Automating Physical Design for an Embedded Control Program

by

Lubomir P. Stanchev

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, February 2005

I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Lubomir P. Stanchev

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Lubomir P. Stanchev

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Acknowledgments

I would like to thank ...

Abstract

A *compiled database application* consists of a collection of modules in a software system that interacts with a common database through a set of predefined transaction types. We will call a compiled database application an *embedded control program* (ECP) if quality of service requirements could be specified for some of the database operations in the different transaction types. Usually, the common database of an ECP is referred to as the *control data*. In this thesis we consider the problem of automating the physical design for the control data of an ECP. In particular, we show how to create, query, and update the data structures for the part of the database on which logarithmic time requirements are specified. Unlike the objectives of earlier works that aim to minimize the response time of the queries and updates in a workload, our goal is to minimize the storage requirements subject to the constraint that enough information is stored to allow for executing every database operation in the specified time-bound. Since not every SQL operation could be executed in logarithmic time, we introduce a subset of SQL that defines what constitutes a valid database operation on which a logarithmic time-bound requirement could be imposed.

Our first task in solving the problem is fixing the physical design space. In particular, we explore two novel physical design encodings. The first one extends traditional indices by allowing records with different attributes to be indexed together. It also introduces branching ordering conditions (i.e, different orders can be specified on different disjoint subsets of the indexed records) and it supports marking bits that can be used to efficient traverse predefined subsets of the indexed records. The second encoding consists of compact structures that have the capability of representing several indices with similar ordering conditions. Both physical designs have the desired compact representation property because both encodings can be used to reduce the need for storing redundant data.

We then construct a solution for each of the physical design models. Since the problem of finding the smallest possible encoding is NP-Hard for both physical designs, we present exponential time exact algorithms and polynomial time approximate algorithms for solving the problem. The conducted experiments with the compact structure encoding show that our approach is capable to finding small physical structures that support fast execution of an OLTP like workload in which a logarithmic time-bound is specified on all database operations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past thirty years database technology has spread into almost every corner of software development. However, for the most part, engineers of embedded control programs have remained reluctant to adopt this technology for the following reasons: (1) the lack of performance guarantees, (2) the large footprint size of *database management systems* (DBMSs), and (3) the large size of the constructed physical design. The world around us is filled with examples of embedded systems: a digital camera, a traffic light, a hand-held device, a network router, to name a few. In the last decade, the introduction of the Internet and hand-held device has fueled the need for creating new and diverse embedded control applications. As a result, more emphasis has been put on automating parts of the development process of an embedded control programs, including exploiting database technology. In this thesis, we address the lack of performance guarantees for DBMSs and the unnecessarily large amount of memory often required to encode the database by considering how one can construct a database engine with limited capabilities that can meet the performance and space requirements for an embedded control program.

The Wikipedia ([40]) encyclopedia defines an *embedded system* as : *A* **special-purpose** *computer system built into a larger device that is typically required to meet very different requirements than a general-purpose personal computer.* The key requirements of an embedded system include:

- reliability,

- conformity to real-time requirements,
- ease of maintenance.

We will refer to the software that governs the behavior of an embedded system as an *embedded control program* (ECP).

An ECP can be viewed as collection of modules responsible for managing the *control data* of a software system (see Figure 1.1). The interaction between an ECP and its control data is characterized in terms of a **predefined** collection of transaction types, where each transaction type consists of data access operations embedded in a programming language with flow control.

In this thesis we explore how the performance requirements of an ECP can be satisfied when deploying database technology. In particular, in an ECP a time limit can be specified on the execution time of each transaction type, and therefore on the execution time of each data access operation. Note that a real-time requirement means that the execution of a task is considered *correct* only when: (1) the correct actions are performed, and (2) those actions are performed within the specified time limit.

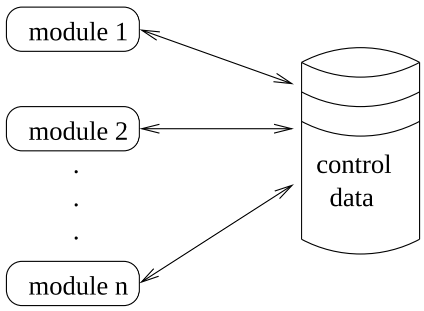

Figure 1.1: An ECP

The software engineer can benefit if the data access operations are expressed in terms of a high-level declarative database language (such as SQL) that references the control data at the logical level (e.g., expressed using a relational representation). The reasons are two-fold. First, there is no need to manually map the logical data design to a physical design. The assumption is that the DBMS can perform this task and produce a physical encoding of the data that allows for executing of all database operations in the required time-bound. Second, the programmer does not need to create low-level data access plans. This makes the code simpler, faster to write, and easier to understand and maintain. The onus is once again on the DBMS to create this code and ensure that all operations execute within the required time-bound.

