Programming Embedded Computing Systems using Static Embedded SQL

Lubomir Stanchev
Computer Science Department
Indiana University-Purdue University Fort Wayne, Indiana, U.S.A.
and
Grant Weddell
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada

The information technology boom in the last decade has made embedded computing systems increasingly common. This has fueled the need for increased automation for large parts of the software development process of such systems. However, such automation must account for the fact that embedded software may require guarantees on response times and can have limited memory available for storing code and data. In this paper, we show how parts of the software can be written in a declarative programming language such as SQL. This is challenging because (1) SQL is a declarative language that abstracts any consideration of execution time, (2) most commercial SQL engines have a large footprint that cannot be stored on an embedded device, and (3) most SQL operations can be executed in satisfactory time only when potentially large amounts of additional storage is available for auxiliary structures, such as indices and materialized views. The paper shows how these challenges can be addressed by using the following strategies.

(1) A subset of SQL is defined, called μSQL, with the property that operations in this subset can always be supported in time logarithmic in the size of the underlying database. Consequently, programmers have immediate guarantees on worst case response times for control data access and update.
(2) All data operations are pre-compiled in order to avoid expensive query parsing and optimization during run-time and to avoid the need for storing large components of a general purpose database engine on the embedded device itself.
(3) Only data required for efficient execution of the predefined operations is stored on the embedded system.
(4) All search structures are implemented using a novel physical design that reduces the need for storing duplicate data.

In summary, the benefits of our approach to the programming of embedded computing systems are two-fold. First, there is no need for programmers to consider the problem of mapping logical data design to concrete data structures. And second, programmers specify their requirements for control data access and update in SQL. We anticipate that the resulting higher level code will therefore be faster to create and easier to understand, test and maintain.

Categories and Subject Descriptors: E2 [**Data**]: Data Storage Representations; D3 [**Software**]: Programming Languages

---

Indiana University - Purdue University Fort Wayne Computer Science Techincal Report CS2008-1

1. INTRODUCTION

We propose a new approach for developing *Realtime Embedded Control Programs* (RECPs) that uses the SQL API currently supported by relational database technology. Consider the typical current architecture of a RECP shown in Figure 1 in which a set of modules that contain control code interact with the control data of the system using a procedural language, such as C or Assembler. With current practice, the onus is on the software developer to design the data structures that encode the control data and the code that accesses this data. This is not a trivial problem for the following reasons:

(1) the data structures must be carefully designed in a way that utilizes the limited storage space, and
(2) the data access operations must be efficient in order to meet the realtime requirements of the system.

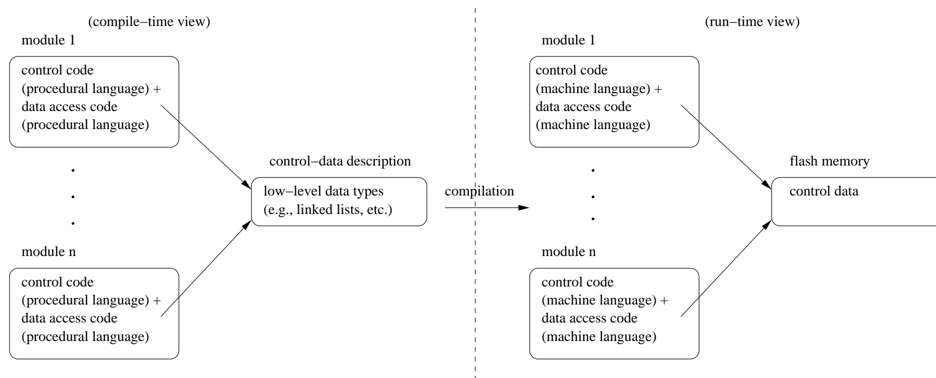

Fig. 1. The typical architecture of a realtime embedded control program.

As an alternative, consider the architecture depicted in Figure 2. In this case, the user can specify the data access code using a declarative language, such as SQL queries and updates, and the description of the control data using a database schema language. As a result, the developers of a RECP can concentrate on the logic of the data, while the details of how the data is stored and manipulated are abstracted.

The control code in Figure 2 can continue to be compiled by an existing language compiler. However, a new approach is needed for compiling data access code that is specified in a declarative language. In the paper, we present an algorithm for performing this compilation. Since the algorithm is part of a system we are currently developing called RECS-DB (short for *Realtime Embedded Control System DataBase*), we will refer to the algorithm as the *RECS-DB algorithm.* The input and output of the algorithm are depicted in Figure 3.

Note that the output of our algorithm becomes the source comprising the compile-time view for the traditional RECP architecture shown in Figure 1. That is, the RECS-DB algorithm can be used to translate the compile-time view from Figure

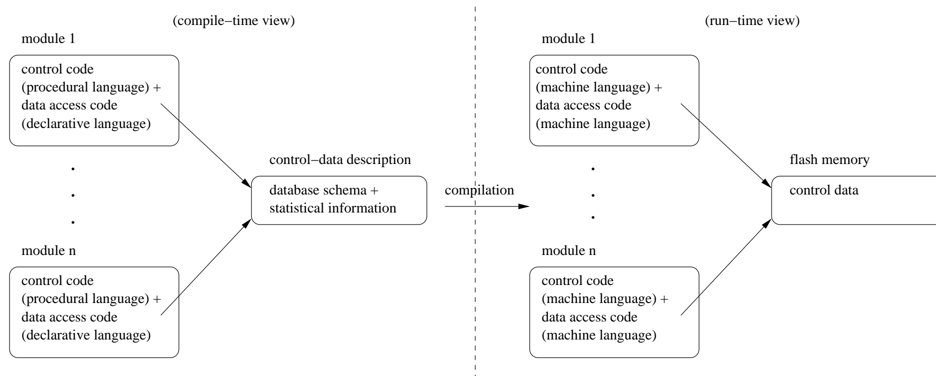

Fig. 2. The new architecture of a realtime embedded control program.

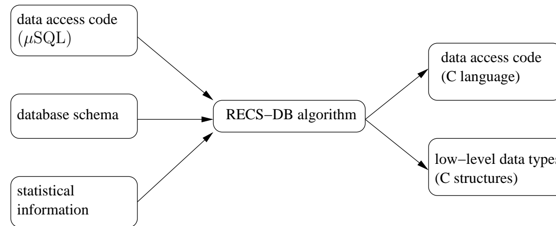

Fig. 3. Input and output of the RECS-DB algorithm.

2 to the compile-time view from Figure 1. The output of the algorithm is peculiar for the particular RECS, that is, the data access code and low-level structures are designed specifically for the SQL operations needed by a particular given RECP. In addition, the data access code generated by the algorithm is efficient in the sense that the computation of a tuple from a query result and the modification of a tuple from a base table are both performed in roughly logarithmic time relative to the size of the database.

SQL is chosen because it is the *de facto* standard for accessing relational databases. Since not every SQL operation can be executed efficiently ([Fredman 1981]), the input is restricted to a dialect named μSQL that has this property. Operations that are not part of this dialect need to be manually broken into operations that are. Fortunately, the μSQL dialect is quite expressive. For example, we were able to directly express 92% of the TPC-C workload using μSQL over efficiently maintainable *materialized views* (MVs).

The logarithmic time-bound for data operations is chosen because it corresponds to the time it takes to probe a tree index. However, data structures that allow record retrieval in sub-logarithmic time, such as *y-fast trees* ([Willard 1983]) and *interpolation search trees* ([K.Mehlhorn and Tsakalidis 1985]), would also be possibilities. Also, the constant in front of the logarithmic time-bound is small and depends on the size of the database schema description, the size of the description of the input queries, and some parameters of the chosen index tree implementation (such as the number of records per node).

The pre-compiled low-level code answers each μSQL query using a sequence of nested loop joins. In order to avoid visiting tuples that do not contribute to the query result, and subsequently failing to adhere to the logarithmic worst-case time bound, the indices and hash structures over which each nested loop join iterates are on newly introduced MVs that contain data guaranteed to contribute to the query result.

The data structures for the control data extend traditional tree indices and hash structures in a way that saves space. This is achieved by allowing data structures that share data to be merged into a single data structure in a way that eliminates unnecessary data replication without sacrificing performance. The merging algorithm tries to produce the data structures that are expected to take the least space, where the statistical information, which is also given as input, is used to approximate the expected size of a data structure.

1.1 Overview of the RECS-DB Algorithm

The seven steps of the RECS-DB algorithm are illustrated in Figure 4. Arrows in the figure are used to denote data flow and edge labels are used to denote the step in which the data flow occurs. The label "+" indicates where results are added to an existing set.

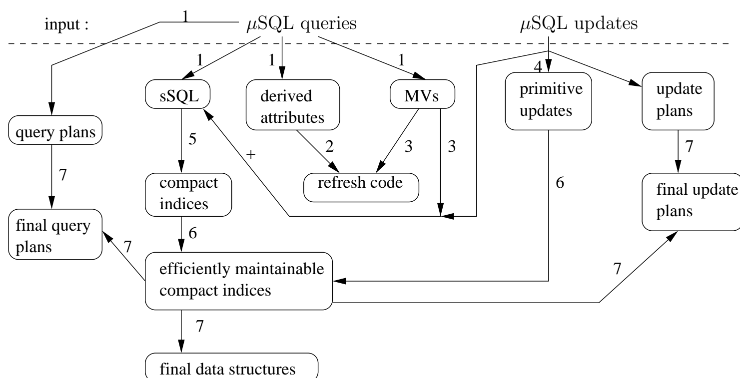

Fig. 4. Steps of the RECS-DB algorithm

The requirement for each step is as follows.

(1) Input queries expressed in μSQL are decomposed into simper formulations in a dialect called sSQL. μSQL is the general dialect for SQL queries that are allowed as input, whereas sSQL admits queries on single tables only. For each μSQL query, the step produces derived attributes, MVs, and a query plan that references newly created sSQL queries. Derived attributes are introduced in order to simplify the syntax of the query description of the generated MVs.

(2) Refresh code for derived attributes is generated.

(3) Refresh code for the MVs is generated. This process can entail the generation of additional sSQL queries.

(4) Updates expressed in μSQL are replaced by low-level code that references *primitive updates* and additional sSQL queries. Informally, a primitive update is an insertion, deletion, or modification that references a single tuple of a single base table.

(5) A single *compact index* for each query is generated. A compact index is a novel data structure introduced in this paper. Compact indices are beneficial because they reduce the need for storing duplicate data.

(6) The compact indices generated in Step 5 are modified in a way that ensures that any primitive update can be performed on any relevant compact index in logarithmic time.

(7) The final step of the algorithm merges compact indices and rewrites the query and update plans to reference the newly introduced data structures. A single merge substitutes a set of indices with a single compact index of smaller size that can efficiently answer the queries supported by the original indices. Thus, index merging is an important optimization for reducing the encoding size of the control data for a RECP.

1.2 Motivating Example

In this subsection we illustrate the behavior of the RECS-DB algorithm on a small example.

1.2.1 *Input.* Consider the scenario where an embedded control device needs to scan incoming network packets and perform operations based on the answers to certain queries. In particular, suppose that the data conforms to the schema shown in Figure 5, where we have used ellipses around *base table* names and round rectangles around attribute types. The table `PACKET` contains information about the last five minutes of packet data, that is, it can be also perceived to be a *sliding window* over streaming data (see for example [Golab and Özsu 2005]). The table `COMPUTER` contains information about the possible computers that are packets destination.

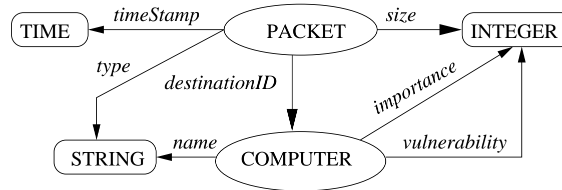

Fig. 5. Example schema

We assume that every table has the system attribute `ID`, which is a global tuple identifier for base tables. Also, note that in our example we assume that there is a foreign key dependency for the attribute *destinationID* of the table `PACKET` that references the attribute `ID` of the table `COMPUTER`. The later implies that for every tuple $t \in$ `PACKET`, there exists a tuple $t' \in$ `COMPUTER` for which $t.distinationID = t'.$`ID`.

Suppose we are given queries `Q1` and `Q2` and updates `U1`, `U2`, and `U3` defined in Table I, where $:P$ is used to denote a query parameter. Query `Q1` asks for

Table I. An example workload

(*name*)	(*query*)
Q1	select * from PACKET as p, COMPUTER as c where $p.destinationID$= c.ID and $c.vulnerability$ > :P and $p.type$ = "TCP/IP" order by $c.importance$ asc
Q2	select * from PACKET as p, COMPUTER as c where $p.destinationID$= c.ID order by $c.vulnerability$ asc, $c.name$ asc
U1	delete from PACKET as p where $p.timpeStamp \leq :P$
U2	insert into PACKET ($timeStamp, size, destinationID, type$) values $\{:P_1,:P_2,:P_3,:P_4\}$
U3	insert into COMPUTER ($name, importance, vulnerability$) values $\{:P_1,:P_2,:P_3\}$

Table II. New derived attributes for the table COMPUTER

(*name*)	(*value for a tuple* $t \in$ COMPUTER)
packets	select count(*) from COMPUTER as c, PACKET as p where $p.destinationID = c$.ID and c.ID = t.ID
tcpPackets	select count(*) from COMPUTER as c, PACKET as p where $p.destinationID = c$.ID and c.ID = t.ID and $p.type$ = 'TCP/IP'

Table III. Intermediate views

(*name*)	(*query*)
V_TCP_PACKET	select * from PACKET as p where $p.type$ = 'TCP/IP'
V_TCP_COMPUTER	select * from COMPUTER as c where $c.tcpPackets > 0$
V_COMPUTER	select * from COMPUTER as c where $c.packets > 0$

recent TCP/IP packets destined for computers with vulnerability greater than a specified threshold, where the result should be ordered by the importance of the destination computer. Query Q2 asks for the recent packets and their destination computer, where the result should be ordered by the vulnerability and name of the destination computer. Updates U1 and U2 represent the expiration of a tuple from and the insertion of a tuple in the table PACKET. Update U3 represents the insertion of a tuple in the table COMPUTER.

