Programming Embedded Computing Systems using Static Embedded SQL

Lubomir Stanchev
Computer Science Department
Indiana University-Purdue University Fort Wayne, Indiana, U.S.A.
stanchel@ipfw.edu

Grant Weddell
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
gweddell@uwaterloo.ca

September 22, 2008

Abstract

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 usually have stringent performance and space requirements.

In this paper, we show how parts of the software can be written in a declarative programming language such as SQL. This is a challenging task 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 applying the following strategies.

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

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, the programmers can specify their requirements for control data access in a high-level declarative language, such as SQL. We anticipate that the resulting higher-level code will therefore be faster to create and easier to understand, test, and maintain.

1 INTRODUCTION

We propose a new approach for developing *Realtime Embedded Control Programs* (RECPs) that exploits the SQL API that is used by relational database technology. Consider the typical 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 assembly language. 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.

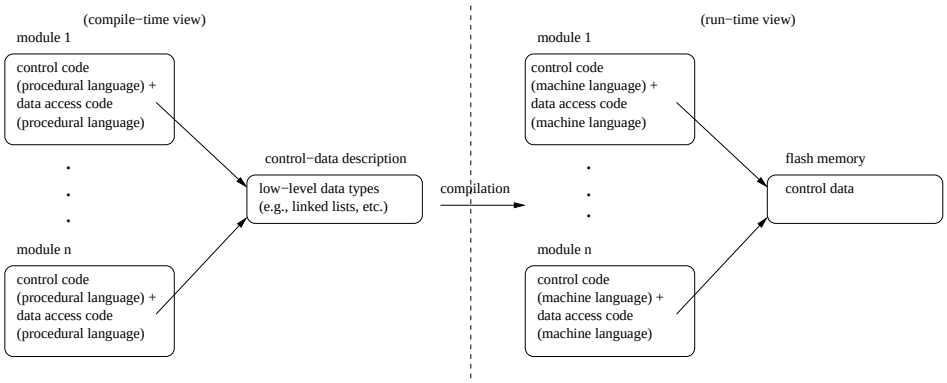

Figure 1: The typical architecture of a realtime embedded control program.

As an alternative, consider the architecture depicted in Figure 2. Now the software engineer can specify the data access code using a declarative language, such as SQL, and describe 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.

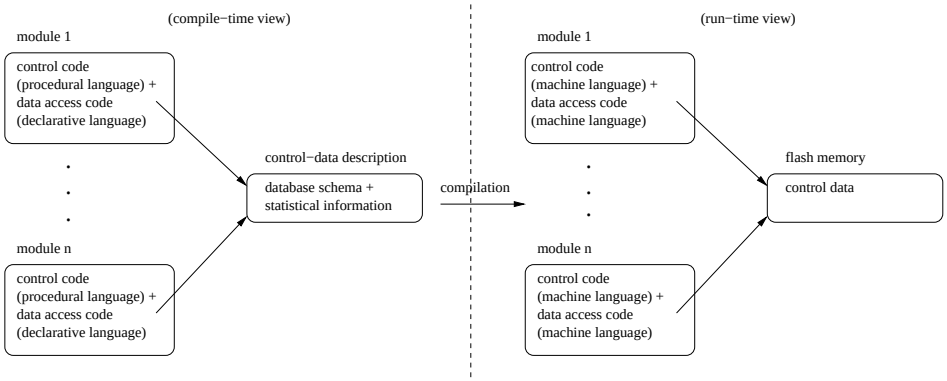

Figure 2: The proposed architecture of a realtime embedded control program.

The control code in Figure 2 can continue to be compiled by an existing language compiler. However, a new approach is needed for compiling the data access code that is specified in a declarative language. In this paper, we present an algorithm for performing this compilation. Since the algorithm is part of a software system that 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.

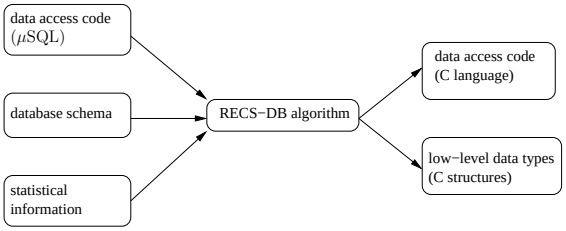

Figure 3: Input and output of the RECS-DB algorithm.

Note that the output of our algorithm becomes part of the source code for 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 2 to the compile-time view from Figure 1. The output of the algorithm depends on the particular RECP, that is, the data access code and low-level structures are custom made for the input SQL workload. In addition, the data access code that is generated by the algorithm is efficient in the sense that each item of a query result and the modification of an item from a table are both done in logarithmic time relative to the size of the database.

The logarithmic time-bound for data operations is chosen because it corresponds to the time it takes to probe a search tree. However, data structures that allow record retrieval in sub-logarithmic time, such as *y-fast trees* ([36]) and *interpolation search trees* ([15]), can also be considered. 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 and hash structure implementation, such as the number of records per node in an index.

The data structures for the control data extend traditional search trees and hash structures in a way that saves space. This is achieved by merging data structures that share data into a single data structure in a way that eliminates unnecessary data redundancy without sacrificing performance. The merging algorithm produces the data structures that are expected to take the least amount of space, where the available statistical information is used to approximate the expected size of a data structure.

Note that the problem that is solved in this paper is hard and relatively unexplored. The reason is that database management systems are traditionally associated with big footprint size and there is no guarantee on the performance of the different operations. However, RECP require small footprint size and guaranteed worst-case bound on all operations.

The proposed algorithm for automatically creating and manipulating the control data is beneficial because the code that creates and manages the control data is predominant for a RECP. For example, when the Embedded Control Benchmark workload ([39]) was rewritten to reference SQL statements instead of C data structures, than 85% of the code lines were SQL statements (i.e., statements that declare or manipulate the control data). Note that the static SQL (i.e., SQL statements that are not dynamically created) that is embedded in a flow control language, such as C, is referred to as *static embedded SQL*. In this paper we describe why using such statements in the creation of RECS is beneficial and our case study in Section 5 strengthens this premises.