Preliminary test-case studies on adopting database technology in the design of ECPs confirm our believe in the benefits of the presented approach. Consider for example the Linux kernel case described in [32]. It shows how this operating system can be viewed as an ECP. Expressing the interaction with the control data of the system using a database language not only helps the software developer, but also yields better performance. The reason is that the simplicity of this approach allows for the creation of efficient code that can outperform even code written by expert programmers.

In this thesis we adopt the scenario shown in Figure 1.2. Note that the DBMS in the figure must be able to provide real-time guarantees. The actual code for the DBMS is constructed during a *pre-compilation* phase, i.e. its footprint size is small and it is only capable of handling the predefined transaction workload.

In order to make our model more concrete, we assume the following:

1. The modules see the control data at the logical level expressed as a relational database;

2. The modules interact with the data via SQL statements; and

3. The flow control in each transaction type is encoded using the **C** programming language.

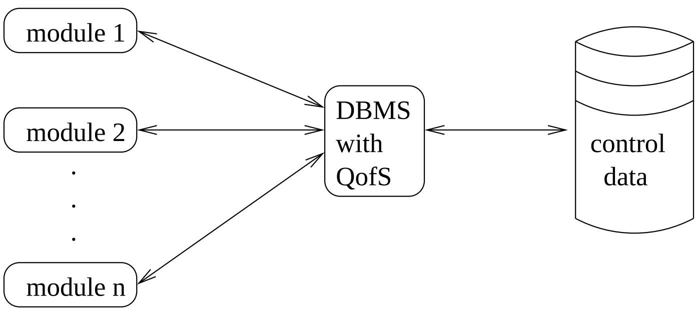

Figure 1.2: Adopting DBMS technology to ECPs

In this thesis, we consider real-time requirements that correspond to *logarithmic* time-bounds for executing queries and updates. We will refer to queries and updates on which such requirements are imposed as *critical* and to queries and updates on which looser or no time requirements are specified as *non-critical*. We have chosen the logarithmic time-bound for critical operations because this bound corresponds to the time it takes to probe an ordered tree index.

Other bounds that are also worth exploring include linear and doubly-logarithmic time. The set of queries and updates that can be answered in linear time relative to the size of the database is quite large. This time-bound is useful for imposing time constraints on database operations that are not critical, but which need to be executed in reasonable time. Conversely, data structures that can produce results in doubly-logarithmic time for a restricted set of queries and updates also exist (e.g. *y-fast tree* [38], *interpolation search trees* [16]). Such a time-bound is useful for imposing constraints on super-critical database operations for which the logarithmic time-bound does not suffice.

Note that in our solution to the problem the constant in front of the logarithmic time-bound is small in all cases and depends on the size of database schema description, i.e. the proposed access plans for executing critical operations can indeed be considered correct in the sense of soft real-time requirements.

A critical database operation can be either a *critical query* or a *critical update*. We require from critical queries that it takes roughly logarithmic time, relative to the size of the database, to return a tuple from the query result. We impose a similar time bound for critical updates. In particular, a *primitive update* is an update that inserts a tuple in a base table, deletes a tuple from a base table, or modifies the content of a tuple. We require that all primitive critical updates can be executed in roughly logarithmic time also in the size of the database. We will refer to an access plan that can execute a critical operation in the required time-bound as an *efficient* access plan[1].

Consider the following query Q:

$\texttt{select}\ sum(C)\ \texttt{as}\ S$
$\texttt{from}\ T$
$\texttt{where}\ T.A > :P_1 \text{ and } T.B > :P_2,$

[1]The precise time requirements for the critical database operations are presented in Chapter 2

where the capital letter P, possibly parameterized, is used throughout the paper to denote a query parameter. Michael Fredman in [12] has shown that $\Omega(N \cdot log^2(N))$ is a lower bound on the worst case time requirement for a sequence of N operations involving insertions, deletions, and two dimensional orthogonal range queries. In other words, there does not exist a physical design $\mathcal{D}$ for which the following statements are true:

1. Any primitive update to the table T can be efficiently performed on $\mathcal{D}$; and

2. There exists a query plan for Q over $\mathcal{D}$ that is efficient.

Now, consider a database schema in which no uniqueness constraints can be inferred. We will call the subset of SQL that can be efficiently answered using a physical design on which every possible primitive update can be executed efficiently *critical SQL* (cSQL) (see Figure 1.3). The above query Q is one example of a query that is outside the cSQL set.

