal{P})$ and therefore $X \in X^c(\mathcal{P})$.

We next formally present the two steps of our algorithm.

4.2 Step 1 - Converting Extended Indices into PARTs

Table V shows the PARTs that we will produce for each given sSQL query type (see Table II). Note that for a sSQL query of Type 3 we do not need to create a PART because we can retrieve an object from its address in constant time without using an index.

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

THEOREM 4.12 (CORRECTNESS OF PART CREATION). *If the mapping from Table V 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: First, suppose that Q is a sSQL query of Type 3 (see Table II). Then Q can be answered in constant time.

Second, suppose that Q is a sSQL query of Type 1 or 2 (see Table II), that is, $\mathcal{P} = \langle R, \langle\{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a\rangle\rangle$. Then $X(\mathcal{P})$ contains only extended indices of the form: $\langle R, \langle A'_1, \ldots, A'_l, A_{l+1}, \ldots, A_a\rangle\rangle$, where $\langle A'_1, \ldots, A'_l\rangle$ is a permutation of the attributes $\{A_i\}_{i=1}^{a}$. We will denote this set as $\bar{X}$.

$\Leftarrow$ Suppose that $X_1 \in X^c(\mathcal{P})$. Then by Definitions 4.9 and 4.11 it follows that if $X \in \bar{X}$, then $Q(X) \subseteq Q(X_1)$. But the extended indices in $\bar{X}$ can be used to efficiently answer Q and therefore X_1 can efficiently answer Q.

$\Rightarrow$ Suppose that X_1 can be used to efficiently answer the sSQL query Q, that is, $Q \in Q(X_1)$. But if in extended index X_2 can be used to efficiently answer the sSQL query Q, then it must be the case that $Q(X) \subseteq Q(X_2)$ for some $X \in \bar{X}$ (see Theorem 3.14). This implies that $X_1 \in \texttt{cover}(\bar{X})$ or $X_1 \in X^c(\mathcal{P})$. ■

4.3 Step 2 – PART Merging

In this section we present a procedure for merging PARTs (i.e., 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.

4.3.1 *Merging Ordering Labels.* A procedure for merging PARTs required a way for merging the ordering labels of 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. A formal definition of the $\oplus$ operation follows.

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

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 set of attributes. Note as well that the "$\oplus$" function is partial, that is, not every two ordering labels are mergeable. Since "$\oplus$" is a commutative and an associative operation, 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 4.13 (*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* iff for all $i, j \in [1..k]$ either L_i is a prefix of L_j or L_j is a prefix of L_i.

THEOREM 4.14. (*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 com-*

patible, then there exists a complete permutation Π^c_{k+1} 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^c_{k+1} be a complete permutation for L_{k+1} and $L'_{k+1} = \Pi^c_{k+1}(L_{k+1})$. Then there exist complete permutations $\{\Pi^c_i\}_{i=1}^k$ 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 $\overset{k}{\underset{i=1}{\oplus}} L_i$ returns `UNDEFINED`, then there do not exist complete permutations $\{\Pi^c_i\}_{i=1}^k$ 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: 1.a) Suppose that $\{\Pi^c_i\}_{i=1}^k$ are complete permutations that convert $\{L_i\}_{i=1}^k$ into the F-components $\{L'_i\}_{i=1}^k$ and the attribute orderings defined by $\{L'_i\}_{i=1}^k$ are compatible. We will prove the there exists a complete permutation Π^c_{k+1} that converts L_{k+1} into L'_{k+1} and for which the attribute orderings defined by $\{L'_i\}_{i=1}^{k+1}$ are compatible.

Since the attribute orderings $\{L'_i\}_{i=1}^k$ are compatible, it must be the case that all labels that have at least j attributes have the same value for the jth attribute - we will refer to it as A_j. Let $|L_{k+1} = l|$ and let Π^c_{k+1} be the permutation that converts L_{k+1} into $A_1, \ldots, A_l$. (The existence of such a permutation can be proven by induction on l and is based on the definition of the "$\oplus$" operation.) Then for this permutation the attribute orderings $\{L'_i\}_{i=1}^{k+1}$ are compatible.

1.b) Suppose that Π^c_{k+1} is a complete permutation that converts L_{k+1} into $L'_{k+1} = A_1, \ldots, A_l$. Let $\{\Pi^c\}_{i=1}^k$ be the complete permutations that convert L_i into $L'_i = A_1, \ldots A_{|L_i|}$. Then the attribute orderings defined by $\{L'_i\}_{i=1}^k$ are compatible. (Again, this can be proven by induction on l using the definition of the "$\oplus$" operation.)

2. Let $\overset{k}{\underset{i=1}{\oplus}} L_i$ returns `UNDEFINED`. Let L' be the resulting prefix generated during the recursive calls and suppose that the $\oplus$ method returned `UNDEFINED` when trying to merge L^1 and L^2, that is: $\Pi_1(L_1) = L' \circ L^1$, $\Pi_2(L_2) = L' \circ L^2$, $|L^1|, |L^2| > 0$, Π_1 and Π_2 are valid permutations for L_1 and L_2, respectively, and one of the following is true for L^1 and L^2.

(1) L^1 starts with the E-component E_1, L^2 starts with the E-component E_2, $E_1 \not\subseteq E_2$, and $E_2 \not\subseteq E_1$.
(2) One of the ordering labels starts with the E-component E, the other starts with the attribute A, and $A \notin E$.
(3) L^1 starts with the attribute A, L^2 starts with the attribute B, and $A \neq B$.

We will show that in all three cases there do not exist complete permutations $\{\Pi^c_i\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into ordering labels with compatible attribute orderings.

Consider first Case 1 and suppose that there exist complete permutations $\{\Pi^c_i\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into two ordering labels with attribute orderings that are compatible. This implies that $E_1 \subseteq E_2$ or $E_2 \subseteq E_1$ – contradiction.