1.1 The SQL Language

We chose SQL as the input language to our algorithm (see Figure 3) because it is the *de facto* standard for accessing relational databases. Moreover, it is a high-level declarative language. This means that SQL queries describe what information is to be retrieved, but not how to retrieve this information. Since not every SQL operation can be executed efficiently ([12]), 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 ([33]) and 93% of the Embedded Control Benchmark workload ([39]) using μSQL. We believe that these results are relevant because TPC-C is an online transaction processing (OLTP) benchmark that consists of a mix of simple queries and updates, which closely resembles the typical throughput generated by a RECP. On the other hand, the Embedded Control Benchmark described in [39] is based on the MINIX

operating system ([30]) and therefore describes the behavior of a typical RECP.

The μSQL languages will be introduced latter in Section 2.3 and is based on the following basic syntax for defining SQL queries.

`select [distinct]` $A_1, \ldots, A_n$
`from` $R_1, \ldots, R_k$
`where` *condition*
`order by` B_1 $dir_1, \ldots, B_b$ dir_b

This query assumes that $\{R_i\}_{i=1}^{k}$ are defined as *tables* in the database schema. Each table consists of a bag of rows (a.k.a. *tuples*), where a row is defined by the values for the different columns (a.k.a. attributes). Every attribute has a type that denotes the domain of values admissible for the specific column. The above query produces a new table from the input tables $\{R_i\}_{i=1}^{k}$. It does so by first performing the cross product of the tables, defined as R_1 X ... X R_k in *relational algebra*. Formally, the cross product is defined as appending all possible combination of tuples from the k tables, that is, the cross product will contain $\prod_{i=1}^{k} |R_i|$ tuples and $\sum_{i=1}^{k} |\texttt{attr}(R_i)|$ attributes, where $|R|$ is used to denote the number of tuples in R and $|\texttt{attr}(R)|$ - the number of attributes in R.

Next, the predicate *condition* is used to select the tuples that pass this predicate, where this is defined as $\sigma_{condition}$ (R_1 X ... X R_k) in relational algebra. Note that *condition* is a Boolean expression that references the attributes of the tables. A combination of a selection that contains a conjunctive of equality constraints and a cross product is sometimes referred to as a *join*. In this operation only the first resulting attribute is preserved for equivalence attributes (i.e., if the query involves the condition $A = B$, then the values for the attributes A and B will be identical in the result and only the column for A is kept).

A a projection is applied on the resulting tuples, which is denoted in relational algebra as $\pi^{d}_{A_1,\ldots,A_n}(\sigma_{condition}$ (R_1 X ... X R_k)). This operation keeps only the columns for the attributes $\{A_i\}_{i=1}^{n}$. The operation π^{d} denotes duplicate preserving projection. Note that when the `distinct` keyword is present after the `select` statement, then duplicates tuples (i.e., tuples that have the same value for all the attributes) are substituted with a single tuple. In relational algebra, the query with duplicate eliminating projection will be expressed as $\pi_{A_1,\ldots,A_n}(\sigma_{condition}$ (R_1 X ... X R_k)).

SQL allows the statement "`select` $A_1, \ldots, A_n$" to be substituted with the shorthand "`select` *". The new statement denotes that all attributes that are not duplicated (i.e., no equality constraint is defined on them) are preserved. In particular, for a join condition $R_i.A = R_j.\texttt{ID}$, the attribute A is preserved, but the attribute `ID` of the table R_j is not. SQL also uses the syntax "`select count`$(*)\ldots$" to denote the number of tuples that pass the specified condition and "`select sum`$(A)\ldots$" to denote the sum of the values for the attribute A of all tuples that pass the specified condition. Sometimes, we will allow tables to be renamed by using the syntax: "R `as` r" in the `from` clause, where R is the name of a table and r is the new name.

The last statement of the query orders the resulting tuples relative to the value of the attribute B_1 in ascending order if $dir_1 =$ `asc` and descending order if $dir_1 =$ `desc`. If the two resulting tuples have the same value for B_1, then they are ordered according to the value for the attribute B_2 in direction dir_2 and so on.

SQL also has a data manipulation part for inserting and deleting tuples from a table. The following syntax is used to insert the tuple t with values for the attributes $\{A_i\} = c_i$, $i = 1$ to k, into the table R.

`insert into` R $(A_1, \ldots, A_k)$ `values` $\{c_1, \ldots, c_k\}$

The operation of deleting tuples form the table R can be specified using the following general syntax.

`delete` from R
`where` *condition*

In the query *condition* is a boolean expression that references the attributes of R. Only the tuples that pass this condition will be deleted. Finally, the following SQL expression is used to denote the operation of changing the values of the attributes $\{A_i\}_{i=1}^{a}$ for all tuples of the table R that pass the specified *condition*.

`update` R
`set` $A_1 = c_1, \ldots, A_a = c_a$
`where` *condition*

1.2 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 that data flows from several places are merged. A brief description of each of the steps follows.

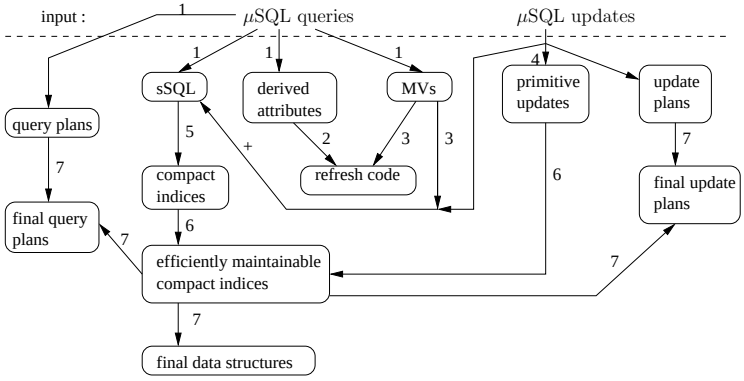

Figure 4: Steps of the RECS-DB algorithm

1. The input queries, which are expressed in μSQL, are decomposed into simpler formulations in a dialect called sSQL. μSQL is the general dialect for SQL queries that are allowed as input, whereas sSQL (stands for simple SQL) admits queries that reference a single table in the `from` statement. For each μSQL query, the step produces *derived attributes*, *materialized views* (MVs), and a *query plan* that references the sSQL queries that are generated. Derived attributes are attributes that have a value that is a function of the other attributes in the database. A MV is a table that store the result of a SQL query over the existing tables. A query plan describes how a query is to be executed and can be written in a general programming language, such as C.