We find a lower bound for the cSQL set by starting with the set of queries that can be efficiently answered using a single regular *endogenous index* and we call this set *simple SQL* (sSQL) (see Table 4.3 and Figure 1.3). (Note that an *endogenous index* is an index that stores the records being indexed, rather than record pointers. An index that stores pointers to the indexed elements as called *exogenous*.) We then expand this set to create the μSQL set, which is still bounded by the cSQL set.

Each query in the μSQL set has the property that it can be efficiently answered using a *perfect nested loop join* of sSQL queries that could potentially reference newly introduced materialized views[2] (MVs), which we require to be efficiently maintainable. (The μSQL set is described in Table 2.2). The innermost operation of the nested loop is the *send* operation that outputs the attributes of a single tuple of the query result. A perfect nested loop join is a nested loop join that references only tuples that contribute to the query result. Efficiently maintainable MVs and efficiently maintainable derived attributes are entities on which any primitive update can be propagated efficiently. It turns out that μSQL is quite expressive. In particular, 92% of the TPC-C workload ([34]) requirements can be directly translated in μSQL. Moreover, μSQL is the biggest subset of the *examined critical* SQL (eSQL) set

[2]Note that the we use the term *materialized view* to denote data that is materialized only when used by queries and/or updates, e.g. it may be the case that the data in a materialized view is not physically stored.

that has the property that if $Q \in \mu$SQL, then Q can be efficiently answered using $|def(Q)|_t$ regular endogenous indices that can be efficiently refreshed after every critical update and constant amount of main-memory (for a proof see Section 5). Note that throughout this work we use $|\cdot|$ to denote the physical size of the enclosed component and $|def(\cdot)|$ to denote the size of its definition. For example, $|T|$ denotes the number of tuples in the table T times the size needed to store a tuple, while $|def(Q)|$ denotes the size of the definition of the query Q. On the other hand, the expression $|def(\cdot)|_t$ is used to denotes the number of tables referenced in the definition of the enclosed component. The described "biggest" property is relative to a certain set of integrity constraints that does not include uniqueness constraints. Considering uniqueness constraints such as keys and functional dependencies will invalidate this property and require new ways of characterizing μSQL, which is an area for future research.

Consider the following query Q.

select $*$
from R_1, R_2
where $R_1.A = R_2.$ID and $R_1.B = {:}P_1$ and $R_2.C = {:}P_2$

We shall use the system attribute ID to denote a tuple identifier, where every table has this system attribute; its semantics and precise implementation is discussed in Chapter 2. For the above query, we assume that A is a non-NULL attribute of R_1 that always references a tuple in R_2. As shown in Chapter 5, the above query cannot be efficiently answered using $|def(Q)|_t$ regular endogenous indices that can be efficiently maintained in the absence of uniqueness constraints and a constant amount of main-memory. The reason is that an index scan on some of the tuples in R_1 or an index scan on some of the tuples in R_2 could result in scanning lots of tuples that do not contribute to the query result. On the other hand, a join index on the tuples in R_1 and R_2 is not efficiently maintainable.

Conversely, the above query can be efficiently answered if we knew that either B is a candidate key for R_1 or C is a candidate key for R_2. In the first case, we can first scan an index on R_1 to find the single tuple in R_1 for which $R_1.B = {:}P_1$, then use an index on R_2 to find the corresponding tuple in R_2, and lastly return the join of the two tuples. In the second case, we can first scan an index on R_2 to find the single tuple in R_2 for which

$R_2.C = :P_2$, then we can use an index on the attributes A and B of R_1 to find the tuples in R_1 that will contribute to the query result, and lastly return the join of the found tuples.

To summarize, the above query Q is an example of a query that is not a μSQL query. However, we know that either B is a candidate key for R_1 or C is a candidate key for R_2, then Q can be efficiently answered using at most two indices. Extended μSQL to include query such as Q is an area for future research. Note as well that the problem of whether Q is a cSQL query in the absence of functional dependencies is an open research question to the best of our knowledge.

The eSQL set (see Figure 1.3) is roughly conjunctive SQL with ordering, grouping, and aggregation[3]. When we defined μSQL, we intentionally stayed within the eSQL boundary in order to keep the problem tractable. In particular, we tried to make the shaded area in Figure 1.3 as small as possible without going beyond queries Q that require more than $|def(Q)|_t$ regular indices and constant amount of main-memory in order to be efficiently answered.

However, the shaded area is not empty. Consider for example the following query Q.

> `select` C
> `from` T
> `where` $T.A > :P_1$ and $T.B > :P_2$

Worm Mortensen in [25] has shown that the query can be efficiently answered using an efficiently maintainable physical design that uses van Emde-Boas priority ques ([36]). However, the query is not in μSQL because it cannot be answered using $|def(Q)|_t$ regular endogenous indices.