Consider next Case 2 and suppose that there exist complete permutations $\{\Pi^c_i\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into two ordering labels with attribute orderings that are com-

patible. Without loss of generality we assume that L^1 starts with A and L^2 starts with E. Therefore, Π^c will convert L^2 into an ordering label that starts with the attribute A and therefore $A \in E$ – contradiction.

Lastly, consider Case 3 and suppose that there exist complete permutations $\{\Pi^c_i\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into two ordering labels with attribute orderings that are compatible. But then every complete permutation will convert L^1 into an ordering label that starts with the attribute A and every complete permutation for L^2 will convert it into an ordering label that starts with the attribute B – contradiction. ■

4.3.2 *Complexity of PART merging.* We define a *simple* PART as follows.

Definition 4.15 (*simple PART*). A PART is *simple* if it has the format shown in Table V.

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 4.16 (*PART merging*). We will say that the PART $\{\mathcal{P}_i\}_{i=1}^k$ are mergeable into the PART $\mathcal{P}$ iff 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 $i \in [1..k]$ there exists $j \in [1..k]$ $(j \neq i)$ 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.

Consider the tables R, R_1, R_2, and R_3 and suppose that $R_1 \subseteq R$, $R_2 \subseteq R$, $R_3 \subseteq R$, $R_2 \cap R_3 = \emptyset$, $R_1 \cap R_2 \neq \emptyset$, and $R_1 \cap R_3 \neq \emptyset$. Consider the PARTs $\mathcal{P} = \langle R, \langle\{A\}\rangle\rangle$ and $\mathcal{P}_i = \langle R_i, \langle\{A, B_i\}\rangle\rangle$ for $i = 1$ to 3. If $\mathcal{P}$ and $\mathcal{P}_1$ are merged together in the PART $\mathcal{P}^1$, then we will show that $\mathcal{P}^1$ and $\mathcal{P}_2$ will not be mergeable, $\mathcal{P}^1$ and $\mathcal{P}_3$ will not be mergeable, and $\mathcal{P}_2$ and $\mathcal{P}_3$ will not be mergeable.

First, suppose that $\mathcal{P}^1$ and $\mathcal{P}_2$ are mergeable. This implies that there exists an extended index X that can be used to efficiently answer the queries shown in Table VI. Let t be an object in R_1 and R_2 (recall that $R_1 \cap R_2 \neq \emptyset$). Then from Theorem 3.14 follows that the the description tree for X must contain a node n with a table R' such that $t \in R'$. Moreover, $\mathcal{L}^{\uparrow}(n)$ must start with the attributes $\{A, B_1\}$ in some order so that the second query from Table VI is efficiently supported by X. $\mathcal{L}^{\uparrow}(n)$ must also start with the labels $\{A, B_2\}$ in some order so that the third query from Table VI is efficiently supported by X - contradiction. The PARTs $\mathcal{P}^1$ and $\mathcal{P}_3$ cannot be merged for the same reason. Lastly, the fact that the PARTs $\mathcal{P}_2$ and $\mathcal{P}_3$ are not mergeable is a direct consequence of Definition 4.16.

A different alternative is creating the PART $\langle R, \langle\{A\}\rangle, [\langle R_2, \langle\{B_2\}\rangle, \langle R_3, \langle\{B_3\}\rangle]\rangle$ as the result of merging $\mathcal{P}$, $\mathcal{P}_2$ and $\mathcal{P}_3$. If the smallest extended indices that correspond to the PARTs $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ are all the same and equal to one unit, then the merging or $\mathcal{P}$ with $\mathcal{P}_1$ will result in a saving of one unit, while the presented

(*name*)	(*value*)
(1)	`select *` `from` R `where` $A =\ :P_1$
(2)	`select *` `from` R_1 `where` $A =\ :P_1$ `and` $B_1 =\ :P_2$
(2)	`select *` `from` R_2 `where` $A =\ :P_1$ `and` $B_2 =\ :P_2$

Table VI. Three example sSQL queries

alternative solution (i.e., merging $\mathcal{P}$ with $\mathcal{P}_2$ and $\mathcal{P}_3$) will save two units. Therefore, just because two PARTs are mergeable does not imply that merging them will produce a solution of the smallest possible size.

We will next extend the presented example to show that PART merging in NP-Hard. The proof will use a reduction from the set packing problem, which is presented in [Garey and Johnson 1972].

Definition 4.17 (*set packing problem*). `INSTANCE` : A collection of finite sets and a positive integer N.
`QUESTION` : Does the collection contain at least N mutually disjoint sets.

THEOREM 4.18 (PART MERGING IS NP-HARD). *Consider the simple PARTs* $\{\mathcal{P}_i\}_{i=1}^k$. *The problem of merging these PARTs in such a way so that the size of the resulting PARTs is as small as possible is NP-Hard.*

Proof: Suppose that we are given an instance of the set packing problems. Let the given sets be $\{\mathcal{M}_i\}_{i=1}^m$. Let $\mathcal{M}$ be the set $\bigcup_{i=1}^{m} \mathcal{M}_i$ and f be a one-to-one function that maps the elements of $\mathcal{M}$ into the objects of the table R with attributes $A, B_1, \ldots, B_m$. Let R_i be the table consisting of the objects that are the result of applying f to the elements of the set $\mathcal{M}_i$, where $i = 1$ to m. We will create the PARTs $\mathcal{P} = \langle R, \langle \{A\} \rangle \rangle$ and $\{\mathcal{P}_i = \langle R_i, \langle \{A, B_i\} \rangle \rangle \}_{i=1}^m$. Without loss of generality, suppose that $\mathcal{P}_i$ is of size one unit for $i \in [1 \ldots m]$.