2. This step generates refresh code for the derived attributes. Note that the value of a derived attribute should change as the values of the attributes it is based on change. In this step the low-level code for doing so is generated.

3. Refresh code for the MVs is generated. This process can entail the generation of additional sSQL queries. Note that every time the underlying tables of a MV are updated (i.e., the tables on which the underlying SQL query of the MV is based), the MV needs to be refreshed accordingly. Usually, the one-time synchronization of a MV is referred to as a *refresh*, while the continuous process of synchronizing the content of a MV to avoid staleness is referred to as *maintenance* ([27]). This process can involve the execution of a SQL query to determine what changes should be applied to the MV that is to be refreshed. Such queries, which are sometimes referred to as *delta queries*, are added to the current set of sSQL queries (i.e., the set of queries that are to be supported efficiently). To avoid confusion, we will refer to tables that do not store query results (i.e., tables that are not MVs) as *base tables*.

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

5. A single *compact index* is generated for each sSQL query. This index will allow the query to be executed efficiently. A compact index is a novel search data structure that is introduced in Section 3. It can consist of a combination of search trees and hash indices. Compact indices are beneficial because they reduce the need for storing redundant 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 current 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.

1.3 Motivating Example

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

1.3.1 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 [14]). The table `COMPUTER` contains information about the computers that are possible packet destinations.

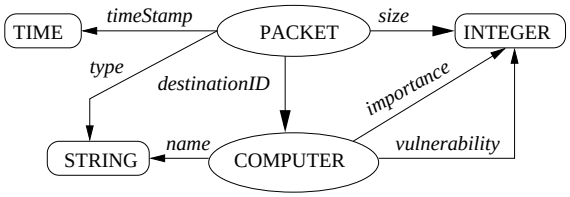

Figure 5: Example schema

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`. This implies that for every tuple $t \in$ `PACKET`, there exists a tuple $t' \in$ `COMPUTER` for which $t.destinationID = t'.\texttt{ID}$.

Suppose that we are given queries `Q1` and `Q2` and the updates `U1`, `U2`, and `U3` that are defined in Table 1, where $:P$ is used to denote a query parameter. Query `Q1` asks for recent TCP/IP packets (the table `PACKET` contains information only about the last five minutes) 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 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`, respectively. Update `U3` represents the insertion of a tuple in the table `COMPUTER`.

We next show how some of the steps of the RECS-DB algorithm work on our example.

1.3.2 Step 1

In this step we break 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 *packets* stores the number of tuples from the table `PACKET` a tuple from the table `COMPUTER` joins with (i.e., the number of tuples that have *destinationID* equal to the `ID` attribute of the tuple), 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 2). Next, RECS-DB creates the MVs shown in Table 3. The MV `V_TCP_PACKET` contains only the TCP/IP packets from the table `PACKET`. The MV `V_TCP_COMPUTER` contains

(*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 1: An example workload

(*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 2: New derived attributes for the table COMPUTER

(*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 $c.packets > 0$

Table 3: Intermediate views

(*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 4: Two example sSQL queries

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 MV 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);
```

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);
```

Note that we have used join to denote the result of joining (i.e., concatenating) the tuples specified as the parameters, where the ID attribute of the second tuple is not preserved. The operator send denotes 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 a tree 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 the table COMPUTER in the outer for-loop 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 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.

1.3.3 Step 5

This step creates search trees or hash structures. For example, an index on the MV V_TCP_COMPUTER and attributes *vulnerability* and *importance* can be used to efficiently answer Q5 from Table 4. Examples of search tree indices include AVL trees ([1]), AA trees ([3]), red-black trees ([18]), B trees [37, 22, 23], CSB^+ trees [20], T trees [17], and B^+ trees ([5]).

(*name*)	(*keys* (k))	(*values*)
$X_1^\dagger$	`select distinct` *c.vulnerability* `from V_COMPUTERS as` c `order by` *c.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 marker bit

‡ $\&X_2(k)$ denotes the address of the index $X_2(k)$

Table 5: Indices for efficiently answering queries `Q5` and `Q6`

(*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* $= :P_2$

Table 6: Linked lists for efficiently answering queries `Q5` and `Q6`

1.3.4 Step 7

The compact indices produced in Step 5 can be used to efficiently answer the set of simple SQL queries. However, we go a step further by performing a compact index merging procedure that reduces the size of the auxiliary data by eliminating redundant data. In order to demonstrate out approach, we will next show how to merge the compact indices for the queries in Table 4

The RECS-DB algorithm will create the three index structures shown in Table 5 and the two link list structures shown in Table 6. Note that we do not specify the exact physical design for an index structure, where the only requirement is that the indexed elements are stored in a balanced search tree. We therefore describe a tree index by the elements it indexes, the order of the elements in the tree, and the additional information that is stored at each node of the tree index.

Index X_1 stores the distinct values of the attribute *vulnerability* in the MV `V_COMPUTER` in ascending order. Each node in the index also stores an extra marker 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. The added marker bit allows one to efficiently search for a given *vulnerability* from the table `V_TCP_COMPUTER`. This can be done in logarithmic time because subtrees that do not have the marker bit of their root node set can be pruned out. In general, marker bits allow for the efficient search in predefined subsets of the indexed elements, where a marker bit needs to be defined for each such subset. Marker bits can also be updated efficiently after every insertion, deletion, or modification (see [26]).

For a value with vulnerability k, the index X_1 also contains a pointer to the index $X_3(k)$ and, in addition, it will contain 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 $X(P)$ for which $P = k$.) 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 from `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. Since different records can have different number of fields, a field management technique, such as the one proposed in [40] or [41], needs to be applied.

In order to understand how the data structures from Figures 5 and 6 can be used, consider an instance of query `Q5` from Table 4 with value $P = 5$. The access plan for executing the query will first search for the left-most node in X_1 that contains a *vulnerability* with value greater than 5 that has vulnerability in `V_TCP_COMPUTER`. The marker bits can be used to prune out subtrees that cannot contain such nodes (remember that a marked node means that the node or one of its descendants contains a *vulnerability* that will contribute to the query result). Next, the algorithm will search for the next node to the right that has this property and so on. For each found node, the algorithm needs to scan the key values inside the node and follow the pointer to an X_2 index. The nodes of each visited X_2 index will be scanned in-order and for each such node the pointed by it linked list of W_1 will be contain the tuples that make the query result.