It turns out that the other areas in Figure 1.3 are also non-empty. Table 1.1 defines the queries Q_1, Q_2, Q_3, Q_4, and Q_5 from Figure 1.3, which are instances of queries in each of the five non-shaded areas. Note that in order for the query Q_2 to be in μSQL, it must be the case that the attribute A is a non-`NULL` attribute of the table R_1 which always reference a tuple in the table R_2.

[3]The precise definition of eSQL is given in Table 2.3

(*name*)	(*query*)
Q_1	`select` $*$ `from` R `where` $A = {:}P_1$ and $B = {:}P_2$
Q_2	`select` $*$ `from` R_1, R_2 `where` $R_1.A = R_2.$`ID`
Q_3	`select` $sum(C)$ `from` R `where` $A > {:}P_1$ and $B > {:}P_2$
Q_4	`select` $*$ `from` R_1 `where` $A = {:}P_1$ `union all` `select` $*$ `from` R_2 `where` $B = {:}P_2$
Q_5	`select` $sum(A_3)$ `as` $sumS$ `from` R_1 `where` $A_1 > {:}P_1$ and $A_2 > {:}P_2$ `union all` `select` $sum(B_3)$ `as` $sumS$ `from` R_2 `where` $B_1 > {:}P_3$ and $B_2 > {:}P_4$

Table 1.1: The five queries of Figure 1.3

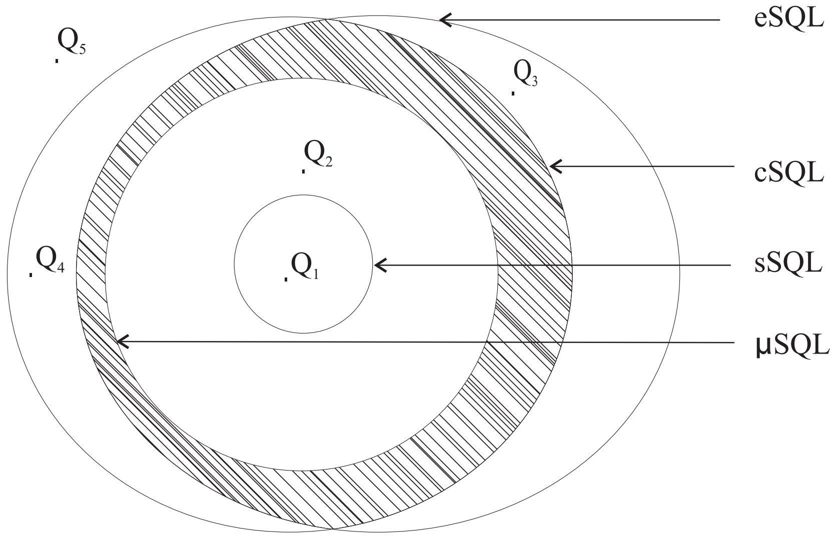

Figure 1.3: Four different subsets of SQL

1.1 Problem Overview

We examine the problem of automatically generating the code for a DBMS with real-time requirements that is used in the scenario shown in Figure 1.2. We will refer to this problem as the *Embedded Control Physical Design* (ECPD) problem. In particular, we are given as input a set of static, possibly parameterized, critical queries and updates, a set of non-critical queries, information about the database schema, and statistics on an instance of the control data (see Figure 1.4). The goal is to perform a pre-compilation step that produces a physical design and access plans for all critical operations and for the parts of the non-critical queries that access critical data (i.e. data accessed by either a critical query of a critical update). Note that the generated execution code for the DBMS is quite restrictive, i.e. it can handle only the specified workload on the critical data. The benefit is its small footprint size and its guaranteed logarithmic performance for each of the specified critical queries and updates. It is not a general purpose DBMS because it cannot handle requests outside the predefined workload. In this work we present the DBMS execution pseudo-code using the **C** programming language.

Our solution to the problem abstracts whether the proposed physical design is stored in