Let $\{\mathcal{P}'_i\}_{i=1}^k$ be the solution to the problem of merging the PARTs $\mathcal{P}$ and $\{\mathcal{P}_i\}_{i=1}^m$. We will show that the PARTs $\{\mathcal{P}_i\}_{i=1}^m$ cannot be merged between themselves. Indeed, if $R_1 \cap R_2 = \emptyset$, then Definition 4.16 states that the PARTs $\mathcal{P}_1$ and $\mathcal{P}_2$ cannot be merged. Conversely, if $R_1 \cap R_2 \neq \emptyset$, then $\mathcal{P}_1$ and $\mathcal{P}_2$ cannot be merged for the reasons explained in the example preceding this theorem. Therefore, the set $\{\mathcal{P}'_i\}_{i=1}^k$ consists of a PART $\mathcal{P}'$ and a subset of the PARTs $\{\mathcal{P}_i\}_{i=1}^m$. The PARTs in the set $\{\mathcal{P}_i\}_{i=1}^m$ that are used to create the PART $\mathcal{P}'$ must be based on disjoint tables (see the example preceding the theorem) and therefore if this set contains at least N elements, then the answer to the particular instance of set packing problem is "Yes". Conversely, if this set does not contain N elements, then the answer to the particular instance of the set packing problem is "No". ■

4.4 Exact Algorithm for PART Merging

In the previous subsection we showed that just because two PARTs are mergeable, this doesn't imply that merging them will be part of the optimal solution. In

this subsection we demonstrate other interesting properties or PART merging that unveil the true complexity of the problem.

Consider the simple PARTs $\mathcal{P}_1 = \langle R_1, \langle \{A\} \rangle \rangle$ and $\mathcal{P}_2 = \langle R_2, \langle \{A, B\} \rangle \rangle$ and suppose that $R_2 \subseteq R_1$. One option is to merge those PARTs into the PART $\mathcal{P}^1 = \langle R_1, \langle \{A\} \rangle, [\langle R_2, \langle \{B\} \rangle \rangle] \rangle$. A problem with this merge is that if there exists a PART $\mathcal{P}_3 = \langle R_3, \langle \{A, B\} \rangle \rangle$ such that $R_3 \subseteq R_1$ but $R_2 \cap R_3 \neq \emptyset$, then there is not an easy way to merge $\mathcal{P}^1$ and $\mathcal{P}_3$. In particular, the node for the table R_2 may need to be removed from the PART $\mathcal{P}^1$ in the resulting PART.

Another possibility is to create the PART $\mathcal{P}^1 = \langle \{\gamma\}, R_1, \langle \{A, B\} \rangle \rangle$ as the result of merging $\mathcal{P}_1$ and $\mathcal{P}_2$, where $\gamma(t)$ holds iff $t \in R_2$. A problem with this merge is that if there exists a PART $\mathcal{P}_3 = \langle R_3, \langle \{A, C\} \rangle \rangle$ such that $R_3 \subseteq R_1$ and $R_2 \cap R_3 = \emptyset$, then there is no easy way to merge $\mathcal{P}^1$ and $\mathcal{P}_3$. In particular, the γ condition from $\mathcal{P}^1$ may have to be removed and two new nodes with tables R_2 and R_3 may have to be added to the PART.

The presented example shows that there can be more than one way of merging two PARTs and it is important to choose the correct way. As well, as the next example shows, merging two PARTs can result in the introduction of new tables in the resulting PART.

Consider the PARTs $\mathcal{P}_1 = \langle R_1, \langle \{A, B\} \rangle, \mathcal{P}_2 = \langle R_2, \langle \{A, B\} \rangle$, and $\mathcal{P} = \langle R, \langle \{A\} \rangle \rangle$ and suppose that $R_1 \cap R_2 \neq \emptyset$, $R_1 \subseteq R$, and $R_2 \subseteq R$. Let $R_3 = R_1 - R_2$, $R_4 = R_1 \cap R_2$, $R_5 = R_2 - R_1$, and $R_6 = R_1 \cup R_2$. One possibility is to merge the PARTs in the PART $\langle \{\gamma_1, \gamma_2\}, R, \langle \{A, B\} \rangle \rangle$, where $\gamma_1(t)$ is `TRUE` iff $t \in R_1$ and $\gamma_2(t)$ is `TRUE` iff $t \in R_2$. Another way of merging them is into the PART $\langle R, \langle \{A\} \rangle, [\langle R_3, \langle \{B\} \rangle \rangle, \langle R_4, \langle \{B\} \rangle \rangle, \langle R_5, \langle \{B\} \rangle \rangle] \rangle$, which will allow a PART such as $\langle R_7, \langle \{A, C\} \rangle \rangle$ to be merged with the result of merging the initial two parts as long as $R_7 \subset R$ and $R_7 \cap R_6 = \emptyset$.

Given the innate complexity of PART merging, we will use an *Oracle* to decide how the initial simple PARTs are to be merged. We do so by enumerating the possible partitionings of the simple PARTs, where we require that all the PARTs from each partition are merged into a single PART and no two PARTs from distinct partitions are merged together. There are $\mathcal{B}_k$ ways to partition k elements, where

$$\mathcal{B}_k = \sum_{i=1}^{k} (\frac{1}{i!} \sum_{j=0}^{i-1} ((-1)^j \binom{i}{j} (i-j)^k))$$

is the kth Bell number [Bell 1934]. Note that $2^k < \mathcal{B}_k < 2^{k \cdot log(k)}$, that is, $\mathcal{B}_k$ grows exponentially.

We will use a different Oracle to determine the tables of the resulting PART. In particular, we will associate three bits with each PART, where we require that each partition contains at least one PART that has at least one bit marked. We also require that if the first bit of a PART is marked, then the second and third bit are not and, conversely, if the second or third bit of a PART is marked, then the first bit is not.