1.4 Related Research

The idea of applying database technology in the development of RECPs is not new. Consider for example the eXtremeDB software ([10]), 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 [19] and TimesTen [31]). However, these systems do not provide realtime guarantees. Note that our solution is also very different than the solution taken by light-weight SQL engines, such as the one implemented in Sybase iAnywhere ([29]). In particular, we do not have a query optimization module and only allow queries that can be processed efficiently.

To summarize, the problem addressed by the RECS-DB system is relatively unexplored. One exception is the paper [35]. 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 (i.e., queries over a single table with only equality constraints in the `where` condition) in the absence of updates. Two other papers ([24, 25]) touch on the problem, but they do not consider the rich set of admissible SQL queries discussed here. Other research on the topic includes [7, 16, 32], but these papers focus more on semantic query optimization rather than providing real-time guarantees. Other papers like [13] describe software patterns that are particular to embedded computer software, but they do not consider writing embedded software code using embedded SQL.

Note that the RECS-DB algorithm solves a problem that differs from the traditional task of automating the physical design for database system. The latter 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 [34] and Microsoft SQL Server [2], 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.

1.5 Paper Outline

In the next section we present definitions that are 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. In Section 5 we present an overview of our test-case study of writing parts of the MINIX operating system using embedded SQL and the RECS-DB algorithm. Section 6 concludes the paper 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 [4]). 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 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 non-reference. We will use $\texttt{attr}(R)$ to denote the attributes of the table R.

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.

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 takes $\sum_{i=1}^{m} log(|T_i|))$ time to return a tuple form the query result, where $\{T_i\}_{i=1}^{m}$ are the underlying tables of Q.*

The query plans for the queries `Q1` and `Q2` from Table 1 produced in Section 1.3.2 are examples of efficient query plans. Conversely, no efficient plan exists for the following query (see [12]), 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 7. Updates `U2` and `U3` from Table 1 are examples of primitive updates, while update `U1` is not a primitive update because it can result in changes to more than one tuple.

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*

(*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.`ID` $= :P$
(3)†‡	`update` T `as` e `set` $e.A = f(e.A)$ `where` e.`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.

Table 7: The five μSQL update types

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 1 is an example of an efficient update that is not a primitive update. As we will see latter, `U1` can be broken down into a sequence of efficient updates.

2.3 Query Languages

In this subsection we describe two SQL dialects: μSQL and sSQL. The first 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 1 are examples of μSQL queries, while queries `Q5` and `Q6` from Table 4 are examples of sSQL 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 all 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. *We will use the symbol* γ *to refer to an efficient predicate.*

For example, the results published in [11] and [21] show that the predicates that can be constructed using the set of integers and a constant 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* $\{R_i\}_{i=1}^r$ *if and only if the formula* θ *is a conjunction of atomic predicates of the form* $R_l.A_k = R_d$.`ID`*, where* $l, d \in [1, r]$*,* $l \neq d$*, and* A_k *is a reference attribute of the table* R_l *that points to the table* R_d*. The* join graph G *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 *if and only if* θ *contains the atomic predicate* $R_l.A_k = R_d$.`ID`*. A join graph is valid if and only if 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 and 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 1) 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". Note that by "induce" we are referring to the graph theory meaning of the work, which implies that the induced graph contains a subset of the original nodes and all the edges between them.

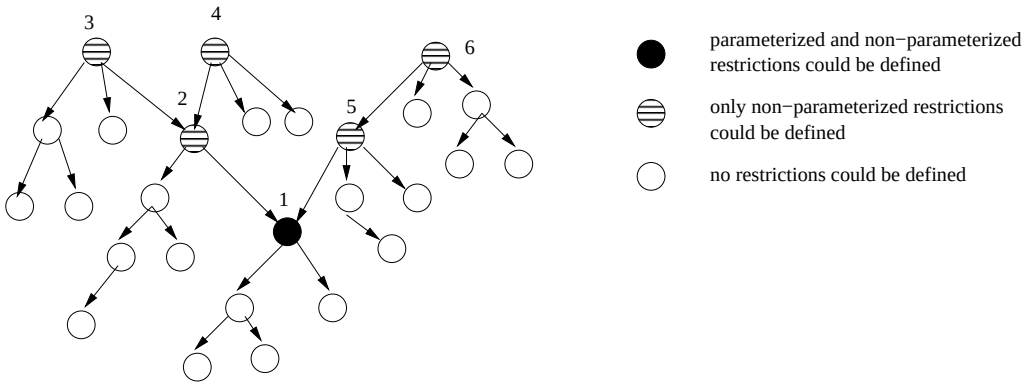

Figure 6: Depicts the shape of the join graph of a μSQL query

Definition 2.5 (tuple ordering and valid tuple ordering in an inverse tree) *A* tuple ordering *is defined using the syntax* $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle \rangle$*, where* R *is a table and* $\{A_i\}_{i=1}^{a}$ *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 if* dir_1 = `desc`*, 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* $\{R_i\}_{i=1}^{k}$*, 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* $\{R_i\}_{i=1}^{k}$*, if and only if the following conditions hold.*

1. *For every* i, $1 \leq i \leq k$*, all the attributes that belong to the same table* R_i *are consecutive in* O*. 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 *(*$1 \leq i \neq j \leq k$*).*
3. *For every* j, $1 \leq j \leq k$*, 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* G^{-t} *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 (i.e., no ordering defined).

The syntax of the sSQL dialect is shown in Table 8, where square brackets are used to denote optional components. Roughly, a sSQL 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 8 in order to make sSQL queries valid. Restriction 3 is added in order to disallow 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 9. The added restrictions extend those for sSQL 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 [26] gives a theoretical proof

(*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.