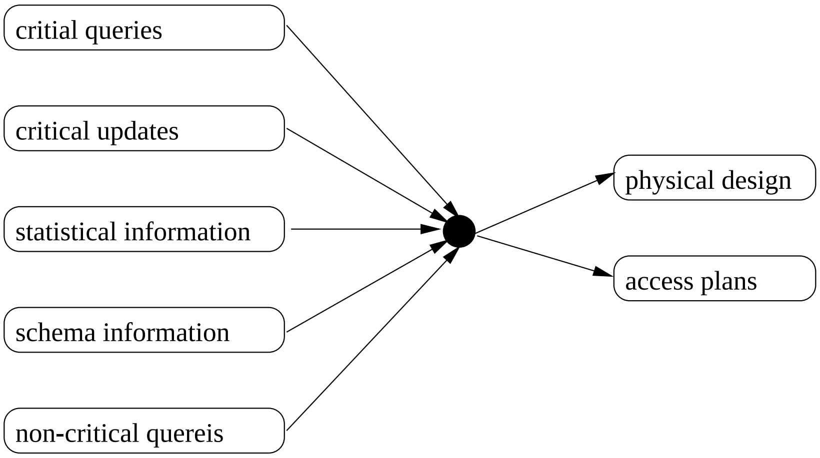

Figure 1.4: The APDECP problem I/O

main-memory or secondary storage. However, the control data of an ECP is usually stored in main-memory. This is why we explore algorithms for compacting the initially produced physical design, in the presence of integrity constraints, by introducing data structures with lots of pointers and access plans with lots of pointer chases. These algorithms allow us to reduce the need for storing the same data item multiple times and are better suited for accessing main-memory than secondary storage. As well, when the control data is stored in main-memory, cache sensitive data structures, such as the ones suggested in [27], and buffering mechanisms, such as the ones suggested in [41] are also worth exploring.

Note that the described problem differs from the traditional problem of automating physical database design. The later problem is usually described as selecting the best possible set of indices and/or materialized views for a given workload, where a workload is usually abstracted as a set of queries and updates together with their frequencies. In commercial systems, like IBM DB2 UDB [35] 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. For ECPs, however, QofS requirements can be specified. This is why the constraint changes to that of creating a physical design that allows for an efficient way of executing the defined critical operations. Since there is more emphasis on saving storage for ECPs, the optimization problem changes to that of minimizing the size of the physical design in a fixed physical design model.

1.2 Motivating Example

In this section we demonstrate different facets of the DECSUDT problem by presenting a solution to the problem for a small example workload. Suppose we are given the company schema shown in Figure 1.5. In the figure we have used ellipses around *base table* names and round rectangles around attribute types. Suppose as well that we are given a MV `REMP_STATS` (where R stands for rich) that stores the average age of the employees that make more than 100K in each department. This view can be expressed using the following underlying query.

materialized view `REMP_STATS` **:**
`select avr(`$e.age$`) as` avr_age, $e.dep$
`from EMP as` e
`where` $e.salary > 100\mathbf{K}$
`group by` $e.dep$

Suppose that we are also given the set of critical queries shown in Table 1.2 and the set of critical updates shown in Table 1.3. Query $Q1$ selects information about unusual employees (i.e. younger than 30 years of age, but making more than 100**K**) in departments with specific ranks and locations, where the result is ordered by the rank of the department. Query $Q2$ selects information about rich employees, where the result is ordered by the location and budget of the corresponding department. The three updates show an example of an insertion, deletion, and modification update.

The first major step in this example involves substituting each critical query with a number of perfect index joins, where indices are initially expressed using sSQL queries (see Table 4.3). This process can involve the introduction of MVs and derived attributes. In our example, we first rewrite `REMP_STATS` into the materialized view `REMP_FSTATS` (`F` stands for full) with the underlying query shown in Table 1.4. Note that this rewrite is along the lines of the algorithm for incrementally maintaining MVs with aggregates suggested in [26]. The purpose of the rewrite is to augment the attributes in the MV `REMP_STATS` to a set of attributes that can be easily incrementally maintained. In particular, an insertion of a tuple t to the table `EMP` can be propagated to `REMP_FSTATS` by increasing sum_age by $t.age$ and C by 1 in the tuple t' in `REMP_FSTATS` for which $t'.dep = t.dep$. This update can be performed

query $Q1$**:**
`select` $*$
`from RICH_EMP_STATS as` e_1, `DEP as` e_2, `COMP as` e_3
`where` $e_1.dep = e_2$`.ID and` $e_2.comp = e_3$`.ID and`
$e_3.loc = {:}P_1$ `and` $e_3.rank > {:}P_2$ `and` $e_1.avr\_age < 30$
`order by` $e_3.rank$ `asc`

query $Q2$**:**
`select` $*$
`from RICH_EMP_STATS as` e_1, `DEP as` e_2, `COMP as` c
`where` $e_1.dep = e_2$`.ID and` $e_2.comp = e_3$`.ID`
`order by` $e_3.loc$ `asc`, $e_3.budget$ `asc`

Table 1.2: Critical queries

insertion update: `insert into COMP` $(loc, budget, rank)$ `values` $\{{:}P_1, {:}P_2, {:}P_3\}$
deletion update: `delete from EMP as` e `where` $e.salary > {:}P_1$
modification update: `update COMP as` e `set` $e.budget = e.budget * 1.1$