If the first bit of a simple PART with table R is marked, then the resulting PART will contain the table R in one of its labels. All the PARTs in a partition that have the second bit marked will be used to create a set of new tables for the resulting PART. In particular, if the simple PARTs that have the second bit marked are

based on the tables $\bar{R}$ and $\{R_i\}_{i=1}^r$ is the result of removing duplicate tables, then the resulting PART will include the tables of the form $R_1^{j_1} \cap \ldots \cap R_r^{j_r}$ that are not empty, where j_i is 1 or -1 for $i \in [1 \ldots r]$, $\{j_i\}_{i=1}^r$ are not all -1, and R_i^1 is the table R_i and R_i^{-1} is the result of the query: $\bigcup_{k \in [1 \ldots r]}^{k \neq i} R_k - R_i$. Note that the so produced tables are disjoint and partition the set of the tuples in the tables $\{R_i\}_{i=1}^r$ into at most $2^r - 1$ parts. We will refer to this tables as `intersections`$(\{R_i\}_{i=1}^r)$.

All the PARTs in a partition that have the third bit marked will be used to create a single new table for the resulting PART. In particular, if the simple PARTs that have the third bit marked are $\{R_i\}_{i=1}^r$, where duplicate tables are eliminated, then the resulting PART will reference the table $\bigcup_{i=1}^{r} R_i$. We will refer to this expression as `union`$(\{R_i\}_{i=1}^r)$.

Observe that there are $O(\mathcal{B}_k \cdot 2^{3k})$ combined cases enumerated by the two Oracles.

Next we present the pseudo-code for the `merge_PARTs` method. It merges the simple PARTs $\{\mathcal{P}_i\}_{i=1}^m$ that are from the same partition. The method returns the resulting PART when the PARTs are mergeable using only the tables defined by the second Oracle and `UNDEFINED` otherwise.

```
(00) merge_PARTs(PART P_1,..., PART P_m) {
(01)     R̄_1 = norm{R|R ∈ table(P_i) for i ∈ [1..m] and P_i.bit_1 = 1};
(02)     R̄_2 = norm{R|R ∈ table(P_i) for i ∈ [1..m] and P_i.bit_2 = 1};
(03)     R̄_3 = norm{R|R ∈ table(P_i) for i ∈ [1..m] and P_i.bit_3 = 1};
(04)     R̄ = R̄_1 ∪ intersections(R̄_2) ∪ union(R̄_3);
(05)     if (build_tree(R̄) == UNDEFINED) return UNDEFINED;
(06)     G^t = build_tree(R̄);
(07)     ⟨n_1,...,n_r⟩ = breath first traversal of G^t;
(08)     for (i = 1; i ≤ r; i++) {
(09)       P̄_i = {P_j| j ∈ [1..m], P_j contributes to the creation of table(n_i)
and P_j.bit_1 = 1 or P_j.bit_2 = 1};
(10)       L_i = ⊕_{L∈L(P),P∈P̄_i} L;
(11)       P_i = ⟨table(n_i), ⟨L_i⟩⟩;
(12)       if (i == 1) P = P_1 else P = table_PART_merge(P, P_i);
(13)       if (P == UNDEFINED) return UNDEFINED;
(14)     }
(15)     P̄ = {P_j | j ∈ [1...m] and P_j.bit_1 = 0 and P_j.bit_2 = 0}
(16)     for P' ∈ P̄ {
(17)       P = gamma_PART_merge(P, P');
(18)       if (P == UNDEFINED) return UNDEFINED;
(19)     }
(20)     return P;
(21) }
```

The method `norm`$(\bar{R})$ eliminates redundant tables. The method `build_tree`$(\bar{R})$ builds a node labeled tree with labels the tables in the set $\bar{R}$. The tree has the properties:

(1) $\texttt{table}(n_1) \subseteq \texttt{table}(n_2)$ when n_2 is a parent of n_1, and

(2) $\texttt{table}(n_1) \cap \texttt{table}(n_2) = \emptyset$ when n_1 and n_2 are siblings.

When such a tree does not exist, then the `build_tree` method returns `UNDEFINED`.

The method `merge_PARTs` first creates a tree from the tables that are suggested by the two Oracles (Lines 1-6 of the method). The created tree will be the tree for the resulting PART. Next, we do a breath-first traversal of the created tree. If a PART has its first or second bit marked, then we use the method `table_PART_merge` to add it to the resulting PART, where this merge will involve the introduction of a new node to the resulting PART and possibly modifying existing ordering labels. Conversely, all PARTs that have their third bit marked will contribute to creating the table for the root node of the PART and therefore their labels are created after the original population of the tree with ordering labels using the method `gamma_PART_merge`. The method is performed in such a way so that only new γ conditions are introduced and existing ordering labels are modified.

The method `table_PART_merge`$(\mathcal{P}_1, \mathcal{P}_2)$ merges an arbitrary PART $\mathcal{P}_1$ with a simple PART $\mathcal{P}_2 = \langle R, \langle L \rangle \rangle$. The pre-condition for executing this method is the existence of a node n_k in $\mathcal{P}_1$ with table R_k such that $R \subset R_k$ and for every table R' that is a table of a child node of n_k in $\mathcal{P}_1$ it is the case that $R' \cap R = \emptyset$. The method adds a new node with table R to $\mathcal{P}_1$ when possible and returns `UNDEFINED` otherwise. If the 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 6 of the pseudo-code that follows). 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 no new node needs to be created. Without loss of generality, assume that the γ-condition of $\mathcal{P}_1$ is $\bar{\gamma}_1$, $K = \langle n_1, \ldots, n_k \rangle$ is the rooted path in $\mathcal{P}_1$ that ends at n_k and $n_i = \langle R_i, \langle L_i \rangle \rangle$ for $i = 1$ to k. The pseudo-code for the `table_PART_merge` method follows.

```
(00) PART table_PART_merge(PART P_1, PART P_2) {
(01)      L' = L_1 ∘ ··· ∘ L_k;
(02)      if (L ⊕ L' == UNDEFINED) return UNDEFINED;
(03)      L'' = L ⊕ L';
(04)      γ' :  γ'(t) holds iff t ∈ R;
(05)      L'_1 ∘ ··· ∘ L'_k ∘ L_{k+1} = L'', where |L'_i| = |L_i| for i = 1 to k;
(06)      if (|L| ≤ |L'|) return UNDEFINED;
(07)      L = L^1 ∘ L^2, where |L^1| = |L'|;
(08)      if ((is_E_component(L^1) || (|L^1| == 0))
(09)         γ = TRUE;
(10)      else
(11)         γ = γ';
(12)      return substitute(P_1, γ̄_1 ∪ γ, n_1 = ⟨R_1, ⟨L'_1⟩⟩, n_2 = ⟨R_2, ⟨L'_2⟩⟩, ...,
n_k = ⟨R_k, ⟨L'_k⟩, [⟨R, ⟨L_{k+1}⟩⟩]⟩);
(13) }
```