1.2.2 *Step 1.* In this step we will break the queries Q1 and Q2 into simple queries that reference single tables (i.e., sSQL queries). In order to do so, RECS-DB first adds the derived attributes *packets* and *tcpPackets* to the table COMPUTER. The attribute *packet* stores the number of tuples from the table PACKET a tuple from the table COMPUTER joins with, while the attribute *tcpPackets* stores the number of TCP/IP packets from the table PACKET a tuple from the table COMPUTER joins with (see Table II). Next, RECS-DB creates the MVs shown in Table III. The MV V_TCP_PACKET contains only the TCP/IP packets from the table PACKET. The MV V_TCP_COMPUTER contains those computers for which at least one TCP/IP packet has been received in the last five minutes, while V_COMPUTER contains those computers for which a packet was received in the last five minutes.

The two derived attributes for the table COMPUTER are introduced in order to allow simpler syntax for the underlined queries of the defined MVs. The MVs V_PACKET and V_COMPUTER are useful because they contain exactly the data from the tables PACKET and COMPUTER, respectively, that is needed for answering Q1. Similarly, the MVs V_COMPUTER and the table PACKET store exactly the data that is needed for answering Q2. In particular, the query Q1(:P) can be efficiently answered using the following query plan.

```
for t2 ∈ select *
        from V_TCP_COMPUTER as c
        where c.vulnerability > :P
        order by c.importance asc
   for t1 ∈ select *
           from V_TCP_PACKET as p
           where p.destinationID = :t2.ID
          send join(t1, t2, t2.destinationID = t1.ID);
```

Similarly, Q2 can be answered using the following query plan.

```
for t2 ∈ select *
        from V_COMPUTER as c
        order by vulnerability asc, c.name asc
   for t1 ∈ select *
           from PACKET as p
           where p.destinationID = :t2.ID
          send join(t1, t2, t2.destinationID = t1.ID);
```

Note that we have used join to denote the result of joining the tuples specified as the first two parameters relative to the condition specified as the third parameter and send to denote the generation of a resulting tuple. Both query plans have no "false drops", that is, every tuple that is produced in an outer loop matches with at least one tuple in the corresponding inner loop. Therefore, if an index is used to answer each of the simple queries, then each tuple from the query result will be generated in time proportional to logarithm of the size of the database. Conversely, note that a query plan that scans an index on the table COMPUTER as the outer operation of the join will not be efficient for answering Q1. In particular, it may be the case that there is no computer for which a TCP/IP packet is destined

Table IV. Two example simple queries

(*name*)	(*query*)
Q5	select * from V_TCP_COMPUTER as c where $c.vulnerability > :P$ order by $c.importance$ asc
Q6	select * from V_COMPUTER as c order by $vulnerability$ asc, $c.name$ asc

Table V. Indices for efficiently answering queries Q_5 and Q_6

(*name*)	(*keys* (k))	(*values*)
$X_1^\dagger$	select distinct $c.vulnerability$ from V_COMPUTERS as c order by $v.vulnerability$ asc	$\&X_3(k)^\ddagger$ and $\&X_2(k)$ if $c \in V_{\texttt{TCP_COMPUTER}}$
$X_2(P)$	select distinct $c.importance$ from V_TCP_COMPUTER as c where $c.vulerability = :P$ order by $c.importance$ asc	$\&W_1(P, k)$
$X_3(P)$	select distinct $c.name$ from V_COMPUTER as c where $c.vulnerability = :P$ order by $c.name$ asc	$\&W_2(P, k)$

† The nodes in X_1 contain an extra marking bit
‡ $\&X_2(k)$ denotes the address of the index $X_2(k)$

Table VI. Linked lists for efficiently answering queries Q_5 and Q_6

(*name*)	(*elements*)
$W_1(:P_1,:P_2)$	select * from V_TCP_COMPUTER as c where $c.vulnerability$ $:P_1$ and $c.importance = :P_2$
$W_2(:P_1,:P_2)$	select * from V_COMPUPTER as c where $c.vulnerability$ $:P_1$ and $c.name = :P_2$

and the query result will be empty. However, the query plan will still scan the whole table COMPUTER, which will result in linear, rather than logarithmic, performance.

1.2.3 *Step 5.* This step creates an index for each simple query from the current set. For example, an index on the MV V_TCP_COMPUTER and attributes *vulnerability* and *importance* can be used to efficiently answer Q5 from Table IV.

1.2.4 *Step 7.* The regular indices produced in Step 5 can be used to efficiently answer the set of simple SQL queries. However, we go a step further by showing how the size of the indices can be reduced in size by reducing the amount of duplicate information. Next, we will show the compressed data structures for the queries from Table IV, where the data structures for the remaining simple queries can be constructed in an analogous fashion.

The algorithm will create the three index structures shown in Table V and the two link list structures shown in Table VI. Note that we do not concretize the exact physical design for an index structure. However, we assume that the elements are

ordered in a search tree, where each node of the tree can contain one or more elements. This means that the index can be an AVL tree [Adelson-Velskii and Landis 1962], a B tree [Wirth 1972; Salzberg 1988; Smith and Barnes 1987], a CSB^+ tree [Rao and Ross 2000], a T tree [Lehman and Carey 1986], to name a few possibilities. We therefore describe an index an by its elements, the order of the elements, and the additional information that is stored at each node of the tree index.

Index X_1 contains the distinct values of the attribute *vulnerability* in the MV `V_COMPUTER` sorted in ascending order. Each node in the index also stores an extra marking bit that is set exactly when the node or one of its descendants contains the vulnerability of a computer for which a TCP/IP packet is destined. Marking bits are described in detail in [Stanchev 2005]. In this example, the added marking bit allows one to search for a *vulnerability* from the table `V_TCP_COMPUTER`. This can be done in logarithmic time because subtrees with unmarked root nodes can be pruned out. In general, marking bits allow for the efficient search in different subsets of the indexed elements, where a marking bit needs to be defined for each subset. Marking trees can also be updated efficiently (see [Stanchev 2005]) because after every tuple insertion, deletion, or modification involves changing the value for the marking bits along a single path of the tree.

For a value with vulnerability k, the index X_1 also contains a pointer to the index $X_3(k)$ and, in addition, a pointer to the index $X_2(k)$ when there exists a computer with vulnerability k for which a TCP/IP packet is destined. (Note that we use $X(k)$ to denote the index in the set of indices X for which the parameter is equal to k.) The index $X_2(P)$ contains the distinct values of the attribute *importance* for all computers with vulnerability P for which a TCP/IP packet is destined. In addition, for a value P_2 for importance, the index $X_2(P_1)$ contains a pointer to the doubly linked list $W_1(P_1, P_2)$ that contains the tuples for computers with *vulnerability* P_1 and *importance* P_2 for which TCP/IP packets are destined. Similarly, the index $X_3(P)$ contains the distinct values of the attribute *name* for all computers with vulnerability P. For a value with *name* P_2 the index $X_3(P_1)$ contains a pointer to the doubly linked list $W_2(P_1, P_2)$ that contains the tuples for computers with *vulnerability* P_1 and *name* P_2.

Note that W_1 and W_2 are two sets of linked lists that contain overlapping tuples form `V_COMPUTER`. In the spirit of avoiding data replication, each tuple will be stored in a record only once and it will have two or one pairs of forward/backward pointers depending on whether it is in `V_TCP_COMPUTER` or not, respectively, where the first tuple will not have a backward pointer and the last tuple will not have a forward pointer. Since different records can have different number of fields, a field management technique, such as the one proposed in [Zibin and Gil 2002] or [Zibin and Gill 2003], needs to be applied.

In order to understand how the data structures from Figures V and VI can be used, consider an instance of query Q_5 with value five for the parameter. The access plan for executing the query will first search for the left most key value in X_1 with value equal or greater than five that is in a marked node (remember that a marked node means that the node or one of its descendants contains a vulnerability from `V_TCP_COMPUTER`). Then the algorithm will search for the next node to the right

that is marked and so on. Note that node marking is done in a way that guarantees that if a node is not marked, then nighter the node nor its children will contain a *vulnerability* that will contribute to the query result and therefore all subtrees with unmarked root nodes can be pruned out. For each found node, the algorithm needs to scan the key values inside the node and follow the pointer to a X_2 index when such exists. The elements of each visited X_2 index will be scanned in order and for each element the pointed by it linked list of W_1 will be scanned to retrieve the query result. Note that the query plan that was described is efficient because it performs a constant number of index scans and record reeds in order to return each tuple from the query result.

1.3 Related Research

The idea of applying database technology in the development of RECPs is not new. Consider for example the eXtremeDB software ([eXtremeDB]), a main-memory database with realtime guarantees. The system allows the user to specify the database at the physical level, that is what lists, indices, hash tables, and so on are to be created. Access to the data is accomplished by using a native language. Although such a system is useful, the onus is on the developer to define the physical design of the database. Another shortfall of the system is the lack of SQL support.

There are several implementations of stand-alone main-memory database systems (e.g., MonetDB [MonetDB] and TimesTen [TimesTen]). However, these systems do not provide realtime guarantees.

To summarize, the problem addressed by RECS-DB is relatively unexplored. One exception is the paper [Weddell 1989]. It explores the problem of creating indices that allow the efficient execution of a predefined workload of queries. However, the paper considers only simple partial-match queries in the absence of updates. Two other papers ([Stanchev and Weddell 2002; 2003]) touch on the problem, but they do not consider the rich set of admissible SQL queries discussed here.

Note that the RECS-DB algorithm solves a problem that differs from the traditional task of automating the physical design for database system. The later problem is usually described as finding the best possible set of indices and/or materialized views for a given workload, where a workload is abstracted as a set of queries and updates together with their frequencies. In commercial systems, like IBM DB2 UDB [Valentin et al. 2000] and Microsoft SQL Server [Agrawal et al. 2000], the problem is formulated as an optimization problem in which the execution time of the queries and updates is minimized, possibly subject to a fixed storage overhead. This is accomplished by enumerating configurations of indices and materialized views that can be potentially useful for performing the data operations in the workload. A query plan based on a proposed configuration can be evaluated using a "what-if" query optimizer that approximates the cost of the plan. For RECPs, however, realtime requirements can be specified on the execution time of queries and updates. Thus, overall requirements shift from a previous need to ensure efficient use of store to a primary need of ensuring query and update efficiency with a secondary need to merge data structures in order to save space.

1.4 Paper Outline

In the next section we present definitions relevant to the RECS-DB algorithm. In Section 3, we describe the novel compact index data structures that allow us to save space. In Section 4, we present in detail the algorithm from Figure 4, where the different subsections correspond to the different steps of the algorithm. Section 5 concludes with our summary comments and suggestions for future research.

2. DEFINITIONS

2.1 Database Schema

The RECS-DB procedure takes as input a database schema that consists of only base tables, where MVs and derived attributes can be added to it as part of the algorithm. We will use T to denote a base table, V to denote a MV, and R to denote a table (i.e., a base table or a MV). Every table has the system attribute `ID` that serves as a global tuple identifier for base tables. It is similar to the global object identifier in the ODMG model (see [Bancilhon and Ferran 1994]). The differences are that: (1) the `ID` attribute of a tuple uniquely determines the physical address of the tuple's record within the data structure in which it is stored and (2) two or more distinct tuples 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 tuples belongs to a MV, that is, two distinct tuples that belong to base tables cannot have the same `ID` value because they are never stored at the same location. A global hash table does the mapping from the `ID` of a tuple to the address of the tuple's record.

The non-`ID` attributes of a table are either *standard* and are of one of the predefined types (e.g., integer, string, etc.) or are *reference* and contain the `ID` of a tuple in a different base table. We require that all reference attributes are not `NULL` and point to an existing tuple, that is, we impose a foreign key constraint for reference attributes. Attribute *destinationID* from Figure 5 is an example of a reference attribute, while the other attributes in the example schema are standard. We will use $\texttt{attr}(R)$ to denote the attributes of the table R and $\texttt{attr}(Q, R)$ to denote the set of attributes in $\texttt{attr}(R)$ that are referenced in the query Q.

2.2 Time Requirements

We have relied up to now on the reader's intuition when we used the terms efficient (i.e., logarithmic) access plans for queries and updates. A precise definition of the two terms follows, where the definition of an efficient update uses the definition of a primitive update.

Definition 2.1 (*efficient plan for a query*). Let Q denote a SQL query that references only base tables and Q_P – an access plan for Q. Assume that the size to encode a value for each of the attributes of the database schema and the size of the definition of Q are constant. The query plan Q_P is *efficient* exactly when it returns each tuple from the result of Q in $\sum_{i=1}^{m} log(|T_i|)$ time, where $\{T_i\}_{i=1}^{m}$ are the underlying tables of Q.

Table VII. The five μSQL update types

(*type*)	(*update*)
(1)‡	`insert into` $T(A_1, \ldots, A_a)$ `values` $(:P_1, \ldots, :P_a)$
(2)‡	`delete from` T `as` e `where` $e.\mathtt{ID} = :P$
(3)†‡	`update` T `as` e `set` $e.A = f(e.A)$ `where` $e.\mathtt{ID} = :P$
(4)†	`delete from` T `as` e `where` $\gamma(e)$
(5)†	`update` T `as` e `set` $e.A = f(e.A)$ `where` $\gamma(e)$

† f is a function that computes $f(x)$ in $O(|x|)$ time.
‡ a primitive update type.

The query plans for the queries `Q1` and `Q1` from Table I produced in Section 1.1 are examples of efficient query plans. Conversely, no efficient plan exists for the following query (see [Fredman 1981]), where A, B, and C are attributes of the table R defined over the domain of integers.

```
select sum(C) as S
from R
where R.A > :P1 and R.B > :P2
```

A primitive update can be of one of the three types shown in Table VII. Updates `U2` and `U3` from Table I are examples of primitive updates, while update `U1` from the same table is not a primitive update.