Table 1.3: Critical updates

materialized view REMP_FSTATS: select sum($e.age$) as sum_age, count($*$) as C, $e.dep$ from EMP as e where $e.salary > 100\mathbf{K}$ group by $e.dep$
materialized view YREMP_FSTATS: select $*$ from REMP_FSTATS as e where $sum\_age/C < 30$
materialized view DEP_REMP_FSTATS: select $e_2.*$ from REMP_FSTATS as e_1, DEP as e_2 where $e_1.dep = e_2$.ID
materialized view DEP_YREMP_FSTATS: select $e_2.*$ from YREMP_FSTATS as e_1, DEP as e_2 where $e_1.dep = e_2$.ID
materialized view COMP_REMP_FSTATS: select $e_3.*$ from DEP_REMP_FSTATS as e_1, COMP as e_2 where $e_1.comp = e_2$.ID
materialized view COMP_YREMP_FSTATS: select $e_3.*$ from DEP_YREMP_FSTATS as e_1, COMP as e_2 where $e_1.comp = e_2$.ID

Table 1.4: Auxiliary Materialized Views

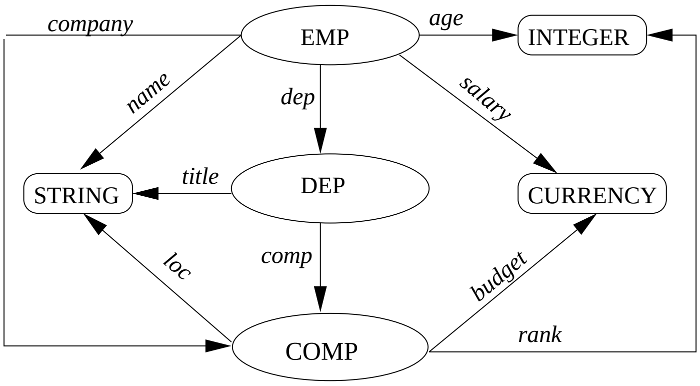

Figure 1.5: Example company schema

efficiently by using an index on the attribute *dep* of the MV `REMP_FSTATS`. Deletions and updates can be handled similarly, with the addition that tuples in `REMP_FSTATS` for which the value for the attribute C becomes 0 should be deleted from the MV. The value for the attribute avr_age is equal to sum_age/c and can be added as a derived attribute to the MV.

Note that the MV `REMP_FSTATS` will be used to answer the query $Q2$. The query $Q1$ only uses those tuples in `REMP_FSTATS` for which $avr\_age < 30$. We will create the MV `YREMP_FSTATS` to store those tuples. Formally, `YREMP_FSTATS` has the underlying query shown in Table 1.4. Note that the MVs `REMP_FSTATS` and `YREMP_FSTATS` enumerate the tuples in `REMP_STATS` that can contribute to the result of the queries $Q1$ and Q_2, respectively, for some values of the parameters of the queries. We also define the MVs `DEP_REMP_FSTATS` and `DEP_YREMP_FSTATS` as all the tuples in the table `DEP` that can contribute to $Q1$ and $Q2$, respectively, for some values of the parameters of the queries, where the underlying queries of the MVs are again shown in Table 1.4. Lastly, we define the MVs `COMP_REMP_FSTATS` and `COMP_YREMP_FSTATS` as all the tuples in the table `COMP` that can contribute to $Q1$ and $Q2$, respectively, for some values of the parameters of the queries. The underlying queries of the two views are again shown in Table 1.4.

Now, $Q1(:P_1,:P_2)$ can be answered using the following query plan.

for $t_2 \in$ `select *`
`from COMP_YREMP_FSTATS as` e

```
        where e.loc = :P1 and e.rank > :P2
        order by e.rank asc
    for t1 ∈ select *
          from DEP_YREMP_FSTATS as e
          where e.comp = :t2.ID
      for t0 ∈ select *
            from YREMP_FSTATS as e
            where e.dep = :t1.ID
          send join(t0, t1, t2);
```

Similarly, Q_2 can be answered using the following query plan.

```
  for t2 ∈ select *
        from COMP_REMP_FSTATS as e
        order by e.loc asc, e.budget asc
    for t1 ∈ select *
          from DEP_REMP_FSTATS as e
          where e.comp = :t2.ID
      for t0 ∈ select *
            from REMP_FSTATS as e
            where e.dep = :t1.ID
          send join(t0, t1, t2);
```

Note that `join` is an operation that joins three tuples from `EMP`, `DEP` and `COMP` according to the join condition in Q_1, which happens to be the same as the join condition in Q_2 and `send` is the primitive that returns the query result tuple by tuple. Note as well that we have used "`for` $t \in Q$" to denote in iterator in which the resulting tuple of Q in the defined order are bound with the tuple variable t one by one.