The method `substitute`$(\mathcal{P}, \bar{\gamma}, n_1 = \mathcal{P}_1, \ldots, n_k = \mathcal{P}_k)$ returns the result of substituting the γ-condition in $\mathcal{P}$ with $\bar{\gamma}$ and the node n_i in $\mathcal{P}$ with the tree for $\mathcal{P}_i$,

$i \in [1..k]$. In order for this method to be well defined, it must be the case that for $i \in [1..k]$ either n_i is a leaf node in the description tree of $\mathcal{P}$ or the tree of $\mathcal{P}_i$ 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 in $\mathcal{P}_1$ with table R_1. When the conditions in Line 8 of the pseudo-code is `TRUE`, then it is the case that 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 3.14 and 4.11). Otherwise, 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 with the ordering label L.

The pseudo-code for the method `gamma_PART_merge` follows. 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`.

```
(00) PART P;
(00) set of tables R̄ = ∅;
(00) efficient predicate γ';
(00) gamma_PART_merge(PART P1, PART P2) {
(01)     ⟨R, ⟨L⟩⟩ = P2;
(02)     {A1, ..., Al}, Al+1, ..., Aa = L;
(03)     mergeable=TRUE;
(04)     γ' :  γ'(t) iff t ∈ R;
(05)     n1 = root node of P1 = ⟨R1, ⟨L1⟩⟩;
(06)     γ̄1 = γ-condition of P1;
(07)     if !(R ⊆ R1) return UNDEFINED;
(08)     if (L ⊕ L1 == UNDEFINED) return UNDEFINED;
(09)     L' = L ⊕ L1;
(10)     if ((is_leaf(n1)) || (|L| ≤ |L1|)) {
(11)        if (R == R1)
(12)           γ = TRUE;
(13)        else
(14)           γ = γ';
(15)        P = substitute(P1, γ̄1 ∪ {γ}, n1 = ⟨R1, ⟨L'⟩⟩);
(16)        return P;
(17)     }
(18)     L^1 ∘ L^2 = L' where |L^1| = |L1|;
(19)     P = substitute(P1, γ̄1, n1 = ⟨R1, ⟨L^1⟩⟩);
(20)     for n' ∈ children(n1, P1)
(21)        mergable = mergable ∧ recursive_PART_merge(n', R, L^2, l);
(22)     if ((mergable) && (R ⊆ ⋃_{R'∈R̄} R')) {
(23)        if (R ⊂ ⋃_{R'∈R̄} R'))
```

```
(24)          P = substitute(P_1, γ̄_1∪{γ'}, n_1 = ⟨table(n_1), L(n_1)⟩);
(25)        return P;
(26)      }
(27)      return UNDEFINED;
(28) }
```

```
(00) Boolean recursive_PART_merge(node n, table R, label L, int
l){
(01)      ⟨R_1, ⟨L_1⟩⟩ = n;
(02)      if (R ∩ R_1 == ∅) return TRUE;
(03)      mergable=TRUE;
(04)      if (L ⊕ L_1 == UNDEFINED) return FALSE;
(05)      L' = L ⊕ L_1;
(06)      if ((is_leaf(n)) || (|L| ≤ |L_1|)) {
(07)        P = substitute(P, n = ⟨R_1, ⟨L'⟩⟩);
(08)        R̄ = R̄ ∪ R_1;
(09)        Let K be the rooted path in P that ends at n;
(10)        Let n_1 the root node of P;
(11)        if K is not clustered relative to l
(12)          P = substitute(P, γ(n_1)∪{γ'}, n_1 = ⟨table(n_1), L(n_1)⟩);
(13)      }
(14)      else {
(15)        L' = L^1 ∘ L^2 where |L^1| = |L_1|;
(16)        P = substitute(P_1, n = ⟨R_1, ⟨L^1⟩⟩);
(17)        for n' ∈ children(n)
(18)          mergable = mergable ∧ recursive_PART_merge(n', R, L^2, l);
(19)      }
(20)      return mergable;
(21) }
```

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 end 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 the table for n' is different than the table R, or when 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 described tables or when the clustering property is not satisfied for one of the relevant paths. The ordering labels in $\mathcal{P}_2$ will be recalculated in analogous fashion to the first case.

We will next present a theorem about the correctness and completeness of the presented PART merging algorithm. The theorem relies on the following definitions.

Definition 4.19 (*simplifying an extended index*). Let X be an extended index. X can be simplified by performing one or more of the following rules one or more times.

(1) A table R of a node in X is changed to a table R' such that $R' \subset R$,
(2) a γ condition is removed from X,
(3) an ordering label L of a node of X is simplified by deleting an attribute from its end, and
(4) a leaf node of X is removed.

If an extended index X is simplified into an extended X', then the physical encoding for X will be greater than the physical encoding for X'.

Definition 4.20 (*similar extended indices*). We will say that the extended index X is similar to the extended index X' iff X can be converted into X' by applying one or more times the following procedure: If $\bigcup_{i=1}^{k} R_i = R$, then split a non-root node in X with label $\langle R, \langle L \rangle\rangle$ into k nodes with tables $\{R_i\}_{i=1}^k$, respectively, ordering label L, and same parent and children.