Definition 2.2 (*efficient plan for an update*). Let U be a SQL update that updates the base table T. Assume that the size to encode a value for each of the attributes of the database schema is constant. If U alters k tuples in the table T and U_P is an access plan for U, then U_P is *efficient* exactly when U_P can be decomposed into a sequence of k *primitive* updates, where each of these updates can be performed in $O(log|T|)$ time.

Update `U1` from Table I is an example of an efficient update that is not a primitive update. As we will see later, `U1` can be broken into a sequence of efficient updates.

2.3 Query Languages

In this subsection we describe two SQL query languages: μSQL query language and cSQL. The first dialect is the input query language for the RECS-DB algorithm (see Figure 3). The algorithm breaks a μSQL query into sSQL queries (see Figure 4). Queries `Q1` and `Q2` from Table I are examples of μSQL queries, while queries `Q5` and `Q6` from Table IV are examples of cSQL queries. The following intermediate definitions are needed to define the two SQL dialects formally.

Definition 2.3 (*efficient predicate*). An *efficient system* is one that has the following properties: (1) it is closed under the Boolean operations, (2) the problems of whether a predicate is in the system and the predicate subsumption problem are decidable, and (3) it can be checked in order length of the predicate definition time

whether a predicate holds for a list of bindings. An element of such a system is an *efficient predicate*. From here on, the symbol γ will refer to an efficient predicate unless explicitly stated otherwise.

For example, the results published in [Ferrante and Rackoff 1975] and [Revesz 1993] show that the predicates that can be constructed using the set of integers and a predefined number of variables over them, the comparison operators "$<$", "$=$", and "$>$", and the arithmetic operators "$+$" and "$-$" form an efficient system of predicates.

Definition 2.4 (*join condition, (valid) join graph, (inverse) join tree*). The *predicate formula* θ is a *join condition* for the set of tables $\{R_i\}_{i=1}^{r}$ iff the formula θ is a conjunction of atomic predicates of the form $R_l.A_k = R_d.\mathtt{ID}$, where $l, d \in [1, r]$, $l \neq d$, and A_k is a reference attribute that has the property that $\pi_{A_k} R_l \subseteq \pi_{\mathtt{ID}} R_d$. The *join graph* for θ has a node for each table in the set $\{R_i\}_{i=1}^{r}$ and there is a directed edge from R_l to R_d labeled A_k in G iff θ contains the atomic predicate $R_l.A_k = R_d.\mathtt{ID}$. A join graph is valid iff it is connected and acyclic. A *join tree* is a join graph that is a tree, where the edges are directed from a parent node to a child node and the root node is the only node without parents (i.e., edges going into it). An *inverse join tree* is a join graph that is a tree, where the edges are directed from a child node to a parent node and the root node is the only node without children (i.e, edges coming out of it).

Throughout the paper we only consider valid join graphs and use the symbol G to refer to such a graph and θ, or $\theta(R_1, \ldots, R_k)$ to refer to its join condition, where $\{R_i\}_{i=1}^{k}$ are the tables in the join condition of the graph.

For example, the two tables from query `Q1` (see Table I) are joined using the valid join condition `PACKET`.*destinationID* = `COMPUTER.ID`. The corresponding graph will contain two nodes with an edge from the node `PACKET` to the node `COMPUTER` labeled as *destinationID*. Figure 6 shows an example of a join graph, where the nodes labeled with digits induce an inverse tree in the graph with root the node labeled as "1".

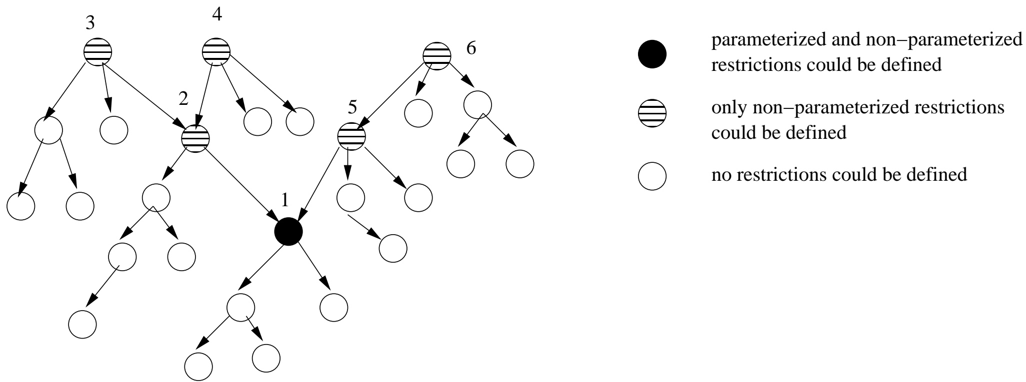

Fig. 6. Depicts the shape of the join graph of a μSQL query

Table VIII. The three sSQL query types

(*type*)	(*query*)
(1)	`select` $D_1, \ldots, D_d$ `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` $D_1, \ldots, D_d$ `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` $D_1, \ldots, D_d$ `from` R `where ID` $= :P_1$

(1) $\{|A_i|\}_{i=1}^{a}$ are distinct attributes of R,
(2) $\{|D_i|\}_{i=1}^{d}$ are distinct attributes of R,
(3) $\{A_i\}_{i=l+1}^{a}$ are non-reference attributes, and
(4) $\{A_i\}_{i=1}^{l}$ are non-`ID` attributes and A_{l+1} is a non-`ID` attribute for queries of the second type.

Definition 2.5 (*tuple ordering and valid tuple ordering in an inverse tree*). For a table R, a *tuple ordering* is defined using the syntax $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle\rangle$, where $\{A_i\}_{i=1}^{a}$ are distinct non-reference attributes of the table R. It denotes an ordering of the tuples in the table R, where the tuples are first ordered relative to the value of A_1 in ascending order if $dir_1 =$ `asc` and in descending order otherwise, next relative to the value of the attribute A_2 in direction dir_2 and so on. For an inverse join tree G^{-t} with nodes $\{R_i\}_{i=1}^{k}$ that is an induced subgraph of the valid join graph G with nodes $\{R_j\}_{j=1}^{r}$ and join condition θ, the tuple ordering $O = \langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle\rangle$ is valid relative to G^{-t}, where R is the join of the tables $\{R_i\}_{i=1}^{k}$, iff the following conditions hold.

(1) All the attributes that belong to the same table R_i are consecutive in O for every $i \in [1..k]$. We will refer to the tuple ordering defined by these attributes for the table R_i as O_i.
(2) If there is a directed path in θ from R_i to R_j and O_i and O_j are both non-empty, then O_j comes before O_i in O $(1 \le i \ne j \le k)$.
(3) For every j, $1 \le j \le k$, either O_j is the empty ordering, or O_j contains the system attribute `ID` of the table R_j, or every table reachable from R_j via a directed path in θ has an empty tuple ordering.

Consider the graph shown in Figure 6 and the subgraph induced by the nodes that are labeled. A valid tuple ordering for the subgraph must start with an ordering for the table with node "1" followed by an ordering on the table with node "2" or node "5". Also, if for example the tuple ordering for node "2" does not contain the attribute `ID`, then the tuple ordering for the nodes "3" and "4" must be empty.

The syntax of the cSQL dialect is shown in Table VIII, where square brackets are used to denote optional components. Roughly, a cSQL query is an admissible query that references a single table. By admissible, we mean that there exists an efficient plan for its execution using a single index. Restrictions 1 and 2 are added to Table VIII in order to make sSQL queries valid. Restriction 3 is added in order to disallow

Table IX. The three μSQL query types

(*type*)	(*query*)
(1)	`select` $D_1, \ldots, D_d$ `from` R_1 `as` $e_1, \ldots, R_r$ `as` e_r `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` $\ldots$ $\gamma_k(e_k)$ `and` $[A_1 = :P_1$ `and` $\ldots$ `and` $A_l = :P_l]$ [`order by` A_{l+1} $dir_{l+1}, \ldots, A_a$ dir_a]
(2)	`select` $D_1, \ldots, D_d$ `from` R_1 `as` $e_1, \ldots, R_r$ `as` e_r `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` $\ldots$ $\gamma_k(e_k)$ `and` $A_1 = :P_1$ `and` $\ldots$ `and` $A_l = :P_l$ `and` A_{l+1} `between` $:P_{l+1}$ `and` $:P_{l+2}$ [`order by` A_{l+1} $dir_{l+1}, \ldots, A_a$ dir_a]
(3)	`select` $e.D_1, \ldots, e.D_d$ `from` R_1 `as` e `where` $e.$`ID` $= :P_1$

(1) $0 \leq k \leq r$.
(2) $\{|D_i|\}_{i=1}^{d}$ are distinct attributes of the tables $\{R_i\}_{i=1}^{r}$.
(3) $\{|A_i|\}_{i=1}^{a}$ are distinct attributes of the tables $\{R_i\}_{i=1}^{k}$.
(4) The tables $\{R_i\}_{i=1}^{k}$ form an inverse tree in θ, which we will denote as G^{-t}.
(5) $\{A_i\}_{i=1}^{l}$ are non-`ID` attributes that belong to the table of the root node of G^{-t}.
(6) For queries of the second type, A_{l+1} belongs to the table of the root node of G^{-t}.
(7) $\{A_i\}_{i=l+1}^{a}$ are non-reference attributes.
(8) The tuple ordering in the `order by` condition of the query, when present, is a valid tuple ordering for G^{-t}.
(9) For every $i \in [k+1..r]$ there exists $j \in [1..k]$ such that there is a directed path in θ from R_j to R_i.
(10) $\{A_i\}_{i=1}^{l}$ are non-`ID` attributes and A_{l+1} is a non-`ID` attribute for queries of the second type.

ordering by reference attributes because the user of the system has no knowledge of how the values for such attributes are assigned. Restriction 4 guarantees that if the query has a partial match restriction on the attribute `ID`, then no other restrictions are specified because the query can return at most 1 tuple.

The syntax of the μSQL dialect is shown in Table IX. The added restrictions extend those for cSQL queries to guarantee that the join graph of a μSQL query has the shape shown in Figure 6 and that the ordering condition of the query is valid relative to Definition 2.5. The work presented in [Stanchev 2005] gives a theoretical proof of why μSQL cannot be extended further without braking certain desirable properties. Here, we give few examples that informally illustrate why this is the case. First, consider the following query.

`select *`
`from PACKET as` p, `COMPUTER as` c
`where` $p.desinationID = c.$`ID` and $p.size = :P_1$ and $c.name = :P_2$

One strategy is to create an index on the attributes *size* and *name* of the join of the two tables, but unfortunately such an index cannot be efficiently updated. The reason is that, for example, the change of the name of a computer can result in modifying all the entries in the index. Another strategy is to perform a nested index join of the two tables. However, it may be the case that for an outer tuple, the query plan scans all inner tuples without producing a single resulting tuple,

which will make the query plan inefficient. This reasoning can be extended to show that an efficient SQL query must contain a parameterized restriction on at most one table under certain assumptions.

Next, consider the following query.

select *
from PACKET as p, COMPUTER as c
where $p.desinationID = c.$ID and $p.size = :P_1$ and $c.name =$ "myPC"

An index on the join of the two tables will not be efficiently maintainable as explained earlier. Next, consider a nested index join of the two tables or subsets of their elements. If PACKET is the outer table, then an index that contains the packets that are destined for computers with name myPC will not be efficiently updateable. For example, suppose that there are only two computers that are both named myPC and half of the packets are destined for one of the computer and the other half for the other. Changing the name of one of the computers will require removing half of the tuples from the index, which cannot be performed efficiently. Alternatively, if COMPUTER is the outer table, then the query plan will not be efficient because it may be the case that all computers with name myPC are scanned without returning a single tuple from the query result. This reasoning can be extended to show that the tables with non-parameterized restrictions must form an inverse tree in the join graph in order for an efficient plan for the query to exist under certain assumptions.

The restriction that the tuple ordering of a μSQL query must be valid comes from the fact that the tables in the inverse tree of the join graph must be scanned in a particular way to maintain efficient execution of the query. For example, the labels in Figure 5 show one possible join order for the tables involved in the query. In general, if there is a direct edge from table R_1 to table R_2 in the join graph of the query, then the tuples from the index for R_2 must be scanned before the tuples from the index for R_1 in order to achieve an efficient query plan.

2.4 Update Language

A μSQL update can be of one of the five types shown in Figure VII, where the first three types characterize primitive updates.

3. COMPACT INDICES

Each sSQL query can be efficiently answered using a single index. However, as shown in the motivating example, creating a separate index for each sSQL query can lead to unnecessary duplication of data, which will not only increase the storage space, but will also slow down updates because multiple copies of the same data will need to be updated. In this section we define the novel concept of a *compact index*. It has the desirable property that it can be used to answer several sSQL queries that cannot be answered by a single regular index.

A compact index can be described by unordered (i.e., there is no order defined on the children of a parent node) node labeled tree, which we will refer to as the *description tree* of the index. The following definition shows how such a tree can be expressed using a string.

Definition 3.1 (*string description of a node labeled tree*). Let G^t be a node labeled tree, where each node of the tree has a label of the form $\langle \ldots \rangle$. We will use $\texttt{label}(n)$ to denote the label of the node n. Let $\mathcal{L}^{\downarrow}$ be a new node labeling function. For a leaf node n, we define $\mathcal{L}^{\downarrow}(n) = \texttt{label}(n)$. For a non-leaf node n with label $\langle L \rangle$ and children $n_1, \ldots, n_k$, we defined $\mathcal{L}^{\downarrow}(n) = \langle L, [\mathcal{L}^{\downarrow}(n_1), \ldots, \mathcal{L}^{\downarrow}(n_k)] \rangle$. We define the string description of G^t to be $\mathcal{L}^{\downarrow}(n^r)$, where n^r is the root node of G^t.

The definition of the syntax of a compact index also uses the following intermediate definition.

Definition 3.2 (*complete path in a tree*). Let G^t be a tree. A complete path in G^t is a path that starts at the root of G^t and ends at a leaf node. We will use $\texttt{cp}(G^t)$ to denote the set of all complete paths in G^t.