Table 8: The three sSQL query types

of why μSQL cannot be extended further without braking certain desirable properties for efficiency. 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 = :P1 and c.name = :P2
```

One strategy for efficiently answering the query 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 a 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 (i.e., use two indices and two for-loops as shown in motivating example). 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 = :P1 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

(*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.

Table 9: The three μSQL query types

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 Table 7, 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. Similarly, hash structures are generally faster than indices for partial match queries. 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 traditional index. More specifically, it represents a combination of indices and hash structures.

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 described 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 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 $\texttt{TRUE}$, 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 $\{\texttt{TRUE}\} \cup \{\texttt{FALSE}\} \cup \{ \bigcup_{\emptyset \neq \bar{\gamma} \subseteq \bar{\gamma}(n)} \bigvee_{\gamma \in \bar{\gamma}} \gamma \}$ as the node's extended γ-condition and write $\bar{\gamma}^e(n)$ to denote it.*

The 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 label of a hash node are non-*`ID`*.*

2. *The attributes in the ordering label of a non-leaf nodes and the ordering conditions of a 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` *if and only if* $t \in R$*. Moreover, for any two such nodes* n *and* n'*, all the intermediate nodes along the path from* n' *to* n*, if any, either have the table* R *as the node's table 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.*

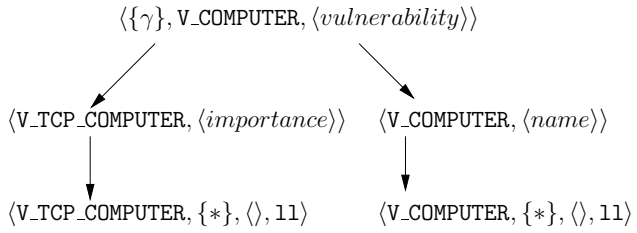

Figure 7: The description tree of the example compact index

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 1.3. In it we produced the compact index that has the description tree shown in Figure 7, 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 5. 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 6, 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: $\langle\{\gamma\}, \texttt{V\_COMPUTER}, \langle \textit{vulnerability}\rangle, [\langle \texttt{V\_TCP\_COMPUTER}, \langle \textit{importance}\rangle, [\langle \texttt{V\_TCP\_COMPUTER}, \{*\}, \langle\rangle, \texttt{11}\rangle]\rangle, \langle \texttt{V\_COMPUTER}, \{\textit{name}\}, [\langle \texttt{V\_COMPUTER}, \{*\}, \langle\rangle, \texttt{11}\rangle]\rangle]\rangle$.

Note that a root node with label $\langle \texttt{V\_COMPUTER}, \{\textit{vulnerability}\}\rangle$ describes a hash structures that stores the distinct values for *vulnerability* of the table `V_COMPUTER`. Each distinct *vulnerability* value is mapped to a one or more data structures described by the children of the root node. Since queries `Q5` and `Q6` from Table 4 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 $\langle \texttt{V\_TCP\_COMPUTER}, \{*\}, \langle \textit{name}\ \texttt{asc}\rangle, \texttt{11}\rangle$. This denotes that the linked lists W_2 from table 6 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 `ID`s to physical addresses. Restriction 2 guarantees that the ordering conditions and ordering labels define valid orders in terms of Definition 2.5. 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 tuples from the

data structure represented by one of the leaf nodes of the subtree with root n. As shown in [26], 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 marker 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 *marker bits, where the* $i^{\texttt{th}}$ *marker bit is set exactly when the node or one of its descendants contains a record that passes the condition* γ_i*.*

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 [26], 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* $\mathcal{L}(n)$*.*

Next, consider a leaf node n *that is not the only node in the description tree. Suppose that* $\{D_i\}_{i=1}^d$ *are the attributes in the ordering condition of* n *union the attributes in the ordering labels of its ancestors in the description tree. Then* n *will represent a set of singly linked lists (or doubly linked lists 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*, if and only if* 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 *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* $i^{\texttt{th}}$ *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* $A_1,\ldots,A_a$*.*

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., tuple 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 [40].*

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-1} and the ordering condition of n_k in this order, where only the attributes in the ordering label of a hash node can be permutated.*

Definition 3.7 (extended label) *Let G^t be a node labeled tree with node labels of the form $\langle \ldots \rangle$. Let $\mathcal{L}^{\uparrow}$ be a new node labeling function that produces the extended label of a node. For the root node n^r, we define $\mathcal{L}^{\uparrow}(n^r) = \texttt{label}(n^r)$. For a non-root node n with label $\langle L \rangle$ and parent node n' with extended label $\langle L' \rangle$ we defined $\mathcal{L}^{\uparrow}(n) = \langle L', L \rangle$.*

Informally, the extended label of a node is a listing of the labels of the nodes of the path that starts at the root node of the tree and ends at the node.

Definition 3.8 (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 10 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 the condition that the attributes in the `where` and `order by` clause of Q are from the attributes along the path K, where only the attributes in a hash structure can change place. The reasoning 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 4 because there is no way to efficiently enumerate the records of 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 cannot be used to efficiently retrieve the elements in order.

Definition 3.9 (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 3.10 *Let J be a compact index. Then the sets $\bigcup_{K \in \texttt{cp}(J)} Q_K(J)$ and $Q(J)$ are equivalent.*

Proof: See the extended version of this paper ([28]).

The proof of the theorem contains detailed pseudo-code showing how compact indices can be used to efficiently answer sSQL queries. The next theorem will describe how compact indices can be refreshed after a 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 3.11 ((admissible primitive updates)) *Given a compact index J, we will say that the primitive update U is* admissible *for J if and only if 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.*

(*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} $dir_{l+1}^w, \ldots, A_a$ dir_a^w]
(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} $dir_{l+1}^w, \ldots, A_a$ dir_a^w]
(3)	`select` $D_1, \ldots, D_d$ `from` R_{k+1} `where ID` $= :P_1$

1. $R_{k+1} =$ `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 `type`$(n_{k+1}) =$ `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 $dir_i^{-1} =$ `asc` if $dir_i =$ `desc` and $dir_i^{-1} =$ `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 `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 $t \in$ `table`(n_i) exactly when there exists a tuple $t' \in$ `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 label of a non-leaf node along the path K, then the node must be an index node.

Table 10: Queries for a complete path $K = \langle n_1, \ldots, n_{k+1} \rangle$

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.*

The reason for the first restriction is that insertions cannot be performed efficiently in an ordered linked list because one needs to first find the place where to do the insertion. The second restriction is due the fact that a specified node cannot be deleted from a linked list without having a pointer to the previous node in the list. The third restriction says that either the modification does not change the order of the elements in the list, or it can be performed as a an efficient deletion followed an efficient insertion.

Theorem 3.12 *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 : See the extended version of this paper ([28]).

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 μSQL query values from $\{R_i\}_{i=2}^r$ can be gathered by executing the following query Q:

```
select A_1, ..., A_a
from R_1 as x_1, ..., R_r as x_r
where θ(x_1, ..., 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 corresponds to R_i is:

```
select x.B_1, ..., 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' ∈ child^θ(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 to the resulting set 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 the 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 an outer `for` 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 9. 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, ..., D_d
from R_1 as e_1, ..., R_r as e_r
where θ(e_1, ..., e_r) and γ_1(e_1) and ... γ_k(e_k) and
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}, ..., 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 *
from R_i as e
where γ_i(e).
```

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

```
select *
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.