The outer most loop of each of the joins produces only tuples that contribute to the query result. Each such tuple then joins only with tuples in the next inner loop that contribute to the query result and so on. In other words, the query plan has no false drops. This means that the original critical query can be executed efficiently as long as the introduced simple queries can be executed efficiently. However, each of the later can be efficiently answered, for example, by using a single index scan on the corresponding MV.

query $Q3$**:**
`select *`
`from COMP_YREMP_FSTATS as` e
`where` $e.loc = :P_1$ `and` $e.rank > :P_2$
`order by` $e.rank$ `asc`

query $Q4$**:**
`select *`
`from COMP_REMP_FSTATS as` e
`order by` $e.loc$ `asc`, $e.budget$ `asc`

Table 1.5: Queries $Q3$ and $Q4$

The second important step of our solution is merging physical designs in order to save space. Consider queries $Q3$ and $Q4$ that are shown in Table 1.5 and that are produced as a result of the query break-up from the previous step. Tables 1.6 and 1.7 sketch a physical design that can efficiently answer them in the presence of the updates from Table 1.3. The index X_1 stores the distinct locations in `COMP_REMP_FSTATS` in ascending direction. Each node in the search tree of the index that stores a location from `COMP_YREMP_FSTATS` or that contains a child node that stores a location from `COMP_YREMP_FSTATS` is marked. In this way we have create an efficient way of traversing those nodes in the index that correspond to locations of departments in which the average age of the rich employees (i.e. those that make more than 100**K** is beyond 30. This allows us to efficiently answer range queries over the distinct values of the attribute *loc* in the MV `COMP_YREMP_FSTATS`. Note that we have used the fact that the tuples in `COMP_YREMP_FSTATS` are contained in `COMP_REMP_FSTATS`.

For each distinct location in `COMP_YREMP_FSTATS` (`COMP_REMP_FSTATS`), the index will store a pointer to an index on the distinct *rank* (*budget*) values for that location. Note that we have denoted these two sets of indices as the *parameterized indices* X_2 and X_3 in Figure 1.6. The value for the parameter P ranges over the referenced locations in X_1. One can think of a parameterized index as a set of indices, where there is an index for each distinct value of the parameters. The indices of type X_2 (X_3) will contain pointers to lists of records that contain all the elements with given *loc* and *rank* (*loc* and *budget*) values.

All the tuples from the table COMP also need to be stored in order to allow for updates on that table. We will store them in a linked list. The linked lists of type W_1 and W_2 will be encoded on top of this linked list by adding a *next* W_1 (W_2) pointer to each element of W_1 (W_2). Note as well that we do not need to use *previous* pointers because no deletions are defined on the table COMP and the update U_2 preserves any order of the records in the table COMP because the function $f(x) = x * 1.1$ is monotonic.

(*name*)	(*keys* (*k*))	(*values*)
$X_1^{\dagger}$	`select distinct` $e.loc$ `from COMP_REMP_FSTATS as` e `order by` $e.loc$ `asc`	$\&X_2(k)^{\ddagger}$ if $e \in$ COMP_YREMP_FSTATS $\&X_3(k)$ if $e \in$ COMP_REMP_FSTATS
$X_2(:P)$	`select distinct` $e.rank$ `from COMP_YREMP_FSTATS as` e `where` $e.loc = :P$ `order by` $e.rank$ `asc`	$\&W_1(:P, k)$
$X_3(:P)$	`select distinct` $e.budget$ `from COMP_REMP_FSTATS as` e `where` $e.loc = :P$ `order by` $e.budget$ `asc`	$\&W_2(:P, k)$

† The nodes in X_1 contain an extra marking bit

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

Table 1.6: Indices for efficiently answering the queries $Q3$ and $Q4$

1.3 Solution Overview

An overview of the our algorithm for solving the APDECP problem is shown in Figure 1.6. The labels of the arrows correspond to steps in the algorithm, except for the arrows labeled

(*name*)	(*elements*)
$W_1(:P_1,:P_2)$	`select *` `from COMP_YREMP_FSTATS as` e `where` $e.loc = :P_1$ and $e.rank = :P_2$
$W_2(:P_1,:P_2)$	`select *` `from COMP_REMP_FSTATS as` e `where` $e.loc = :P_1$ and $e.budget = :P_2$

Table 1.7: Record lists for efficiently answering the queries $Q3$ and $Q4$

"+", which denote the set union operation. All steps use the schema information and some of the steps modify it. However, for simplicity, this is not shown in the figure. As well, some steps produce access plan code, but again, in order to keep the figure simple, this data flow is not shown. Another event which is not shown in the figure is the construction of the non-critical part of the database (see Figure 2.1).