Definition 3.3 (*syntax of a compact index*). The description tree of a compact index is unordered node labeled tree. The general syntax of the label of a non-leaf node is either $\langle \bar{\gamma}, R, \langle A_1, \ldots, A_a \rangle \rangle$ or $\langle R, \{A_1, \ldots, A_b\} \rangle$, where R is a table, $\{A_i\}_{i=1}^{a} \subseteq \texttt{attr}(R)$, $\bar{\gamma}$ is a set of efficient predicates, $a \geq 0$, and $b > 0$. We will refer to nodes of the first type as *index nodes* and to nodes of the second type as *hash nodes*. Note that when $\bar{\gamma}$ contains only the predicate TRUE, we will use the syntax $\langle R, \langle A_1, \ldots, A_a \rangle \rangle$ to represent an index node. Given a non-leaf node n, we will refer to R as the node's table and write $\texttt{table}(n)$, to $\bar{\gamma}$ as the node's γ-condition and write $\bar{\gamma}(n)$, and to $\langle A_1 \ldots, A_a \rangle$ and $\{A_1, \ldots, A_a\}$ as the node's ordering label and write $\mathcal{L}(n)$. We will also refer to $\{\texttt{TRUE}\} \cup \{\texttt{FALSE}\} \cup \{ \bigcup_{\emptyset \not\equiv \bar{\gamma} \subseteq \bar{\gamma}(n)} \bigvee_{\gamma \in \bar{\gamma}} \gamma \}$ as the node's extended γ-condition and write $\bar{\gamma}^e(n)$ to denote it.

The general syntax of a leaf node n is $\langle R, \{C_1, \ldots, C_c\}, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle, type \rangle$, where $\{A_i\}_{i=1}^{a} \cup \{C_i\}_{i=1}^{c} \subseteq \texttt{attr}(R)$ and $type \in \{\texttt{ll}, \texttt{dll}\}$. We will refer to R as the node's table and write $\texttt{table}(n)$ to denote it, to $\{C_1, \ldots, C_c\}$ as the node's stored attributes and write $\texttt{st}(n)$ to denote them, to $\langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle$ as the node's ordering condition and write $\mathcal{L}(n)$ to denote it, and to *type* as the node's type and write $\texttt{type}(n)$ to denote it. We will refer to nodes for which $type = \texttt{ll}$ as linked list nodes and to nodes for which $type = \texttt{dll}$ as doubly linked list nodes. In order for a compact index to be valid, we impose the following restrictions.

(1) The attributes in the ordering labels of hash nodes are non-ID.

(2) The attributes in the ordering labels of non-leaf nodes and ordering conditions of leaf nodes are non-reference.

(3) If $\{n_i\}_{i=1}^{k}$ are the children of the node n, then $\pi_{\texttt{ID}}\texttt{table}(n) = \bigcup_{i=1}^{k} \pi_{\texttt{ID}}\texttt{table}(n_i)$.

(4) If the node n' with table R' has a non-trivial γ-condition that contains γ, then there must exists a leaf node n that is a descendent of n' and that has a table R such that $\gamma(t)$ is TRUE iff $t \in R$. Moreover, for any two such nodes n and n', all the intermediate nodes along the path from n' to n, if any, are either based on the table R or contain the efficient predicate γ in their extended γ-condition.

(5) If K is a complete path in G^t, then the attributes referenced in the ordering labels and ordering condition of the nodes along the path K are all distinct.

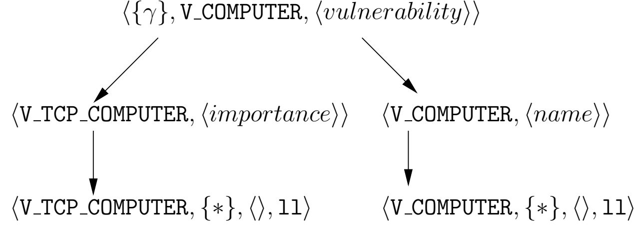

Fig. 7. The description tree of the example compact index

Before presenting a formal definition of the semantics of a compact index, we will informally explain the physical design it represents. In particular, each node in the description tree of a compact index corresponds to one or more data structures. Consider our motivating example from Section 1.1. In it we produced the compact index that has the description tree shown in Figure 7, where where γ is a predicate that is only true for tuples in the table V_TCP_COMPUTER. The root node of the tree corresponds to the index X_1 from Table V. It has two children that correspond to the set of indices X_2 and X_3, respectively. The leaf nodes in the figure correspond to the set of linked lists W_1 and W_2 from Table VI, respectively. Note that we have used $*$ to denote all the attributes of a node's table. According to definition 3.1, this compact index can be represented by the string: $\langle\{\gamma\}, \texttt{V_COMPUTER}, \langle vulnerability\rangle, [\langle\texttt{V_TCP_COMPUTER}, \langle importance\rangle, [\langle\texttt{V_TCP_COMPUTER}, \{*\}, \langle\rangle, \texttt{ll}\rangle]\rangle, \langle\texttt{V_COMPUTER}, \{name\}, [\langle\texttt{V_COMPUTER}, \{*\}, \langle\rangle, \texttt{ll}\rangle]\rangle]\rangle$.

Note that a root node with label $\langle\texttt{V_COMPUTER}, \{vulnerability\}\rangle$ describes a hash structures that stores the distinct values for *vulnerability* of the table V_COMPUTER and pointers to the data structures for the children of the node. Because queries Q5 and Q6 from Table IV contain a range predicate on the attribute *vulnerability*, we constructed an index on the distinct values of *vulnerability* rather than a hash structure. Also, note that a leaf node can have an ordering condition specified on it. For example, suppose that the left leaf node in Figure 7 has the label $\langle\texttt{V_TCP_COMPUTER}, \{*\}, \langle name\ \texttt{asc}\rangle, \texttt{ll}\rangle$. This denotes that the linked lists W_2 from table VI will be ordered according to the attribute *name* in ascending direction. The last parameter of the label of a leaf node represents whether the linked lists denoted by the node are singly linked or doubly linked.

We next describe the reasoning behind the five restrictions on the syntax of compact indices. Restriction 1 guarantees that no hash nodes on the ID attribute are created. The reason is that, as explained earlier, there is a global hash function that maps IDs to physical addresses. Restriction 2 guarantees that the ordering conditions and ordering labels define valid orders in terms of Definition 2.5. All compact indices do have valid orders in their nodes because, as we will see later, compact indices are created from μSQL queries. Restriction 3 guarantees that every element of a non-leaf data structure points to at least one data structure. Restriction 4 comes from the fact that queries can retrieve elements only from data structures that are represented by leaf nodes and therefore a γ-condition on a non-leaf node n is only meaningful if it helps to collect resulting tuple from the data structure represented by one of the leaf nodes of the subtree with root n. As shown

in [Stanchev 2005], adding a predicate from the extended γ-condition of a node to the node will not change the semantics of the compact index and, moreover, the extended γ-condition of a node contains exactly the set of predicates with that property. Restriction 5 is imposed because the attributes of a valid ordering must be distinct.

A compact index consists of a number of hash structures, linked lists, indices, and indices with marking bits. We will refer to the last as *marked indices*, where a formal definition follows.

Definition 3.4 (*syntax and semantics of a marked index*). The general syntax of a marked index is $\langle\langle\gamma_1,\ldots,\gamma_m\rangle, R, \langle A_1,\ldots,A_a\rangle\rangle$, where Condition 3 from Definition 2.3 for $\{\gamma_i\}_{i=1}^m$ is lifted. It represents a tree index of the tuples in R in the order $\langle A_1\ \texttt{asc},\ldots,A_a\ \texttt{asc}\rangle$. Each node in the tree has m marking bits, where the $i^{\texttt{th}}$ marking bit is set exactly when the node or one of its descendants contains a record that passes the condition γ_i.

Recall that Condition 3 from Definition 2.3 states that "it can be checked in order length of predicate definition time whether a predicate holds for a list of bindings". This restriction does not apply for the predicates of a marked index. However, as we will see in the proof of Theorem 3.9, the marking indices that are part of a compact index have desirable properties that permit their efficient update.

The acute reader may have noticed that we consider only indices in which all the attributes are in ascending direction. The reason is that, as shown in [Stanchev 2005], the marked index $\langle\langle\gamma_1,\ldots,\gamma_m\rangle, R, \langle A_1,\ldots,A_a\rangle\rangle$ can be used to efficiently retrieve tuples in the order $\langle A_1\ dir_1,\ldots,A_a\ dir_a\rangle$, where the elements of the set $\{dir_i\}_{i=1}^a$ can be either `asc` or `desc`.

Definition 3.5 (*semantics of a compact index*). Each node in the description tree of a compact index represents one or more data structures. If the description tree contains a single node n, then the node will represent a singly linked list (or doubly linked list when $\texttt{type}(n) = \texttt{dll}$) of the records $\pi^d_{\texttt{st}(n)}(\texttt{table}(n))$ in the order $\mathcal{L}(n)$, where π^d is a duplicate preserving projection.

Next, consider a leaf node n that is not the only node in the description tree. Suppose that $\{D_i\}_{i=1}^d$ are all the attributes in the ordering condition of n and its ancestors in the description tree. Then n will represent a set of singly linked lists (or doubly linked list when $\texttt{type}(n) = \texttt{dll}$). This set will contain one linked list for each distinct value of the attributes $\{D_i\}_{i=1}^d$ in R, where the linked list for $\{D_i = c_i\}_{i=1}^d$ contains the elements $\pi^d_{\texttt{st}(n)}\sigma_{D_1=c_1\wedge\ldots\wedge D_d=c_d}(\texttt{table}(n))$ in the order $\mathcal{L}(n)$.

Next, consider a root node n with children $\{n_i\}_{i=1}^u$. Let $\mathcal{L}(n) = \langle D_1,\ldots,D_d\rangle$. We introduce a table R' as the table $\pi_{D_1,\ldots,D_d}(R)$ with u derived attribute, where for a tuple $t \in R'$ the $i^{\texttt{th}}$ derived attribute, $i = 1$ to u, is a pointer to the data structure of n_i with values $\{D_i = t.D_i\}_{i=1}^d$. If n is an index node and $\bar{\gamma}(n) = \{\gamma_1,\ldots,\gamma_l\}$, then let $\{\gamma'_i\}_{i=1}^l$ be new predicates that have the property that $\gamma'_i(t')$ holds for a tuple $t' \in R'$, $i = 1$ to l, iff R contains a tuple t such that $\gamma_i(t)$ holds and $t.D_j = t'.D_j$ for $j = 1$ to d. In this case n is an will represent the marked index $\langle\{\gamma'_1,\ldots,\gamma'_l\}, R', \langle D_1,\ldots,D_d\rangle\rangle$. On the other hand, if n is a hash node with label $\langle R, \{D_1,\ldots,D_d\}\rangle$, then it will represent a hash structure for the table R' with

search attributes $D_1, \ldots, D_d$.

Finally, consider the case where n is a non-root and a non-leaf node with table R, attributes $D_1, \ldots, D_d$ in its ordering label, and child nodes $\{n_i\}_{i=1}^u$. Let $\{M_i\}_{i=1}^m$ be the attributes in $\mathcal{L}^{\uparrow}(n)$ that are not in the set $\{D_i\}_{i=1}^d$. Then n will represent a number of structures, where there will be a distinct structure for each distinct value of the attributes $\{M_i\}_{i=1}^m$ in R. For $\{M_i = c_i\}_{i=1}^m$ we define the table $R'_{c_1,\ldots,c_m}$ as $\pi_{D_1,\ldots,D_d}(\sigma_{M_1=c_1 \wedge \ldots \wedge M_m=c_m} R)$ with u derived attributes, where for a tuple $t \in R'$ the $i^{\mathtt{th}}$ derived attribute, $i = 1$ to u, is a pointer to the data structure of n_i with values $\{M_i = c_i\}_{i=1}^m$ and $\{D_i = t.D_i\}_{i=1}^d$. If n is an index node and $\bar{\gamma}(n) = \{\gamma_1, \ldots, \gamma_l\}$, then let $\{\gamma'_{i,c_1,\ldots,c_m}\}_{i=1}^l$ be new predicates that have the property that $\gamma'_{i,c_1,\ldots,c_m}(t')$ holds for a tuple $t' \in R'_{c_1,\ldots,c_m}$, $i = 1$ to m, exactly when R contains a tuple t such that $\gamma_i(t)$, $\{t.M_j = t'.M_j\}_{j=1}^m$, and $\{t.A_j = t'.A_j\}_{j=1}^a$ all hold. In this case the data structure of n for $\{M_i = c_i\}_{i=1}^m$ will represent the marked index $\langle \{\gamma'_{1,c_1,\ldots,c_m}, \ldots, \gamma'_{l,c_1,\ldots,c_m}\}, R'_{c_1,\ldots,c_m}, \langle A_1, \ldots, A_a \rangle \rangle$. On the other hand, if n is a hash node, then the data structure of n for $\{M_i = c_i\}_{i=1}^m$, will represent a hash structure with table $R'_{c_1,\ldots,c_m}$ and search attributes $A_1, \ldots, A_a$.

Note that the leaf nodes in a set of compact indices correspond to a set of linked lists. We will allow these lists to overlap, that is, when two or more linked lists include the same logical tuple (i.e., tuples with the same `ID`), then only a single record will be stored. This implies that different records can have different number of next/previous pointers and the offset for an attribute may change during a linked list traversal. This is why we also store information on where each attribute is located in each record. This can be achieved, for example, by using the data structure proposed by Zibin and Gil in [Zibin and Gil 2002].

Recall that Step 7 of the RECS-DB algorithm (see Figure 4) tries to find a set of compact indices of the smallest possible size that can be used to efficiently answer the current set of sSQL queries. In order for this goal to be well defined, we need to know what sSQL queries can be efficiently answered by a compact index. A theorem describing this set follows after several intermediate definitions.

Definition 3.6 (*permutation of a complete path*). Let K be a complete path in the description tree of the compact index J and let $\langle n_1, \ldots, n_k \rangle$ be the nodes along K (n_1 is the root node). Then a permutation Π converts K into a list of attributes that contains the attributes from the ordering labels of n_1 up to n_k in this order, where only the attributes in the ordering label of a hash node can be permutated.