THEOREM 4.21 (PROPERTIES OF THE PART MERGING ALGORITHM). *The following statements are true.*

(*1*) *Suppose that the sSQL queries* $\{Q_i\}_{i=1}^k$ *generate the simple PARTs* $\{\mathcal{P}\}_{i=1}^k$ *and for a suggestion of the second Oracle* `merge_PARTs`$(\mathcal{P}_1, \ldots, \mathcal{P}_k)$ *produces an answer different than* `UNDEFINED`. *Let* $\bar{\mathcal{P}}$ *be the set of PARTs produced by* `merge_PARTs` *under the different possibilities suggest by the second Oracle (i.e, answers different than* `UNDEFINED`*). Then the following statements are true.*

1.1 If $\mathcal{P} \in \bar{\mathcal{P}}$, *then* $X^c(\mathcal{P}) = \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$.

1.2 If the extended index X *can be used to efficiently answer the queries* $\{Q_i\}_{i=1}^k$ *and there does not exist an extended index that is created by simplifying* X *and that can efficiently answer the enumerated queries, then* X *is similar to some extended index* X' *for which there exists* $\mathcal{P} \in \bar{\mathcal{P}}$ *such that* $X' \in X(P)$.

(*2*) *If the* `merge_PARTs` *method returns* `UNDEFINED` *for every possibility suggested by second Oracle when trying to merge the PARTs* $\{\mathcal{P}\}_{i=1}^k$ *that were generated by the queries* $\{Q_i\}_{i=1}^k$, *then* $\bigcap_{i=1}^{k} X^c(\mathcal{P}_i) = \emptyset$.

Proof:

1.1 $\Leftarrow$ We will first prove that if $\mathcal{P} \in \bar{\mathcal{P}}$ and $X \in \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$, then $X \in X^c(\mathcal{P})$. This means that $X \in X^c(\mathcal{P}_i)$ for all $i \in [1..k]$. Let $\bar{X}_i$ be the set of extended indices that $\mathcal{P}_i$ represents, that is, $\bar{X}_i = X(\mathcal{P}_i)$ for $i = 1$ to k. Therefore, it must be the case that $X \in$ `cover`$(\bar{X}_i)$ for $i = 1$ to k.

Consider the PART $\mathcal{P}_i$ for some $i \in [1..k]$. Note that $\mathcal{P}_i$ will have the form $\langle R, \langle \{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a \rangle\rangle$. Since $X \in$ `cover`$(\bar{X}_i)$, there must exist extended

indices $X_i \in \bar{X}_i$ ($i \in [1 \ldots k]$) such that $Q(X_i) \subseteq Q(X)$ for $i \in [1 \ldots k]$. Note that $X_i = \langle R, \langle A'_1, \ldots, A'_l, A_{l+1}, \ldots, A_a \rangle\rangle$, where $\langle A'_1, \ldots, A'_l \rangle$ is a permutation of the attributes $\{A_i\}_{i=1}^{l}$. Let $\bar{X}$ be the set of extended indices in $X(\mathcal{P})$ that respect the permutation decisions of $\{X_i\}_{i=1}^{k}$. In other words, for $i \in [1..k]$, if the attribute A comes before the attribute B in in X_i, then the attribute A comes before the attribute B in all rooted paths of an extended indices in $\bar{X}$. Note that the `table_PART_merge` and `gamma_PART_merge` methods are such that $P_i = \langle R, \langle \{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a \rangle\rangle$ maps to one or more rooted paths in $\mathcal{P}$, which we will denote as $\bar{K}_i$, that have the following properties.

(1) If the end nodes of the paths $\bar{K}_i$ are the tables of $\bar{n}_i$, then $R \subseteq \bigcup_{R' \in \gamma(table(n))}^{n \in \bar{n}_i, \gamma \in \gamma^e(\mathcal{P})} R'$.
When we have strict containment, the appropriate γ-condition is added to $\mathcal{P}$.

(2) The concatenation of the attributes of the ordering labels of each path $\bar{K}_i$ starts with a permutation of the attributes $\{A_i\}_{i=1}^{l}$ followed by the attributes $\langle A_{l+1}, \ldots, A_a \rangle$ in this order.

(3) Each path in the set $\bar{K}_i$ is clustered relative to l or the appropriate γ-condition is added to $\mathcal{P}$.

But then Theorem 3.14 implies that, for any $X' \in \bar{X}$, $Q(X') \subseteq Q(X)$, and therefore $X \in$ `cover`$(\bar{X})$, which implies that $X \in X^c(\mathcal{P})$.

$\Rightarrow$ Suppose that $X \in X^c(\mathcal{P})$. Let i be an arbitrary integer in the rage $[1 \ldots i]$. We will prove that $X \in X^c(\mathcal{P}_i)$. Let X_i be such that $X_i \in X(\mathcal{P}_i)$ and the ordering label of the single node of X_i preserves the attribute ordering of the concatenation of the ordering labels along the rooted paths of X. Note that the `table_PART_merge` and `gamma_PART_merge` methods are such that $P_i = \langle R, \langle \{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a \rangle\rangle$ maps to one or more rooted paths in $\mathcal{P}$, which we will denote as $\bar{K}_i$, that have the following properties.

(1) If the end nodes of the paths $\bar{K}_i$ are the tables $\bar{n}_i$, then $R \subseteq \bigcup_{R' \in \gamma(table(n))}^{n \in \bar{n}_i, \gamma \in \gamma^e(\mathcal{P})} R'$.
When we have strict containment, $\mathcal{P}$ contains the appropriate γ-condition.

(2) The concatenation of the attributes of the ordering labels of each path $\bar{K}_i$ starts with a permutation of the attributes $\{A_i\}_{i=1}^{l}$ followed by the attributes $\langle A_{l+1}, \ldots, A_a \rangle$ in this order.

(3) Each path in the set $\bar{K}_i$ is clustered relative to l or an appropriate γ-condition is present.