```
select count(*)
from R_1 as e_1, ..., R_r as e_r
where θ(e_1, ..., e_r) and γ_1(e_1) and ... and γ_{r-1}(e_{r-1}) and e_r.ID= t.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:

```
select count(*)
from R_j as e, R'_1 as e_1, ..., R'_m as e_m
where θ'(e, e_1, ..., e_m) and γ_j(e) and γ'_1(e_1) and ... and γ'_m(e_m) and e.ID= :t.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.

```
select sum(C_j)
from R_j as e
where γ_j(e) and e.Pf^θ_(j,i) = t.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.

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, ..., A_a
from T as e
where γ(e)
```

Consider a MV V of this type 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 if and only if $\gamma(t)$ 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 if and only if $\gamma(t^{\texttt{old}})$ does not hold, but $\gamma(t^{\texttt{new}})$ does,
2. a deletion of the tuple with `ID` t.`ID` from V if and only if $\gamma(t^{\texttt{old}})$ 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 if and only if $\gamma(t^{\texttt{old}})$ and $\gamma(t^{\texttt{new}})$ both hold.

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 7 are already primitive updates. Consider an update of the forth type, that is, an update that has the following general syntax.

(*type*)	(*compact index*)
(1)	$\langle R, \{A_1, \ldots, A_l\}, [\langle R, \{D_1, \ldots, D_d\}, \langle A_{l+1}\ dir_{l+1}, \ldots, A_a\ dir_a\rangle, \texttt{11}\rangle]\rangle$
(2)	$\langle R, \{A_1, \ldots, A_l\}, [\langle R, \langle A_{l+1}\rangle,$ $[\langle R, \{D_1, \ldots, D_d\}, \langle A_{l+2}\ dir_{l+2}, \ldots, A_a\ dir_a\rangle, \texttt{11}\rangle]\rangle]\rangle$
(3)	$\langle R, \{D_1, \ldots, D_d\}, \langle\rangle, \texttt{11}\rangle$

Table 11: The compact indices for the three sSQL query types

```
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 sets.

An update of fifth type (see Figure 7) 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 11 shows how to convert a sSQL from each of the three types to a compact index.

Consider query `Q5` from Table 4. 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 given compact index can answer.

Definition 4.1 (cover of a compact index) *Let J be a compact index. We define* `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 11.

Theorem 4.2 *If the translation rules from Table 11 produce the compact index J from the query Q, then* `cover`(J) *contains exactly the set of compact indices that can be used to efficiently answer the query Q.*

Proof: See [28].

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 required is because only admissible primitive updates (see Definition 3.11) can be performed on a compact index efficiently (see Theorem 3.12).

Consider a compact index J and a leaf node n in the description tree 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 n. 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, then 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 down 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 3.10 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 3.12 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 11 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 tree 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 mergeable 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 [8]). 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 and MVs over this table are clustered together. Compact indices from different clusters will be based on different attributes and will not be mergeable (see Definition 4.3).