Definition 3.7 (*queries of a complete path of a compact index*). Let J be a compact index with description tree G^t. Let K be a complete path in G^t and let $n_1, \ldots, n_{k+1}$ be the nodes along the path K, where $k \geq 0$ and n_1 is the root node of the tree. Table X shows the set of queries that we will associate with the path K. We will refer to this set as $Q_K(J)$.

Informally, $Q_K(J)$ is the set of queries that can be efficiently answered from the data structures along the path K. Condition 1 describes that only records from the linked lists can be retrieved. The reason is that we do not consider queries that select distinct values. Condition 2 describes that the attributes in the `where` and `order by` condition of Q are from the ordering conditions along the path K, where only the attributes in a hash structure can change place. The reasoning

Table X. Queries for a complete path $K = \langle n_1, \ldots, n_{k+1} \rangle$

(*type*)	(*query*)
(1)	select $D_1, \ldots, D_d$ from R_{k+1} [where $A_1 = :P_1$ and ... and $A_l = :P_l$] [order by $A_{l+1}\ dir^w_{l+1}, \ldots, A_a\ dir^w_a$]
(2)	select $D_1, \ldots, D_d$ from R_{k+1} 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 $A_{l+1}\ dir^w_{l+1}, \ldots, A_a\ dir^w_a$]
(3)	select $D_1, \ldots, D_d$ from R_{k+1} where ID $= :P_1$

(1) $R_{k+1} = \texttt{table}(n_{k+1})$ and $\{D_i\}_{i=1}^d$ are distinct attributes of n_{k+1}.
(2) There exists a permutation Π for $\langle n_1, \ldots, n_{k+1} \rangle$ s.t. $A_1, \ldots, A_a$ is a prefix of $\Pi(\langle n_1, \ldots, n_{k+1} \rangle)$ and $\{A_i\}_{i=l+1}^a$ are non-reference attributes.
(3) If A_i appears in the ordering condition of the node n_{k+1}, then it appears there with direction dir_i.
(4) If $\texttt{type}(n_{k+1}) = \texttt{dll}$, then w is either 1 or -1, else $w = 1$. Note that dir_i is asc or desc, $dir_i^1 = dir_i$, and $dir_i^{-1} = \texttt{asc}$ if $dir_i = \texttt{desc}$ and $dir_i^{-1} = \texttt{desc}$ otherwise.
(5) If n_r is the node in K with the biggest subscript that contains only attributes from the set $\{A_i\}_{i=1}^l$ in $\mathcal{L}^\uparrow(n_r)$, then each node in the set $\{n_i\}_{i=r+1}^k$ has the property that either $\texttt{table}(n_i) = R_{k+1}$ or $\bar{\gamma}^e(n_i)$ contains a predicate γ, where we define $\gamma(t)$ to be true for a tuple $t \in \texttt{table}(n_i)$ exactly when there exists a tuple $t' \in \texttt{table}(n_{k+1})$ such that t' and t have the same value for the search attributes of n_i.
(6) If an attribute from the set $\{A_i\}_{i=l+1}^a$ appears in the ordering condition of a non-leaf node along the path K, then the node must be an index node.

behind conditions 3 and 4 are that the records in a linked list can be retrieved efficiently only in a forward direction when the list is singly linked and in forward and backwards direction when the list is doubly linked. In order to understand condition 5, consider the compact index from Figure 7 and suppose that the root node has no γ-condition. Then the compact index will not be able to answer query Q5 from Table IV because there is no way to efficiently enumerate the records to X_1 that contribute to the query result (in this case $r = 0$). Condition 6 guarantees that ordering conditions cannot be defined on attributes that appear in the ordering condition of a hash node along the path K because hash structures do not have defined order.

Definition 3.8 (*queries of a compact index*). Let J be a compact index. We will denote by $Q(J)$ the set of sSQL queries that can be efficiently answered using only J and that reference a table from a leaf node in J.

We next present the theorem that describes what queries can be efficiently answered using a compact index.

THEOREM 3.9. *Let J be a compact index. Then the sets* $\bigcup_{K \in \texttt{cp}(J)} Q_K(J)$ *and* $Q(J)$ *are equivalent.*

Proof $\Rightarrow$ We will first show that $\bigcup_{K \in \texttt{cp}(J)} Q_K(J) \subseteq Q(J)$. Let $Q \in \bigcup_{K \in \texttt{cp}(J)}$. This means that there exists a complete path K in J such that $Q \in Q_K(J)$. Suppose

that the nodes along K starting from the root of the description tree of J are $\langle n_1, \ldots, n_{k+1} \rangle$ in this order and let the table of n_i be R_i ($i = 1$ to $k+1$). We will show why $Q \in Q(J)$ or more precisely how Q can be efficiently answered by using some of the data structures represented by the nodes along the path K.

(1) Consider first the case when Q is a query of type 1 (see Table X). Let r be the subscript of the node with the biggest subscript in K that has an ordering label that contains exclusively attributes from the set $\{A_i\}_{i=1}^{l}$. We set $r = 0$ when such a node does not exists. Let $A_1, \ldots, A_{m_1}$ be the attributes in the ordering label of n_1 and $A_{m_{i-1}+1}, \ldots, A_{m_i}$ be the attributes of the ordering label n_i for $i = 2$ to k. For consistency, we define $m_0 = 0$ and $dir_j =$ `asc` for $j = 1$ to l. We also define $\gamma_i(t)$ to be true for a tuple t, $i \in [r+1 \ldots k]$, only when there exists a tuple $t' \in R_{k+1}$ such that t' and t have the same value for the attributes in the ordering label of n_i.

Our claim is that Q can be efficiently answered by using the pseudo-code bellow. The first parameter i corresponds to the node n_i in K that is currently being visited. The second parameter $\mathcal{D}$ represents the data structure for n_i that we are visiting. Initially, $i = 0$ and $\mathcal{D}$ is the data structure for the node n_1.

```
00 follow(int i, physical design D){
01   if (i = k + 1){ //base case
02     if (w = 1) t = first record of D else t = last record of D;
03     while there are more records {
04       send construct_result(D_1 = t.D_1, ..., D_d = t.D_d);
05       if (w = 1) t = next record of D else t = previous record of D;
06     }
07   }
08   else {
09     if (i ≤ r){
10       D' = the data structure pointed to by the record in D with values
                              A_{m_{i-1}+1} = P_{m_{i-1}+1}, ..., A_{m_i} = P_{m_i};
11       follow(i + 1, D');
12     }
13     else {
14       if (i = r + 1) and the ordering label of n_{r+1} contains at least
                              one attribute from {A_i}_{i=1}^l {
15         let D' iterate over the structures pointed to by the records in
           D for which A_{m_{i-1}+1} = P_{m_{i-1}+1}, ..., A_l = P_l, where the data
           structures are visited in order A_{l+1} dir^w_{l+1}, ..., A_{m_i} dir^w_{m_i}
16           follow(i + 1, D');
17       } else {
18         if (type(D) = hash)
19           let D' iterate over the structures pointed to by the records in
             D
20             follow(i + 1, D');
21         } else {
22           let D' iterate over the structures pointed to by the records
             in D that pass the predicate γ_i in the order
             A_{m_{i-1}+1} dir^w_{m_{i-1}+1}, ..., A_{m_i} dir^w_{m_i}
```

```
23          follow(i + 1, D');
24        }
25      }
26    }
27  }
28 }
```

The access plan returns the desired values. It starts by consecutively probing the required data structures for the nodes $n_1, \ldots, n_r$ with the given values for the parameters. In this case only a single lookup in a hash structured or a marked index needs to be perform. Node n_{r+1} is special in the sense that both partial match and ordering attributes could have been specified on its attributes. If this is the case, then we need to scan the appropriate marked index, where we use the property that the direction for the ordering attributes can be flipped without compromising the search time of the index. Lines 18-24 cover the case when we need to return all records from the data structure, where ordering for the records can only be defined for marked indices. Note that Condition 5 from Table 3.7 guarantees that if $\gamma_i \neq$ `TRUE` is used in Line 22 of the pseudo-code, then γ_i is in the extended γ-condition of n_i and therefore the operation of Line 22 is well defined. In particular, from Definition 3.3 it follows that γ_i is the disjunction of several predicates for which there are marking bits in the marked index $\mathcal{D}$. We will refer to this predicates as $\bar{\gamma}$ A node in the marked index that does not have a bit set for at least one of the predicates in the set γ can be pruned out together with its subtree. Conversely, from Condition 4 of Definition 3.3 it follows that γ will hold for a record in the marked index exactly when (1) the record contains a pointer to a linked list or a marked index defined on a table R that is defined to contain exactly the elements that pass one of the predicates in the set γ (2) the record points a a marked index that has a bit set in its root node for one of the predicates in the set γ.

Note that only records that contribute to the query result are returned from each scan of a marked index or hash structure. Since n such scans are performed, the pseudo-code is efficient and therefore $Q \in Q(J)$.

(2) The pseudo-code when Q is a query of type 2 (see Table X) is similar to the code for the case when Q is a query of type 1. The only difference is in the case when $i = r + 1$. The reason is that n_{r+1} is the node that contains in its ordering label zero or more partial match attributes of the query Q and, in addition, it contains the attribute A_{l+1} on which the range predicate is defined. The pseudo-code for the case $i = r+1$ first tries to find a record t in the structure $\mathcal{D}$ for which $t.A_i = P_i$ for $i = m_{r-1} + 1$ to l and for which $t.A_{l+1} = P$, where $P = P_{l+1}$ if $dir^w_{l+1} =$ `asc` and $P = P_{l+2}$ otherwise. If such a record exists, then this tuple will be the first selected record from $\mathcal{D}$. If it does not, then the pseudo-code tries to find the first record t for which $t.A_i = P_i$ for $i = m_{r-1} + 1$ to l and $t.A_{l+1}$ is in the defined range. Then the scan continues until a record t for which $t.A_{l+1}$ is out of the range $[P_{l+1} \ldots P_{l+2}]$ is reached. This guarantees that the range predicate defined on A_{l+1} is handled correctly and in the required time bound.

(3)Finally, consider a query of type 3 (see Table X). Then Q can be answered by retrieving the record with the specified `ID`. This can be done using the global hash

function that maps `ID` values to physical addresses.

$\Leftarrow$ We will next prove that $Q(J) \subseteq \bigcup_{K \in cp(J)} Q_K(J)$. Indeed, let $Q \in Q(J)$. Let K be the complete path in the description tree of J that ends at the leaf node that references the table over which Q is defined. Let $\langle n_1, \ldots, n_{k+1} \rangle$ be the nodes along the path K and let $\langle R_1, \ldots, R_{k+1} \rangle$ be their respective tables. If Q is a sSQL query of type i (see Table VIII), then Q will also have the syntax of a critical query for the path K of type i (see Table X). As explained earlier, all conditions from Table X are needed, where some are required in order for the query to be a sSQL query and the rest are required in order for the query to be efficiently executable. Therefore, $Q \in Q_K(X)$. ■

The theorem shows how compact indices can be used to efficiently answer sSQL queries. The following theorem will describe how compact indices can be maintained after primitive update. However, since not every type of primitive update can be performed on every compact index efficiently, we impose a set of restrictions in the following definition.

Definition 3.10 (*admissible primitive update*) . Given a compact index J, we will say that the primitive update U is *admissible* for J iff the following statements are true.

(1) If n is a leaf node in J with table R and U is a primitive insertion that can affect the table R, then n does not contain an ordering condition.
(2) If n is a leaf node in J with table R and U is a deletion that can affect R, then the type of n should be `dll`.
(3) If n is a leaf node in J with table R and a U is a modification that can affect R, then we examine two cases:
—if the modification preserves the order defined by the ordering condition of n, then no restrictions are imposed,
—if the modification does not preserves the order defined by the ordering condition of n, then we require that the modification can be efficiently performed by a deletion followed by an insertion.

THEOREM 3.11. *Let J be a compact index and R be the table of the root node of J. If the size of the schema, the size of the description string J, and the maximum number of records in a node of a marked index in J are all constant, then (1) any admissible primitive update can be performed in logarithmic time relative to the size of R and (2) any primitive update that is not admissible cannot be performed in this time bound.*

Proof :

(1) An insertion of a tuple t can be performed using the following recursive pseudo-code, where $\mathcal{D}$ is initially the physical design for the root node of the description tree of J.

```
00 insert(tuple t, physical design D) {
01   if (type(D) = list)
02     if t hasn't already been inserted, then insert it at the
       beginning of D;
```

```
03     if t has already been inserted, then add the necessary
       pointers to make t the first element of D;
04   else {
05     {n_i}_{i=1}^k = children(n);
06     search for the record in D with all attributes matching those of t;
07     if such a record does not exist {
08       insert a record in D with values from the attributes of t;
09       for i = 1 to k
10         let D_i be an empty data structure for the subtree with root n_i;
11       add to the inserted record pointers to {D_i}_{i=1}^k;
12     }
13     else {
14       for i = 1 to k
15         let D_i be the structure for n_i pointed to by the found record;
16     }
17     for i = 1 to k
18       insert(t, D_i);
19     }
20   }
21 }
```

Lines 1-3 of the code cover the case when $\mathcal{D}$ is a linked list. In this scenario t is added to the beginning of the linked list if it hasn't been already. Note that when t is part of several linked lists, we store a single copy of t. Therefore, we only insert a record for t when it is added to the first linked lists and denote the belonging of t to other linked lists by adding the necessary pointers.

The rest of the code covers the case when $\mathcal{D}$ is not represented by a leaf node in J. Lines 7-12 cover the case when a new record needs to be inserted in $\mathcal{D}$. In this scenario the newly inserted records points to newly created empty physical designs, which are filled with information from t in Lines 17-18. Lines 13-15 cover the scenario when a record with the values from t already exists in $\mathcal{D}$. In this case $\{\mathcal{D}_i\}_{i=1}^k$ are set to the physical designs pointed to be the found record and lines 17-18 recursively call the `insert` procedure on them.