But then Theorem 3.14 implies that $Q(X_i) \subseteq Q(X)$ and therefore $X \in X^c(\mathcal{P}_i)$, which is what we wanted to prove.

1.2 Suppose that the extended index X can be used to efficiently answer the queries $\{Q_i\}_{i=1}^{k}$ and that X cannot be simplified further and still be used to efficiently answer these queries. Suppose that applying the procedure from Table V translates Q_i into the PART $\mathcal{P}_i = \langle R_i, A_1, \ldots, A_l, A_{l+1}, \ldots, A_a \rangle$ for $i \in [1 \ldots k]$. Then, from Theorem 3.14 it follows that X must contain the nodes $\bar{n}_i$ that satisfy the following properties.

(1) If $\bar{K}_i$ is the rooted path that ends at n_i in X, then $R \subseteq \bigcup_{R' \in \gamma(table(n))}^{n \in \bar{n}_i, \gamma \in \gamma^e(X)} R'$. When we have strict containment, then X contains the appropriate γ-condition.
(2) The concatenation of the attributes of the ordering labels of each path $\bar{K}_i$ starts with a permutation of the attributes $\{A_i\}_{i=1}^{l}$ followed by the attributes $\langle A_{l+1}, \ldots, A_a \rangle$ in this order.
(3) Each path in the set $\bar{K}_i$ is clustered relative to l or appropriate γ-condition is present.

Since X is the smallest possible extended index that has the capabilities to answer the queries $\{Q_i\}_{i=1}^{k}$, X will contain only the nodes $\bigcup_{i=1}^{k} \bar{n}_i$. Suppose that X contains a node n with label L and table R that has the following properties:

a) there does not exist a query in the set $\{Q_i\}_{i=1}^{k}$ that references the table R, and
b) R is the union of several tables that are referenced in the queries $\{Q_i\}_{i=1}^{k}$.

If n is not the root node of X, then n can be split into several nodes. Let X' be the extended index that is similar to X and that does not contain non-root nodes with the described property. Therefore, only the table of the root node of X' will be the union of several tables that are referenced in the queries $\{Q_i\}_{i=1}^{k}$. The other nodes in X' will be based on either:

a) tables that are referenced in the set of queries $\{Q_i\}_{i=1}^{k}$ or
b) tables that can be computed as a boolean combination of tables that reference queries in the set $\{Q_i\}_{i=1}^{k}$ except for the operation that takes the union of all tables and the operation that takes the tuples that do not belong to the tables.

Consider a node in X' that is based on a table of the second kind. This node can be split into several nodes, where each of the created nodes references only tables of the form $R_1^{j_1}$ `interesection` ... `interesection` $R_r^{j_r}$, where $\{R_l\}_{l=1}^{r}$ are some tables referenced in $\{Q_i\}_{i=1}^{k}$, j_i is 1 ore -1 for $i \in [1 \ldots r]$ and $\{j_i\}_{i=1}^{r}$ are not all -1. Let X'' be an extended index similar to X' in which the nodes are split in the described way. We have shown that the tables of the nodes of X'' will be generated under at least one possibility suggested by the second Oracle. Let's fix one such possibility and suppose that $\mathcal{P}$ is the PART produced by it.

Note that if a table R is created by the `table_PART_merge` method, then it must be the case that the ordering label of the node for R will not be empty. Without loss of generality, suppose that $\mathcal{P}$ cannot be rewritten to an equivalent PART $\mathcal{P}'$ in which the table of the node for R will be empty or not present. If this could not be done, then another possibility suggested by the second Oracle will generate the described PART. Recall that our algorithm for creating $\mathcal{P}$ fills the ordering labels of $\mathcal{P}$ in a breath first traversal of the tree and for every PART in the set $\{\mathcal{P}_i\}_{i=1}^{k}$ that has its first bit marked a node is created and for the subset of PARTs in the set $\{\mathcal{P}_i\}_{i=1}^{k}$ that have their second bit marked a set of nodes is created. Moreover, the algorithm succeeds exactly when the ordering labels of the enumerated PARTs extend to the inserted node. The ordering labels generated by the PARTs that have their bit marked can be added last because they are defined on the table of

the node of the root of the tree. However, since X'' cannot be simplified further, the ordering labels in X'' will be the same as the ordering label of $\mathcal{P}$, that is, we have shown that $X'' \in X(\mathcal{P})$.

2. Suppose that $\bigcap_{i=1}^{k} X^c(\mathcal{P}_i) \neq \emptyset$. Therefore, there exists an extended index X s.t. $X \in \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$. We will show that under at least one permutation suggested by the second Oracle `merge_PARTS` produces a PART $\mathcal{P}$ such that $X \in X^c(\mathcal{P})$. Without loss of generality, assume that X cannot be simplified further and consider the tables of the nodes of the description tree of X. If X does not reference a set of tables that could be generated by a possibility suggested by the second Oracle, then X can be rewritten into a similar extended index X' that references a set of tables suggested by the second Oracle, where this procedure will involve several node breakups. Therefore, suppose that X references the tables suggested by one of the possibilities of the second Oracle. Under this possibility the `merge_PARTs` method will generate a PART $\mathcal{P}$ such that $X \in X(\mathcal{P})$, that is, under at least one possibility suggested by the second Oracle `merge_PARTS` will not generate `UNDEFINED`. ■

4.4.1 *Approximate Algorithms for merging PARTs.* In the previous section we showed an exponential algorithm for merging PARTs and therefore proved that PART merging is NP-Complete (Theorem 4.18 states that PART merging is NP-Hard). A drawback of the presented algorithm is that it takes exponential time for any input. In this section we present an approximate algorithm for merging PARTs. This algorithm does not always produce the smallest possible set of PARTs, but has the advantage that it has worst-case polynomial running time.