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

```
00 compact_indices merge(compact_indices J̄) {
01   result = ∅;
02   result_size = ∞;
03   let J̄'' be the indices in J̄ with single node description
     trees;
04   J̄ = J̄ − J̄'';
05   for {J̄'}_{i=1}^m ∈ find_clustering(J̄) {
06    for i = 1 to m {
07     if (merge_into_one(J̄'_i) = NULL)
08      skip to next iteration of the outer for loop;
09     J_i^r = merge_into_one(J̄_i);
10    }
11    if calculate_size({J_i^r}_{i=1}^m ∪ J̄'') < result_size {
12     result = {J_i^r}_{i=1}^m ∪ J̄'';
13     result_size = calculate_size({J_i^r}_{i=1}^m ∪ J̄'');
14    }
15   }
16   if result = ∅, then return J̄, else return result;
17 }


00 compact_index merge_into_one(compact_indices J_1, ..., 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 = union_{i=1}^k R_i;
06   γ̄ = ∅;
07   J̄' = ∅;
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(J̄′)]⟩;
20    else
21      return ⟨γ̄, R, ⟨A⟩, [merge(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 ([6]), 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 [8] 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 prefix of the ordering labels of the nodes, where the attributes of a hash node can be permutated. 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 `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 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: See [28].

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 describe 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 in groups. In this approximate algorithm we will execute this method differently, where the rest of the algorithm will not change. In particular, 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 into the first cluster and then the procedure is applied on the remaining compact indices. The procedure is repeatedly applied until all input compact indices are assigned to clusters.

Note that the method `find_clustering` will now create a single clustering and therefore the `for` loop at Line 5 of the `merge` method will 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 Case Study

We have performed case studies to show:

1. To what extent will the resulting solution be suboptimal if the approximate algorithm for merging compact indices is applied;
2. When a typical RECP workload is rewritten using embedded SQL, what percentage of the resulting SQL queries will be μSQL queries;
3. How much storage can be saved by using the presented compact indices on a typical RECP; and
4. What is the performance benefit of merging.

We believe these studies strengthen our premiss about the benefits of applying the RECS-DB algorithm for the creation of RECPs.

TPC ([33]) is the typical transaction workload that is used to measure the performance a database management system. In particular, TPC-C is a workload that consists of simple queries and frequent updates and is used to simulate the typical *online transaction processing* (OLTP) workload. We believe that this workload closely resembles the typical RECP workload and in [24] we have done experimental studies that show how index merging can be beneficial for such workloads. In this paper we go a step further by considering the Embedded Control Workload ([39]) - a workload that is specifically designed for embedded control programs. The workload is based on the MINIX operating system described in [30]. We follow [39] and consider only the UNIX operations `fork`, `exit`, `waitpid`, `exec`, and `brk`, where the probability of occurrence of each operation is shown in Table 12. These operators are selected because they are well known and are highly related to data structures that represent process and memory chunk descriptions.

We also simplify some of the functionality of the five system operations. In particular, we flatten the system call processing to one layer (there are four layers in MINIX). From the kernel layer, we keep the code that maintains the ready queue, but ignore the processing of the register values. From the file system

(*transaction type*)	(*probability*)
`fork`	11.7%
`exit`	11.7%
`waitpid`	9.3%
`exec`	9.3%
`brk`	58%

Table 12: The probabilities of the transaction types

server, we take the part that processes the file descriptors and the code for duplicating the file descriptors in `fork` and closing the files in `exit` and `exec`. We also combine the process descriptor data from the different layers, where more details are presented in [39].

We consider the C language code for the five UNIX operations (for the actual code see [30] or [39]). This C language code can be rewritten as *embedded static SQL code* (see [39]), that is, code that contains static SQL statements (i.e., SQL statements that are not dynamically generated) that are embedded in C language code.The SQL code is based on the tables `segment`, `proc`, `filp`, `proc_filp`, `hole`, and `ready_user_proc`. The `segment` table contains information about the segments, where a segment can be a data segment, a code segment, or a stack segment. The `proc` table contains information about the processes. The `filp` (stands for file pointer) table contains the file descriptors of the opened files. The `proc_filp` table contains the mapping between the processes and the file descriptors of the files that are in use by the processes. The main memory in MINIX consists of *holes* and the table `hole` holds information about the holes in main memory that are free. Lastly, the `ready_user_proc` table contains a list of processes that are ready to run (i.e., the processes that from the waiting queue).

We next describe the results of our case studies.

5.1 Quality of Approximate Merging Algorithm

For the selected MINIX workload, there is no difference between the approximate and exact merging algorithms. There are four index merging opportunities and both algorithm select the tree possible merges that will produce the compact indices of the smallest size. In order for the exact algorithm to outperform the approximate index merging algorithm, it must be the case that several choices for index merging exist.

5.2 Percent of μSQL Queries

The MINIX workload consists of 103 SQL queries, where 96 (or 93%) of them are μSQL queries. The queries that are not μSQL queries cannot be executed efficiently, but can be rewritten as C code embedded with μSQL queries. This shows that it is reasonable to restrict the type of queries that are allowed as input to μSQL queries and to manually break queries that are not in this dialect into μSQL queries.

5.3 Amount of Storage Saved

We follow the ratio of the size of the tables recommendation in [39]. However, we multiply the size of each table by 10 in order to get reasonable size for the tables (Creating index and hash structures will significantly improve performance only for reasonably size tables (e.g., more than 20 tuples per table)). In particular, we assume that there are 320 processes, 960 segments (i.e., a text, data, and stack segment for every process), 1280 open file descriptors, 1280 holes of free main memory, and 50 processes that are stopped, but are ready to run. We also assume that there are 10 files opened by each process. This translates to the data shown in Table 13. The table also shows the size of a record of each type, where we store an integer in 2 bytes and a time stamp in 10 bytes. When populating the data we will assume that that half of the processes have child processes. We have used AA trees ([3]) for the index structures (this requires the level of each tree node to be stored in the node), where we have extended these trees with the addition of marker bits. We have used separate chaining ([38]) to implement the hash structures, where we assumed that initially 70% of the hash table is full and the size of the hash table is resized when 75% of the table becomes full. Our implementation is in GNU C++ under Cygwin ([9]). Figure 8 shows the total size of the original control

(*table*)	(*number of tuples*)	(*tuple size*)	(*total size*)
`segment`	960	8B	7,680B
`proc`	320	34B	10,880B