An efficient (logarithmic time) procedure for inserting an element in and deleting an element from a marked index is presented in [Stanchev 2005]. Informally, when a tuple is inserted into a marked index, one needs to check whether the tuple passes one of the defined γ-conditions and update the corresponding marking bit of the node in the marked index where the tuple is inserted and the parent nodes when necessary. When a tuple is deleted, updating the marking bit for the node from which the tuple was deleted will depend on whether there are any tuples that pass the corresponding γ-condition in the subtree rooted at the node. Again, the marking bits of the parent nodes may also have to be updated. A rotation may also need to be performed to balance the tree after each primitive operation, where marking bits will need to be updated during the operation. During the rotation, deciding whether a record passes a γ-condition can be done in constant time using the procedure described in the proof of Theorem 3.9.

The pseudo-code takes time logarithmic the size of R because it performs a constant number of index and hash scans of data structures that contain subsets of the elements of R.

A deletion can be performed using the following recursive pseudo-code. Initially, the global variable p is set to the ID of the tuple to be deleted, the global variable t to the tuple itself, and $\mathcal{D}$ to the physical design for the root node in the description tree of J.

```
00 int delete(physical design D) {
01     let n be the node for D;
02     if (π_ID t ∉ π_ID table(n)) return !empty(D);
03     if (type(D) = list) {
04         if the record that maps to the ID p has not been deleted, then
           delete it from all linked lists it appears in;
05         return !empty(D);
06     }
07     else {
08         let {n_i}_{i=1}^k be the children of the node n;
09         deletion_required = TRUE;
10         for i = 1 to k {
11             let D_i be the structure in n_i pointed to by the record in D
               with values for the search attributes from t;
12             if (delete(D_i) = TRUE) deletion_required = FALSE;
13         }
14         if (deletion_required = TRUE) delete from D the record
           with values for the search attributes from t;
15         return !empty(D);
16     }
17 }
```

Line 2 of the code checks whether the tuple t actually belongs to the table for the physical design $\mathcal{D}$. If this is not the case, then no deletion needs to be performed and !empty($\mathcal{D}$) is returned. Note that the we require from the delete pseudo-code to return FALSE when $\mathcal{D}$ is empty after the necessary operations on $\mathcal{D}$ are performed and return TRUE otherwise. Lines 3-5 cover the case when $\mathcal{D}$ is a linked list. We check that t has not already been delete and if it has not, then the procedure deletes t from all linked lists the tuple appears in. Since deletion is only allowed for doubly linked lists, the operation can be performed in constant time. Lines 8-15 cover the case when $\mathcal{D}$ is not a linked list. First, all physical designs for the descendants of n that contain the values for the corresponding attributes of t are found. The deletion is performed recursively on them. The variable *deletion_required* is initially set to TRUE. However, if any of the visited physical designs is not empty, then the *deletion_required* variable is set to FALSE. When the later is the case, no record needs to be deleted from $\mathcal{D}$.

Since the pseudo code visits a constant number of nodes in the description tree of J and it spends at most $O(log(|R|))$ time on each node, the worst case running time is proportional to logarithm of the size of R.

(2) Consider a primitive update that is not admissible (see Definition 3.10). If the first condition does not hold, then the primitive update is an insertion and J contains a leaf node that can be affected by the update. However, insertion in an ordered linked (or doubly linked) list cannot be performed in logarithmic time without the necessary indices and therefore the required logarithmic time bound cannot be achieved. If the second condition does not hold, then the primitive update is a deletion and J contains a leaf node that is of type linked list. However, deletion of a middle element from a linked list cannot be performed in logarithmic time without having both a forward and a backwards pointer. If the third condition does not hold, then the primitive update is a modification that changes the order of the records inside some of the linked lists of J. However, the modified record cannot be efficiently moved to its new place because either an efficient deletion or an efficient insertion cannot be performed. ■

4. THE RECS-DB ALGORITHM

In this section we explain in detail the steps from Figure 4, where subsections are numbered according to the step of the algorithm they describe.

4.1 Query Breakup

In this step we describe how to create a query plan for a μSQL query. The produced query plan consists of sSQL queries that can potentially reference newly introduced MVs.

Consider the general join graph of a μSQL query shown in Figure 6. We will first describe how to collect information about the white nodes (i.e., the nodes that represent tables on which no restrictions are defined). Consider a subtree $\{R_i\}_{i=1}^r$, where R_1 is the root table and $\{R_i\}_{i=2}^r$ are all white nodes. If we know the ID for a tuple in R_1, then the selected in the μSQL query values from $\{R_i\}_{i=2}^r$ can be gathered by executing the following query Q:

select $A_1, \ldots, A_a$
from R_1 as $x_1, \ldots, R_r$ as x_r
where $\theta(x_1, \ldots, x_r)$ and x_1.ID = :P_1,

where θ is the join graph for the examined subtree.

We will answer this query by first creating a sSQL query for each table in the set $\{R_i\}_{i=1}^r$. The query Q_{R_i} that corresponding to R_i is:

select $x.B_1, \ldots, x.B_b, x$.ID
from R_i as x
where x.ID = :P,

where $\{B_i\}_{i=1}^b$ are the attributes in R_i that are referenced in Q. Next, consider the following pseudo-code.

(1) tuple extract_attributes(table R, param P, join tree θ){
(2) $t = Q_R(P)$;
(3) x=attributes(t);
(4) for $R' \in$ child$^\theta(R)$
(5) P' = extract_ID(t,R');

(6) x+=extract_attributes(R',P',θ);
(7) }
(8) return x;
(9) }

We claim that Q can be answered in time proportional to the size of the query definition by executing extract_attributes(R_1, P_1, θ). In line 3, the function attributes extracts the non-ID attributes of t into x. In line 4, the function child$^\theta(R)$ returns the tables that are direct children of the table R in the join tree defined by θ. In line 5, the function extract_ID(t, R') returns the value of the reference attribute of t that references R'. Line 6 of the code does a recursive call to collect the values for the attributes of the descendants of R in the join tree defined by θ and adds the values that are collected to x.

First, note that the pseudo-code returns the result of Q because it traverses the join tree defined by θ to collect the values for all required attributes. Second, note that each query Q_i, $i = 1$ to r, can be executed in constant time. The extract_attributes function calls the queries $\{Q_i\}_{i=1}^r$ and therefore it takes $O(r) = O(|def(Q)|)$ time to answer Q, where we have use $|def(.)|$ to denote the size of the definition of the enclosed component.

We next present our algorithm for decomposing a μSQL query into sSQL queries. Visually, the algorithm scans the filled and dashed-filled nodes in Figure 6. The join order is such that, if there is directed path from node n_1 to node n_2, then the table for n_2 appears in a outer loop relative to the table for n_1. (A possible ordering for the nodes is shown in Figure 6 - see labels 1 through 6.) However, in order for the resulting query plan to be efficient, the table for each node will contain only the tuples that can join with tuples from inner tables in the nested loop join. We efficiently maintain this information by using a tuple counting technique (i.e., to each tuple of a table of a filled or dashed-filed node we add a derived attribute that counts how many tuples it joins with in the μSQL query). The details of the algorithm follow.

Recall that a μSQL query can be of one of three types shown in Table IX. A μSQL query of type 3 and of type 1 and 2 that is based on a single table is already a sSQL query. Therefore, assume that Q is a μSQL query of type 1 or 2 that references more than one table. Q will have the following general syntax, where some of the components may be missing.

select $D_1, \ldots, D_d$
from R_1 as $e_1, \ldots, R_r$ as e_r
where $\theta(e_1, \ldots, e_r)$ and $\gamma_1(e_1)$ and $\ldots$ $\gamma_k(e_k)$ and
$A_1 =$:P_1 and $\ldots$ and $A_l =$:P_l and A_{l+1} between :P_{l+1} and :P_{l+2}
order by A_{l+1} $dir_{l+1}, \ldots, A_a$ dir_a

Let G^{-t} be the inverse tree of Q that contains the tables on which restrictions can be placed (i.e., the tables $\{R_i\}_{i=1}^k$). Our first step is to add to the table R the derived attribute C_R^Q for $R \in \{R_i\}_{i=1}^k$. We will maintain these attributes in a way so that at the end of each transaction $t.C_R^Q$ is equal to the result of the following query for $t \in R$.

select count(*)

```
from R'_1 as e_1, ..., R'_m as e_m
where θ'(e_1, ..., e_m) and γ'_1(e_1) and ... and γ'_{m-1}(e_{m-1}) and
e_m.ID = t.ID
```

In the above query $\{R'_i\}_{i=1}^{m-1}$ are the tables in G^{-t} from which there is a directed path to R, R'_m is the table R, θ' contains the atomic predicates from γ that reference the tables $\{R'_i\}_{i=1}^{m}$, and γ'_i is the predicate in Q for R'_i ($i = 1$ to $m-1$). Informally, the introduced derived attributes have the property that for a tuple $t \in R$, the derived attribute C_R^Q stores the number of tuples t joins with in the G^{-t} part of the query Q.

Without loss of generality, we assume that the tables $\{R_i\}_{i=1}^{k}$ that form G^{-t} are ordered in such a way so that:

—R_1 is the root of G^{-t},

—if there is a directed path from R_x to R_y in G^{-t}, $1 \leq x, y \leq k$, then $y < x$, and

—if the attribute $A \in R_x$ comes before the attribute $B \in R_y$ in the ordering condition of the query Q, then $x \leq y$.

We next define the MVs $\{V_i\}_{i=1}^{k}$. For a table R_i that is a leaf node in G^{-t}, we define V_i using the following underlying query.

```
select e.*
from R_i as e
where γ_i(e).
```

In all other cases we define V_i using the underlying query:

```
select x.*
from R_i as e
where γ_i(e) and e.C^Q_{R_i} > 0.
```

Note that each MV consists of a subset of the tuples of a based table and therefore each MV will share its ID values with the ID values of the corresponding base table. This implies that each tuple from a MV will be stored at the same location as the base table tuple it represents. Note that the creation of a MV (or the definition of a base table) does not translate into the immediate creation of a data structure to store its elements. Actual data structures are created only in the last step of the algorithm and depend on the defined access requirements on the control data.

The generated MVs store the tuples from the tables in G^{-t} that can contribute to Q. As we will see in Section 4.3, these MVs are efficiently maintainable, that is, every primitive update can be applied to them in logarithmic time, relative to the size of the database, in the presence of appropriate indices. The MVs were created because Q will be broken down into sSQL queries that reference these views. In particular, let Q_1 be the query:

```
select attr(Q, V_1)
from V_1
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 O_1
```

and Q_i, $2 \leq i \leq k$ be the query:

select attr(Q, V_i)
from V_i as e
where $e.C_i = :P$
order by O_i.

Note that we have used O_i to refer to the part of the ordering condition in Q that references attributes from R_i and C_i is the label of the edge that begins at R_i in G^{-t}, $1 \leq i \leq k$,.

Then Q can be efficiently answered using the following query plan.

```
(1)  for t_1 ∈ Q_1(P_1, ..., P_l, P_{l+1}, P_{l+2}){
(2)    for t_2 ∈ Q_2(t_{index(child^{G^{-t}}(R_2))}.ID){
(3)    ...
(4)     for t_k ∈ Q_k(t_{index(child^{G^{-t}}(R_k))}.ID){
(5)      t = ∅;
(6)      for (int i = 1; i ≤ k; i++)
(7)       t+ = extract_attributes(R_i, t_i.ID, θ_i);
(8)      send t;
(9)     }
(10)    ...
(11)    }
(12) }
```

Note that $\texttt{child}^{G^{-t}}(R)$ is used to denote the table that the edge from R points to. The index function returns the subscript of the input relation, that is, $\texttt{index}(R_i) = i$. We have used θ_i to denote the join tree in θ induced by R_i and the set $\{R_j \mid k+1 \leq j \leq r$ and there exists directed path from R_i to R_j in θ that has the property that only R_i has a subscript in the range $[1 \ldots k]$ in the path$\}$. Informally, θ_i is the join tree consisting of the white nodes underneath the table R_i.

First, note that the query plan returns exactly the tuples from Q. The loops in Lines 1-4 traverse G^{-t} and scan only tuples that pass the where condition of Q. Note that there may be more than one way to do the join ordering, where the actual join ordering depends on the previous assumption on the ordering of $\{R_i\}_{i=1}^{k}$. Lines 5-8 then traverse the subtrees with white nodes rooted at the nodes $\{R_i\}_{i=1}^{k}$. The required attributes of the scanned tuples are stored in t and Line 8 of the pseudo-code returns a tuple from the query result.

Second, note that the query plan sends the resulting tuples in the correct order. In particular, the query plan traverses $\{R_i\}_{i=1}^{r}$ in the order defined by the ordering condition of the query Q. The fact that this ordering condition is valid for G^{-t} guarantees that the ordering conditions on each table are total orders except for the last visited non-empty ordering condition on a table, which guarantees that the tuples are returned in the correct order.

Lastly, note that the above query plan is efficient. In particular, no operations need to be performed for tuples outside the result of Q. On the other hand, in the worst case, for each tuple in the query result, r simple queries need to be executed. Each of $\{Q_i\}_{i=1}^{k}$ will take $O(log(x))$ time, where x is used to denote the total size of the tables $\{R_i\}_{i=1}^{k}$. The extract_attributes queries that need to be executed for each tuple of the query result will take $O(r)$ time. Therefore,

the total worst case execution time to return a tuple from the query result will be $O(log(x) \cdot k + r) = O(log(x) \cdot |def(Q)|)$.