Our approximate algorithm first clusters the input PARTs relative to the tables they reference. We then cluster the PARTs in each cluster, where the PARTs in each new cluster have the property that their labels can be merged using the "$\oplus$" procedure. The second clustering can be done in polynomial time using a greedy algorithm. Our next step is to build a graph that has a node for each cluster. In particular, each node in the graph will contain a single PART that is the merge of the PARTs in the corresponding cluster. The graph will contain two types of edges: directed and undirected. There will be a directed edge from the PART $\mathcal{P}_1$ to the PART $\mathcal{P}_2$ iff $\texttt{table}(\mathcal{P}_2) \subseteq_{\texttt{ID}} \texttt{table}(\mathcal{P}_1)$. There will be an undirected edge between $\mathcal{P}_1$ and $\mathcal{P}_2$ iff $\texttt{table}(\mathcal{P}_1) \cap \texttt{table}(\mathcal{P}_2) \neq \emptyset$. We will then simplify the graph by removing directed edges from a PART $\mathcal{P}_1$ to a PART $\mathcal{P}_3$ whenever there is a directed edge from $\mathcal{P}_1$ to some PART $\mathcal{P}_2$ and from $\mathcal{P}_2$ to $\mathcal{P}_3$. Note that the so constructed graph will be acyclic relative to the directed edges.

We will refer to nodes in the graph with no incoming directed edges as root nodes. For a node n, its child nodes are the nodes for which there is a directed edge from n to them. For the first root node, relative to some order, we will exam all of the node's children and merge as much of them as possible with the node using the `table_PART_merge` procedure. In particular, if the node contains children with undirected edges between them and ordering labels of the children that can be merged using the $\oplus$ operation, then the method splits the nodes using the `intersections` method when this will help for the nodes to be merged with the root node. The procedure is repeated recursively for the children of the children's

nodes and so on until nodes with no outgoing arcs are reached. After the described procedure is applied for the first root note, the nodes that contributed to the created PART are removed from the graph and the edges between the nodes of the graph are recalculated using the same procedure as the one described in the previous paragraph. Then the described method is applied on one of the root nodes in the new graph and so on. After all nodes have been removed from the graph, we examine all created PARTs that were created from the single original PART. We then try to add them to the other PARTs using the `gamma_PART_merge`.

The presented algorithm runs in polynomial time, but it does not guarantee that the it will produce the optimal solution. It is just one example of how an approximate algorithm for the PART merging problem can be created. Other avenues for future research that we believe are worth exploring include designing approximate algorithms for the PART merging problem using dynamic programming and iterative dynamic programming ([Kossmann and Stocker 2000]).

5. EXPERIMENTAL EVALUATION

We conducted experimental results using the example database schema shown in Figure 3. The code was written in Java and can be found at [**?**]. We chose to use this approach rather than use a recognized benchmark, such as TPC-C ([**?**]), because our system is an index merger and the quality of the solution relies on the sSQL queries generated by an external advisor.

In the beginning, the database was populated with 130650 customer, 3146 managers, 6500 workers, and 10400 trainees. The data was uniform, i.e., there were 676 distinct customer names and 201 different balances from (from -100 to +100) for each customer and so on. We next created a workload that consisted of queries and updates, where the update ratio (i.e., the ratio of primitive updates to the number of primitive updates plus the number of retrieved objects) was set. We examined two cases: the *naive* approach where an index was created for each query and the *RECS-DB* approach where the index merging algorithm was applied. The result are shown in Figure 4.

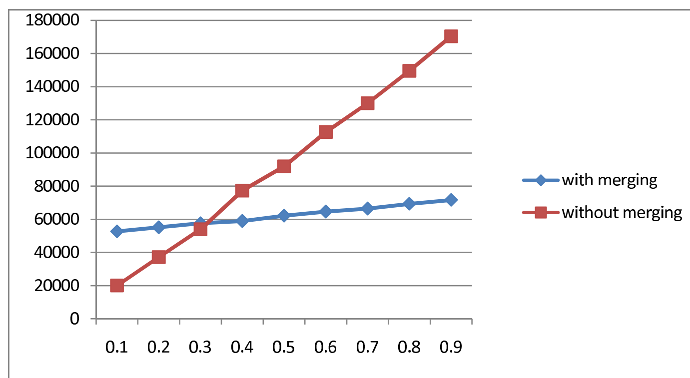

Fig. 4. Number of accessed nodes for different update ratios

6. CONCLUSION

We presented a way for creating compact index structures that save space and speedup updates. We showed theoretically and validated experimentally the benefits of these data structures.

REFERENCES

ADELSON-VELSKII, G. M. AND LANDIS, E. M. 1962. An Algorithm for the Organization of Information. *Soviet Math. Doklady 3*, 1259–1263.

AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. 2000. Automated Selection of Materialized Views and Indexes for SQL Databases. *VLDB*, 496–505.

ANDERSSON, A. 1993. Balanced search trees made simple. *Workshop on Algorithms and Data Structures*, 60–71.

BANCILHON, F. AND FERRAN, G. 1994. ODMG-93: The Object Database Standard. *IEEE Data Eng. Bull. 17,* 4, 3–14.

BAYER AND MCCREIGHT. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica 1,* 3.

BELL, E. T. 1934. Exponential Numbers. *Amer. Math. Monthly 41*, 411–419.

BRUNO, N. AND CHAUDHURI, S. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. *SIGMOD 2005*, 227–238.

CHAUDHURI AND NARASAYYA. 1999. Index Merging. *ICDE*, 296–303.

FINKELSTEIN, S., SCHKOLNICK, M., AND TIBERIO, P. 1988. Physical Database Design for Relational Databases. *ACM Transaction on Database Systems 13,* 1 (March), 91–128.

GAREY, M. AND JOHNSON, D. 1972. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H.Freeman and Company.

KOSSMANN, D. AND STOCKER, K. 2000. Iterative Dynamic Programming: A new Class of Query Optimization Algorithms. *ACM TODS*, 43–82.

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.