`filp`	1,280	10B	12,800B
`proc_filp`	3,200	4B	12,800B
`hole`	1,280	4B	5,120B
`ready_user_proc`	50	2B	100B
total			49,380B

Table 13: The created tables for the MINIX workload

data, the size of the control data if appropriate indices and materialized views are used for each query, the size of the control indices if compact indices and materialized views are used, and the size of the control data after the merging of compact indices is applied. The initial size of the compact indices is smaller than the initial size of the regular indices because the created hash structures in the compact indices take less space than the corresponding search trees in the regular indices.

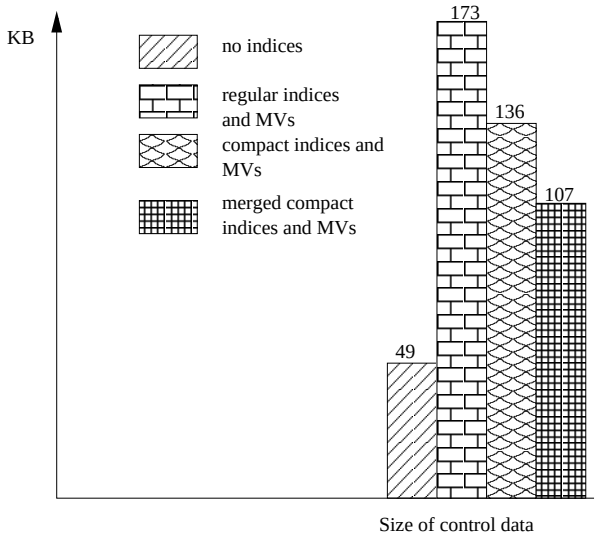

Figure 8: Initial control data size

5.4 Performance benefit on index merging

The experiments were conducted under the Cygwin operating running on a dual core 64 bit AMD Turion CPU running at 1900Mhz. We ran the Embedded Control Program Workload([39]) and have used the frequencies of the different operations shown in Table 12. A transaction executes one of the five operations with the described probability. We ran our experiments for 2 hours. Figure 9 shows the number of transactions per second for the three different physical designs from Figure 8. The solutions that use compact indices significantly outperform the one that does not because of the use of hash structures. The solution with merged compact indices outperforms the solutions that does not apply the merging algorithm because the updates in the workload are predominant (85 of the 103 queries in the workload are update queries). Our merging procedures is beneficial when the update ratio is high because fewer redundant data has to be modified at each update, which makes the updates faster.

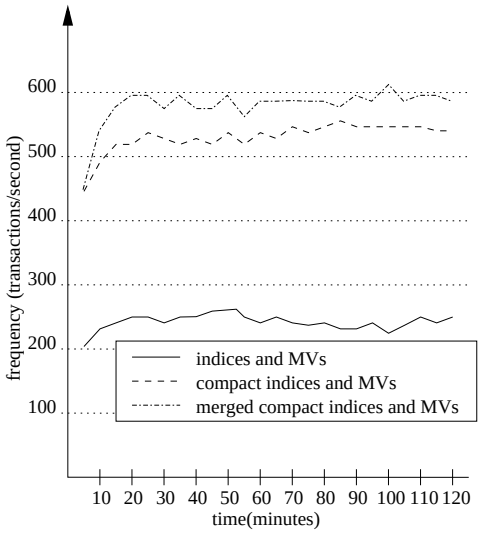

Figure 9: Throughput for the three different data structures

6 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. The case studies that we have conducted show empirically the benefits of our approach, that is, our approach can be applied to the development of a RECP in a way that saves storage and makes updates faster.

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 environment. 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.

APENDIX

A Summary of Notation

A attribute of a table

$\texttt{attr}(R)$ the attributes of the table R

B attribute of a table

C attribute of a table

c constant

$cp(G^t)$ the set of complete path in the three G^t

D attribute of a table

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

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

$f(\cdot)$ function over the enclosed components

G graph

G^t tree

`ID` a global tuple identifier

J compact index

K path in a graph

$\texttt{label}(n)$ the label of the node n

n^r root node in a tree

$O(\cdot)$ big O notation from complexity theory

P query parameter

Pf path function

$Pf^G_{n_1,n_2}$ the sequence of labels along the directed path from n_1 to n_2 in the graph G; result is `ID` when $n_1 \equiv n_2$

Q query

Q_P a query plan for the query Q

R table

T base table

$type$ constant in the set $\{\texttt{ll}, \texttt{dll}\}$

U update

U_P an access plan for the update U

V materialized view

W linked list

X regular index

$\mathcal{L}^{\uparrow}(\cdot)$ node labeling function – see Definition 3.7

$\mathcal{L}^{\downarrow}(\cdot)$ node labeling function – see Definition 3.1

$\theta(\ldots)$ valid join condition

γ efficient predicate

$\gamma(\cdot)$ result of applying the predicate γ over the enclosed component; result is Boolean value

σ selection in relational algebra

π projection in relational algebra

π^d duplicate preserving projection in relational algebra

$\Pi(\cdot)$ permutation that can change the order of the attributes in a hash node (see Definitions 3.6)

$|\cdot|$ size of the enclosed component

$\{\ldots\}$ a set of elements

$\langle\ldots\rangle$ a list of elements

$\{|\ldots|\}$ a bag of elements

References

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

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. *VLDB*, pages 496–505, September 2000.

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

[4] François Bancilhon and Guy Ferran. ODMG-93: The Object Database Standard. *IEEE Data Eng. Bull.*, 17(4):3–14, 1994.

[5] Bayer and McCreight. Organization and Maintenance of Large Ordered Indexes, 1972.

[6] Eric Temple Bell. Exponential Numbers. *Amer. Math. Monthly*, 41:411–419, 1934.

[7] Peter Chan. Optimizing OQL on Legacy Main Memory Data Structures with Existential Graphs. *University of Waterloo Master's Thesis*, 1997.

[8] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards estimation error guarantees for distinct values. *PODS*, pages 268–279, 2000.

[9] Cygwin, http://cygwin.com. *Cygwin.*

[10] eXtremeDB, www.mcoject.com. *eXtremeDB User's Guide.*

[11] Jeanne Ferrante and Charles Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM J. Comput*, 4(1):69–76, 1975.

[12] Michael L. Fredman. A Lower Bound on the Complexity of Orthogonal Range Queries. *J. ACM*, 28(4):696–705, 1981.

[13] David Gay, Philip Levisand, and David Culler. Software Design Patterns for TinyOS, 2007.

[14] Lukasz Golab and Tamer Özsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. *SIGMOD*, pages 658–669, 2005.

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

[16] M.Y.S. Lai. On integrating legacy main meomory data structures with database schema. 1999.

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

[18] L.J.Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, 8(21), 1978.

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

[20] Jun Rao and Kenneth Ross. Making B^+ Trees Cache Conscious in Main Memory. *ACM SIGMOD*, pages 475–486, 2000.

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

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

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

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

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

[26] Lubmoir Stanchev. Automating Physical Design for an Embedded Control Program. *Ph.D. thesis*, 2005.

[27] Lubomir Stanchev. Reducing the Size of Auxiliary Data Needed to Support Materialized View Maintenance in a Data Warehouse Environment. *Eighteenth International Resource Management Association Conference*, 2007.

[28] Lubomir Stanchev and Grant Weddell. Programming Embedded Computing Systems using Static Embedded SQL. *Indiana University - Purdue Universitcy Computer Science Technical Reoprt CS2008-01, http://www.cs.ipfw.edu/reports/2008/ecp.pdf*, 2008.

[29] Sybase, http://www.sybase.com/ianywhere. *Sybase iAnyWhere*.

[30] Andrew Tanenbaum and Albert Woodhull. *Operating Systems: Design and Implementation*. 3rd edition edition, 2006.

[31] TimesTen, www.timesten.com. *TimesTen User's Guide*.

[32] David Toman and Grant Weddell. Query Processing in Embedded Control Programs. *Proceedings of Databases in Telecommunications*, pages 68–87, 2001.

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

[34] Gray Valentin, Michael Zulian, Daniel C. Zilio, Guy Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes. *ICDE*, pages 101–110, February 2000.

[35] Grant Weddell. Selection of Indexes to Memory-Resident Entities for Semantic Data Models. *IEEE TKDE*, 1(2):274–284, 1989.

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

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

[38] W.W.Peterson. Addressing for Random Acess Storage. *IBM Journal of Research and Development*, 1:130–146, 1957.

[39] Heng Yu and Grant Weddell. Building and Embedded Conntrol Program Workload. *University of Waterloo techical report: CS-2005-04, http://www.cs.uwaterloo.ca/research/tr/2005/04/report.pdf*, 2005.

[40] Yoav Zibin and Joseph Yossi Gil. Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-dispatching. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 142–160, 2002.

[41] Yoav Zibin and Joseph Gill. Two-Dimensional Bi-Directional Object Layout. *ECOOP 2003*, pages 329–350, 2003.