4.2 Efficiently Maintaining Derived Attributes

In the previous subsection we introduced derived attributes as part of the μSQL query breakup step. In this subsection we present an algorithm for efficiently maintaining the derived attribute C_r that belongs to a table R_r that is the root of an inverse tree G^{-t} with join condition θ and tables $\{R_i\}_{i=1}^{r}$. For a tuple $t \in R_r$, $t.C_r$ is defined as follows.

$$\texttt{select count}(*)$$
$$\texttt{from } R_1 \texttt{ as } e_1, \ldots, R_r \texttt{ as } e_r$$
$$\texttt{where } \theta(e_1, \ldots, e_r) \texttt{ and } \gamma_1(e_1) \texttt{ and } \ldots \texttt{ and } \gamma_{r-1}(e_{r-1}) \texttt{ and } e_r.\texttt{ID}= t.\texttt{ID}$$

The algorithm first adds the derived attributes $C_{(R_i,R_j)}$ to the table R_i when there is a directed edge from R_j to R_i in θ $(1 \leq i, j \leq r)$. For $t \in R_i$, we define $t.C_{(R_i,R_j)}$ as the result of the query:

$$\texttt{select } count(*)$$
$$\texttt{from } R_j \text{ as } e,\ R'_1 \texttt{ as } e_1, \ldots, R'_m \texttt{ as } e_m$$
$$\texttt{where } \theta'(e, e_1, \ldots, e_m) \texttt{ and } \gamma_j(e) \texttt{ and } \gamma'_1(e_1) \texttt{ and } \ldots \texttt{ and } \gamma'_m(e_m) \texttt{ and}$$
$$e.\texttt{ID}= :t.\texttt{ID},$$

where $\{R'_q\}_{q=1}^{m}$ is the set of tables for which there is a directed path to R_j, γ'_q is the predicate condition for R'_q for $q = 1$ to m, and θ' is the part of θ that applies to the tables in the set $\{R_j\} \cup \{R'_q\}_{q=1}^{m}$. We also add to each table R_i, $i = 1$ to r, the derived attribute C_i, where $C_i = 1$ when R_i is the table of a leaf node in G^{-t} and $C_i = \prod_{R \in \texttt{parent}^{\theta}(R_i)} C_{(R_i,R)}$ otherwise, where $\texttt{parent}^{\theta}(R)$ is used to denote the set of tables from which there is a directed edge to R. Next, note that $t.C_{(R_i,R_j)}$ for $t \in R_i$ can also be expressed as the result of the following query.

$$\texttt{select } sum(C_j)$$
$$\texttt{from } R_j \texttt{ as } e$$
$$\texttt{where } \gamma_j(e) \texttt{ and } e.Pf^{\theta}_{(j,i)} = t.\texttt{ID}$$

The expression $Pf^{\theta}_{(j,i)}$ is used to denote the reference attribute of R_j that points to R_i in θ. We will refer to the above query expression as the *alternative definition* of $C_{(R_i,R_j)}$.

We will next show how the newly introduced derived attributes can be maintained. Consider a primitive update to R_y, $1 \leq y \leq r$. Such an update can only affect the derived attributes of the tables along the path from R_y to R_r in θ. We will propagate the update to the derived attributes of these tables in this order, where for each table R_i we will first refresh the attributes $C_{(R_i,R_j)}$ $(R_j \in \texttt{parent}^{\theta}(R_i))$ when R_i is not a leaf node. The attribute C_i can then be easily refreshed because its value is a function of the attributes $\{C_{(R_i,R_j)}\}_{R_j \in \texttt{parent}^{\theta}(R_i)}$.

First, note that a primitive deletion or a modification to R_y will not affect the derived attributes of R_y. A primitive insertion will introduce a new tuple t to R_y. If R_y is a leaf node in G^{-t}, then $t.C_y$ will be set to 1. Otherwise, $t.C_y$ will be set to

0 because the defined foreign key constraint on derived attributes guarantees that t cannot be initially referenced.

Second, consider how the derived attribute $C_{(R_i,R_j)}$ can be refreshed after a primitive update assuming that the derived attributes of R_j have already been updated ($1 \leq i \neq j \leq r$ and $R_j \in \texttt{parent}^{\theta}(R_i)$). A primitive update will affect at most two tuples in R_j. If no tuples in R_j are affected, then, by the alternative definition of $C_{(R_i,R_j)}$, it follows that $C_{(R_i,R_j)}$ will not be affected. If a tuple t is inserted in R_j and $\gamma_j(t)$ holds, then $(t.Pf^{\theta}_{(R_j,R_i)}).C_{(R_i,R_j)}$ will be incremented by 1 if R_j is a leaf node in G^{-t}. Similarly, if a tuple t is deleted from R_j and $\gamma_j(t)$ holds, then $(t.Pf^{\theta}_{(R_j,R_i)}).C_{(R_i,R_j)}$ will be decremented by 1 if R_j is a leaf node. Lastly, consider the case where a tuple in R_j is modified from $t^{\texttt{old}}$ to $t^{\texttt{new}}$. If $\gamma_j(t^{\texttt{old}})$ holds, then we will decrease $(t^{\texttt{old}}.Pf^{\theta}_{(R_j,R_i)}).C_{(R_i,R_j)}$ by $t^{\texttt{old}}.C_j$. Similarly, if $\gamma_j(t^{\texttt{new}})$ holds, then we will increase $(t^{\texttt{new}}.Pf^{\theta}_{(R_j,R_i)}).C_{(R_i,R_j)}$ by $t^{\texttt{new}}.C_j$. In all other cases, no action needs to be performed.

Note that the number of different $C_{(R_i,R_j)}$ attributes is equal to $r-1$ and the number of different C_i attributes is equal to r and therefore it takes order the size of the schema time to refresh a constant number of derived attributes. Also, note that there are other types of derived attributes that can be refreshed in constant time, but they are excluded from this work. For example, [Stanchev 2005] shows how to efficiently update derived attributes based on queries involving grouping and summation.

This step produces a refresh algorithm for derived attributes, which will be applied after every primitive update that can affect a derived attribute. In the next subsection, we describe the maintenance procedure for MVs.

4.3 Efficiently Maintaining Materialized Views

The only MVs that we introduced in the μSQL query breakup step were of the following form.

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

However, as described in [Stanchev 2005], the set of MVs to which a primitive update can be propagated in logarithmic time the size of database is much richer.

Consider the above MV V and an insertion of a tuple t to T. If $\gamma(t)$ holds, then the tuple $\pi_{A_1,\ldots,A_a} t$ will be inserted in V, where the `ID` value of the tuple will be preserved. Conversely, a deletion of a tuple t from T will lead to the deletion of the tuple with `ID` $t.$`ID` from V iff $\gamma(t)$ holds. Finally, a modification of a tuple t from $t^{\texttt{old}}$ to $t^{\texttt{new}}$ in T will involve:

(1) an insertion of $\pi_{A_1,\ldots,A_a} t^{\texttt{new}}$ to V iff $\gamma(t^{\texttt{old}})$ does not hold, but $\gamma(t^{\texttt{new}})$ does,

(2) a deletion of the tuple with `ID` $t.$`ID` from V iff $\gamma(t^{\texttt{old}})$ holds, but $\gamma(t^{\texttt{new}})$ does not, and

(3) a modification of the tuple with `ID` $t.$`ID` to $\pi_{A_1,\ldots,A_a} t^{\texttt{new}}$ in V iff $\gamma(t^{\texttt{old}})$ and $\gamma(t^{\texttt{new}})$ both hold.

Table XI. The compact indices for the three sSQL query types

(*type*)	(*compact index*)
(1)	$\langle R, \{A_1, \ldots, A_l\}, [\langle R, \{D_1, \ldots, D_d\}, \langle A_{l+1}\ dir_{l+1}, \ldots, A_a\ dir_a\rangle, \texttt{ll}\rangle]\rangle$
(2)	$\langle R, \{A_1, \ldots, A_l\}, [\langle R, \langle A_{l+1}\rangle,$ $[\langle R, \{D_1, \ldots, D_d\}, \langle A_{l+2}\ dir_{l+2}, \ldots, A_a\ dir_a\rangle, \texttt{ll}\rangle]\rangle]\rangle$
(3)	$\langle R, \{D_1, \ldots, D_d\}, \langle\rangle, \texttt{ll}\rangle$

Note that this step produces a set of parameterized primitive updates, which will be added to the set of primitive updates. The step also produces a MV refresh code that will be performed after every primitive update.

4.4 μSQL Update Breakup

In this section we describe how a μSQL update can be broken down into a set of primitive updates. Recall that the first thee types from Figure VII are already primitive updates. Consider an update of the forth type, that is an update that has the following general syntax.

```
delete from T as e
where γ(e)
```

Let Q be the following query:

```
select ID
from T as e
where γ(e),
```

and U be the following primitive update:

```
delete from T as e
where e.ID = :P.
```

Then the deletion can be performed using the following update plan.

```
for t ∈ Q
  execute U(t.ID);
```

For each such deletion, the update plan, the update, and the sSQL query that are generated will be added to the respective current sets.

An update of type (see Figure VII) modifies the tuples that pass a certain predicate. It can be broken down into a series of primitive modifications in an analogous fashion. If integrity constraints are defined, extra care needs to be taken to assure that the primitive updates are ordered in a way that does not violate the consistency of the database.

4.5 Converting sSQL Queries into Compact Indices

In this subsection we describe the algorithm for choosing the initial set of compact indices for the current set of sSQL queries.

Table XI shows how to convert a sSQL from each of the three types to a compact index.

Consider query `Q5` from Table IV. One regular index that can efficiently answer the query is on the table `V_TCP_COMPUTER` and the attributes *vulnerability* and *importance*. However, adding more attributes to this index will not impair its ability

to efficiently answer `Q5`. In other words, there are multiple indices that can be used to efficiently answer a given query. The following definition describes the set of compact indices that can efficiently answer the set of sSQL queries that a select compact index can answer.

Definition 4.1 (*cover of a compact index*). Let J be a compact index. We define $\texttt{cover}(J)$ as the set $\{J' \mid Q(J) \subseteq Q(J')\}$.

The following theorem describes the correctness and minimality of the mapping from Table XI.

THEOREM 4.2. *If the translation rules from Table XI produce the compact index J from the query Q, then $\texttt{cover}(J)$ is equivalent to the set of compact indices that can be used to efficiently answer the query Q.*

Proof:
$\Leftarrow$ Let $J' \in \texttt{cover}(J)$. From Definition 4.1 it follows that $Q(J) \subseteq Q(J')$. From Theorem 3.9 it follows that $Q \in Q(J)$ and therefore $Q \in Q(J')$.
$\Rightarrow$ We will show that if $Q \in Q(J')$, then $J' \in \texttt{cover}(J)$.
First, suppose that Q is a sSQL query of type 1 or 2 (see Figure VIII). Since $Q \in Q(J')$, from Theorem 3.9 it follows that J' must contain a complete path $K = \langle n_1, \ldots, n_{k+1} \rangle$ that has the properties described in Table X. Therefore, every query in the set $Q(J)$ is in the set $Q_K(J')$ and from Theorem 3.9 and Definition 4.1 it follows that $J' \in \texttt{cover}(J)$.
Second, suppose that Q is a sSQL query of type 3. From the fact that $Q \in Q(J')$ it follows that J' must contain a leaf node n with table R and stored attributes that are a superset of the attributes $\{D_i\}_{i=1}^{d}$. Therefore, every query in the set $Q(J)$ is in the set $Q(J')$ and from Definition 4.1 it follows that $J' \in \texttt{cover}(J)$. ■

4.6 Making Compact Indices Efficiently Maintainable

In this step we modify the compact indices created in the previous step of the algorithm in such a way so that the defined primitive updates can be efficiently propagated to them. The reason this step is require is because only admissible primitive updates (see Definition 3.10) can be performed on a compact index efficiently (see Theorem 3.11).

Consider a compact index J and a leaf node n of the index with table R. If a primitive insertion could affect R, then we will move the ordering condition from n and append it to the ordering label of the parent node of J. If a parent node does not exist, then a parent index node with table R is created before the procedure is applied. Conversely, if a primitive deletion is defined on R, the type of n needs to be changed to `dll` if it is `ll`. Finally, a modification that can affect R and that does not preserve the order defined by the ordering condition of n needs to be first broken into a primitive deletion followed by a primitive insertion and then J needs to be be modified in such a way so that the produced deletion and insertion are admissible for the compact index. From Theorem 3.9 it follows that the rewrites are correct in the sense that a rewritten compact index can efficiently answer all the sSQL queries that can be efficiently answered by the original index. Theorem 3.11 guarantees that the rewrites are minimal in the sense that they are needed to make each of the original indices efficiently maintainable.

We will refer to the type of compact indices that can be created after applying the rules from Table XI and this step as *simple compact indices*.

4.7 Merging Compact Indices

In the previous subsections we described how sSQL queries can be converted into compact indices and how these compact indices can be made efficiently maintainable. However, compact indices are evolved artifacts and we still have not demonstrated their full potential. In particular, the type of compact indices that can be created after applying the previous steps of the algorithm comprise only a small part of of their full dialect. The general syntax for the description string of a compact index is achieved by merging compact indices, where the idea of the merge is to save space without reducing the set of queries that can be efficiently answered. The following definition describes what condition must hold in order for several simple compact indices to be mergable into a single compact index.

Definition 4.3 (*compact index merging*). *Compact index merging* substitutes a set of simple compact indices $\{J_i\}_{i=1}^{k}$ with a simple compact index J. In order for the merge be *valid*, it must be the case that $J \in \bigcap_{i=1}^{k} \texttt{cover}(J_i)$. In order for the merge to be likely to save space, we require that the ordering label or ordering condition of the root nodes of the description trees of $\{J_i\}_{i=1}^{k}$ share an attribute in common, which is also the first attribute for leaf and index root nodes. In order for J to be as small as possible, we require that:

(1) No node or part of a node label can be removed from J without sacrificing the validity of the merge.

(2) An index node in J cannot be converted into a hash node without sacrificing the validity of the merge.

(3) J does not contain index nodes with more than one attribute in their ordering label.

Although it is possible to merge compact indices that do not follow the rule for common attribute prefix, we do not consider this case because it is unlikely to save space. For example, consider merging two compact indices with root nodes $\langle R, \langle A_1 \rangle\rangle$ and $\langle R, \langle A_2 \rangle\rangle$. Although it is possible to merge the compact indices into a compact index with root node $\langle R, \langle A \rangle\rangle$, where the data structure for the new node is an index on all distinct values of A_1 and A_2, doing so is unlikely to save space because distinct attributes usually do not share values in common.

In order to understand why we require that J does not contain index nodes with more than one attribute in the ordering label, consider the case where a node with the ordering label $\langle A_1, \ldots, A_a \rangle$, $a > 1$, is split into a index nodes, where the $i^{\texttt{th}}$ index node contains in its ordering label only the attribute A_i. The newly created compact index will be of smaller size because the distinct values of the attribute A_1 will be stored only once.

We next present an exponential time exact algorithm for merging a set of simple compact indices. Note however that the algorithms uses fine-grained statistics, like the number of distinct values of an attribute or a set of attributes, which

are usually unstable (see [Charikar et al. 2000]). This is why we also present an approximate polynomial-time algorithm that uses course-grained statistics. Note that both algorithms are static in the sense that they select a physical design only once based on the given statistical values. However, they can be made dynamic if we choose to run them periodically when new statistical information is available. Doing so will not change the execution time for the input queries, but can reduce the size of the storage.

4.7.1 *Exact Algorithm.* The exact algorithm first removes any compact indices based on singles nodes. The reason is that, as explained in Definition 3.5, leaf nodes represent linked lists that are implicitly merged. Next, the algorithm partitions the compact indices relative to the table they reference, where all the compact indexes that reference the same base table or MVs that references only that base table are clustered together. Compact indices from different clusters will be based on different attributes and will not be mergable (see Definition 4.3).

The simple compact indices in each cluster are merged using the following pseudo-code.

```
00 compact_indices merge(compact_indices $\bar{J}$) {
01   $result = \emptyset$;
02   $result\_size = \infty$;
03   let $\bar{J}''$ be the indices in $\bar{J}$ with single node description
     trees;
04   $\bar{J} = \bar{J} - \bar{J}''$;
05   for $\{\bar{J}'\}_{i=1}^{m} \in$ find_clustering($\bar{J}$) {
06    for $i = 1$ to $m$ {
07     if (merge_into_one($\bar{J}'_i$) = NULL)
08      skip to next iteration of the outer for loop;
09     $J_i^r$ = merge_into_one($\bar{J}_i$);
10    }
11    if calculate_size($\{J_i^r\}_{i=1}^{m} \cup \bar{J}''$) $< result\_size$ {
12     $result = \{J_i^r\}_{i=1}^{m} \cup \bar{J}''$;
13     $result\_size$ = calculate_size($\{J_i^r\}_{i=1}^{m} \cup \bar{J}''$);
14    }
15   }
16   if $result = \emptyset$, then return $\bar{J}$, else return $result$;
17 }
```

```
00 compact_index merge_into_one(compact_indices $J_1, \ldots, J_k$) {
01   for $i = 1$ to $k$ { $n_i$ = root($J_i$); $R_i$ = table($n_i$); }
02   if (find_smallest_prefix($\{n_i\}_{i=1}^{k}$) = NULL)
03    if ($k > 1$), then return NULL, else return $J_1$;
04   $A$ = find_smallest_prefix($\{n_i\}_{i=1}^{k}$);
05   $R = \mathrm{union}_{i=1}^{k} R_i$;
06   $\bar{\gamma} = \emptyset$;
07   $\bar{J}' = \emptyset$;
08   $is\_hash$ = TRUE;
```

```
09    for i = 1 to k {
10     if type(n_i) ≠ hash, then is_hash = FALSE;
11     if R_i ⊂ R {
12      let γ be such that γ(t) is true iff π_ID t ∈ π_ID R_i;
13      γ̄ = γ̄ ∪ {γ};
14      is_hash = FALSE;
15     }
16     J̄' = J̄' ∪ remove(J_i, A);
17    }
18    if is_hash = TRUE
19     return ⟨R, {A}, [merge_cPARTs(J̄')]⟩;
20    else
21     return ⟨γ̄, R, ⟨A⟩, [merge_cPARTs(J̄')]⟩;
22 }
```

The `merge` pseudo-code starts by removing the compact indices that have single node description trees. Since such indices are explicitly merged, they do not need to participate in the merge code. Line 5 iterates over all possible ways to split the input compact indices into groups, where there are $\mathcal{B}_k$ ways to do so ([Bell 1934]), where $2^k < \mathcal{B}_k < 2^{k \cdot log(k)}$ (i.e., $\mathcal{B}_k$ grows exponentially). Next, we check whether the compact indices in each partition can be merged into a single compact index by calling the `merge_into_one` function. If this is possible, then we compute the size of the solution and compare it to the current best result.

Note that the `calculate_size` method needs very detailed statistical information to estimate the size of the given compact indices. For example, calculating the size of the compact index shown in Figure 7 requires knowing the number of distinct values for the combination of attributes *vulnerability* and *name* of the table `V_COMPUTER`. The problem is that such fine-grained statistics is rarely available. When it is not, approximation algorithm such at the ones described in [Charikar et al. 2000] need to be applied.

The `merge_into_one` pseudo-code tries to merge several compact indices into one, where `NULL` is returned when this cannot be done. In Line 2, the `find_small_prefix` method tries to find a common attribute prefix for some permutations of the root nodes of the compact indices. The method returns `NULL` when such a prefix does not exist. Note that when all the input compact indices have root nodes that are hash nodes, it is possible that more than one attributes will qualify for a common prefix attribute. In this case, the `find_small_prefix` method returns the attribute that has the property that building a hash structure on it will save the most space, where statistical information can be used to find the answer.

The table R for the resulting compact index will be the union of the tables for the root nodes of the input compact indices. The *is_hash* variable is used to determine whether the root node of the newly created compact index should be a hash node or an index node. If any of the root nodes of the input compact indices is an index node or if the produced root node requires a γ-condition, then an index node is chosen.

Line 16 of the pseudo-code is called multiple times to remove the found common prefix attribute from the root nodes of the input compact indices. In particular, if

the common attribute is the last attribute in the ordering label of an index node or a hash node, then $\texttt{remove}(J_i, A)$ returns J_i without the root node (note that this operation returns a single compact index because it is applied on a simple compact index).

Lines 18-21 construct the result of the merge. In particular, the root node of the result has a table R, ordering label containing only the the found common prefix attribute, and the appropriate γ-condition. The child nodes of the resulting compact index are determined by recursively calling the `merge` method on the input compact indices from which the common prefix attribute has been removed.

The following theorem shows the correctness of the compact index merging algorithm.

THEOREM 4.4. *The simple compact index merging algorithm that is presented is valid in the sense described in Definition 4.3.*

Proof:

1. We will first show that if the algorithm merges the simple compact indices $\{J_i\}_{i=1}^k$ into the compact index J, then $J \in \bigcap_{i=1}^{k} \texttt{cover}(J_i)$. Let i be an arbitrary integer in the range $[1 \dots k]$. We will prove that $J \in \texttt{cover}(J_i)$, that is, if $Q \in Q(J_i)$, then $Q \in Q(J)$, where Q is a sSQL query. Indeed, suppose that $Q \in Q(J_i)$ and Q references the table R.

First, consider the case when Q is a simple query of type 1 or 2 (see Table VIII). Because J_i is a simple compact index, its description tree will contain at most three nodes that form a single complete path (see Table XI), where all nodes will be based on the same table. We will refer to this path as K. Then, from Theorem 3.9, it follows that the partial match, range, and ordering attributes of Q are from a permutation of K. However, J is constructed in such a way so that it contains a complete path K' that has these attributes as prefix for some permutation of K'. Moreover, the range and ordering attributes of Q do not appear in hash nodes of J. All the nodes along the path K' are based on the table R or are index nodes that contain in their γ-condition a predicate γ such that $\gamma(t)$ is `TRUE` iff $t \in R$. From Definition 3.7 it follows that $Q \in Q'_K(J)$ and from Theorem 3.9 it follows that $Q \in Q(J)$.

If Q is a sSQL query of type 3 (see Table VIII), then from Theorem 3.9 it follows that J_i will contain a leaf node with table R. However, because the merge preserves leaf nodes, J will also contain a leaf node based on the table R and from the reverse direction of Theorem 3.9 it follows that $Q \in Q(J)$.

2. Suppose that the algorithm is given the compact indices $\{J_i\}_{i=1}^k$ $(k > 1)$ and is unable to merge them (i.e. `merge_into_one` returns `NULL`). This can only happen when the root nodes of the compact indices do not share a common prefix. However, as stated in Definition 4.3, the compact indices in this case should not be merged because it is unlikely that the merge will save space. ■

Note that the merge introduces new nodes only as needed, creates hash nodes instead of index nodes when possible, and inserts a single attribute in the ordering labels of index nodes. In other words, the algorithm produces the smallest possible compact index that is a result of a valid merge, where the "smallest possible" is in the sense described in Definition 4.3.

Note as well that the algorithm is exponential because it considers all possible partitionings for $\{J_i\}_{i=1}^k$, which takes $O(\mathcal{B}_k)$ time.

4.7.2 *Approximate Algorithm.* We next present a polynomial time approximate algorithm for merging simple compact indices. Note that the exponential complexity of the previous algorithm came from the method `find_clustering` that enumerates all possible ways the input compact indices can be clustered. Conversely, the new algorithm uses greedy heuristics to select a single clustering. In order to describe how the new version of `find_clustering` works, suppose that $\{J_i\}_{i=1}^k$ are the input simple compact indices and $\{K_i\}_{i=1}^k$ are the corresponding complete paths in their description trees (the description trees will be paths because the compact indices are simple).

The algorithm starts by looking at the root nodes of the paths $\{K_i\}_{i=1}^k$. In particular, all attributes of the ordering labels of hash nodes and the first attribute of the ordering labels of index nodes and ordering conditions of leaf nodes are considered. For each such attribute A, we estimate how much space will be saved if we merge the compact indices that contributed the attribute A in a new compact index with root node that contains A in its ordering label. This can be achieved by multiplying the expected number of distinct values of A by the size needed to represent an element of A and then multiplying the result by the number of compact indices that will participate in this merge. If l is the sum of the sizes of the description strings of $\{J_i\}_{i=1}^k$, then this step will take $O(l \cdot k)$ time.

The compact indices that produced the attribute that is expected to save the most storage are put in one cluster and then the procedure is applied on the remaining compact indices. The procedure is repeatedly applied until all input compact indices are put in clusters.

Note that `find_clustering` will always return a single clustering and therefore the `for` loop at Line 5 of the `merge` method will always be executed at most once. Lines 11-14 of the pseudo-code can be substituted with Line 12 because the new algorithm finds at most one solution and there is no need to compare it to other solutions. Since `merge_into_one` is recursively called $O(l)$ time, the total running time of this approximate algorithm is $O(l^2 \cdot k)$

5. CONCLUSION

In this paper, we presented the RECS-DB algorithm. The goal of the algorithm is to aid the software engineers of RECPs. In particular, we introduced the novel concepts of μSQL and compact indices. μSQL defines a dialect of SQL that is efficiently executable. The syntactic description of this language is useful because it guides the potential users of the system regarding what queries and updates can be specified as input to RECS-DB. Compact indices allow the RECS-DB algorithm to produce smaller physical designs than what can be achieved by current commercial database engines. Moreover, unlike existing database engines, the RECS-DB algorithm provides a boundary on the worst-case execution time.

One topic for future research is making the physical design process dynamic. In other words, the selected physical design may need to be evolved as the statistical information changes. Another possible area for future research is developing a system that supports SQL with realtime requirements in a distributed server envi-

ronment. A possible assumption for such a model can be that there is a guaranteed throughput between the different servers. Novel problems, such as data placement, parallelized algorithms, and CPU utilization, arise in this scenario.

REFERENCES

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

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

Bancilhon, F. and Ferran, G. 1994. ODMG-93: The Object Database Standard. *IEEE Data Eng. Bull. 17,* 4, 3–14.

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

Charikar, M., Chaudhuri, S., Motwani, R., and Narasayya, V. R. 2000. Towards estimation error guarantees for distinct values. *PODS*, 268–279.

eXtremeDB. *eXtremeDB User's Guide.* eXtremeDB, www.mcoject.com.

Ferrante, J. and Rackoff, C. 1975. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM J. Comput 4,* 1, 69–76.

Fredman, M. L. 1981. A Lower Bound on the Complexity of Orthogonal Range Queries. *J. ACM 28,* 4, 696–705.

Golab, L. and Özsu, T. 2005. Update-Pattern-Aware Modeling and Processing of Continuous Queries. *SIGMOD*, 658–669.

K.Mehlhorn and Tsakalidis, A. 1985. Dynamic Interpolation Search. *In Proceedings of the 12th international conference on automata, lanaguages and programming*.

Lehman, T. and Carey, M. 1986. Query Processing in Main Memory Database Management System. *SIGMOD*, 239–250.

MonetDB. *MonetDB User's Guide.* MonetDB, http://monetdb.cwi.nl.

Rao, J. and Ross, K. 2000. Making B^+ Trees Cache Conscious in Main Memory. *ACM SIGMOD*, 475–486.

Revesz, P. Z. 1993. A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *TCS 116,* 1, 117–149.

Salzberg, B. 1988. *File Structures: An Analytic Approach.* Prentice-Hall.

Smith, P. and Barnes, G. 1987. *Files and Datbases: An Introduction.* Addison-Wesley.

Stanchev, L. 2005. Automating Physical Design for an Embedded Control Program. *Ph.D. thesis*.

Stanchev, L. and Weddell, G. 2002. Index Selection for Compiled Database Applications in Embedded Control Programs. *Centers for Advance Studies Conference (CASCON)*, 156–170.

Stanchev, L. and Weddell, G. 2003. Index Selection for Embedded Control Applications using Description Logics. *International Workshop on Description Logics*, 9–18.

TimesTen. *TimesTen User's Guide.* TimesTen, www.timesten.com.

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.

Weddell, G. 1989. Selection of Indexes to Memory-Resident Entities for Semantic Data Models. *IEEE TKDE 1,* 2, 274–284.

Willard, D. 1983. Log-logarithmic worst case range queries are possible in space O(N). *Inform. Process. Lett. 17*, 81–89.

Wirth, N. 1972. *Algorithms+Data Structures=Programs.* Prentice-Hall.

Zibin, Y. and Gil, J. Y. 2002. Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-dispatching. *Conference on Object Oriented Programming Systems Languages and Applications*, 142–160.

Zibin, Y. and Gill, J. 2003. Two-Dimensional Bi-Directional Object Layout. *ECOOP 2003*, 329–350.