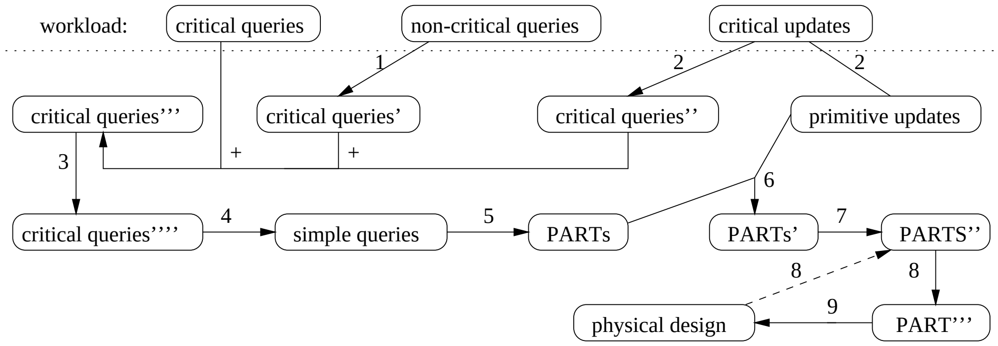

Figure 1.6: Algorithm Overview

Step 1 of the algorithm first breaks each non-critical query into a set of μSQL queries that access the critical part of the database and queries that access the non-critical part of the database. The first set is added to the input set of critical queries.

Step 2 of the algorithm transforms each critical update into an access plan that performs a primitive update for each tuple in the result of a critical query. The query is added to the set of critical queries and the updates are added as a parameterized update to the set of primitive updates.

Step 3 of the algorithm does a semantic query rewriting of the current set of critical queries. In other words, the defined integrity constraints are used to simplify the critical queries when possible.

Step 4 of the algorithm produces a query plan that breaks each critical query into a set of simple queries. This step may introduce MVs and derived attributes.

Step 5 of the algorithm converts each simple query into a *Parameterized Access Requirement Type* (PART). A PART represents a set of alternative physical designs. In particular, if a simple query is used to create a PART, then the PART represents exactly the set of physical designs that can be used to answer the query efficiently.

Step 6 of the algorithm modifies the PARTs created in the previous step by making them efficiently maintainable. In particular, efficient access plan for propagating every primitive update to each of the physical designs represented by the ePARTs is created. Note that a primitive update to a base table may need to be propagated to one or MVs. Section 3.1.2 described how this can be done for efficiently maintainable materialized views, which are the only views that we allow.

Step 7 of the algorithm is the PART merging step. Note that a PART represents one or more alternative physical designs, anyone of which can be selected to represent the PART in the final physical design. This step enumerates a set of possible solutions, where the size of each solution is found after performing Steps 8 and 9. The solution of the smallest size is then chosen.

Step 8 of the algorithm performs semantic PART simplification. For example, this step uses integrity constraints to transform PARTs into smaller, but equivalent versions.

The final step of the algorithm transforms each PART produced by the previous step of the algorithm into one of the smallest physical designs represented by the PART. It may be the case that there are several smallest physical designs represented by a PART, in which case this step will arbitrarily choose one of them.

1.4 Related Research

The idea of applying database technology in the development of ECPs is not new. Consider for example the eXtremeDB software ([9]) that is a main memory database that claims to produce QofS guarantees. It allows the user to specify the database at the physical level, i.e. what lists, indices, hash tables and so on are to be created. The search is done 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. TimesTen [31], MonetDB [24]). However, these systems do not provide QofS guarantees.

To summarize, the problem of automating physical design in the presence of QofS guarantees is relatively unexplored. One exception is the paper [37]. It explores the problem of creating indices that allow the efficient execution of a predefined workload of queries. However, the paper considers only simple partial-match queries in the absence of updates.

Note that our results in this thesis build on existing research in areas such as schema inference, semantic query rewriting, materialized view maintenance, object identifier theory, physical design structures, and so on. The corresponding references to these works are distributed throughout the thesis, i.e. whenever we present an idea based on existing research or include existing algorithms in our pseudo-code, we include a reference to the appropriate source.

1.5 Thesis Contributions

The main contribution of this thesis is the introduction of a system for automating physical design for embedded control programs. We believe that such a system can significantly reduce the development cost of an ECP. Our contributions are as follows.

1. We introduce novel endogenous and exogenous physical design structures that allow for a significant space economy. This is achieved by avoiding storing the same data multiple times in some cases. These data structures do not slow down the execution of the queries and, moreover, allow for updates to be propagated faster (if we avoid storing multiple copies of the same data item, then synchronized updates to this item will be faster).

- The major novelties of the endogenous physical design structure are twofold. First, it allows the index ordering to be in the shape of a tree. For example, a different ordering can be specified on records that have different values for certain attributes. Second, we allow storing additional marking bits in each node of the search tree of the index (see the motivating example in Section 1.2). Marking bits are advantageous because they allow us to answer in logarithmic time range queries over predefined collections of the indexed records.
- The major novelty of the exogenous physical design structure is that it can support several different orderings on the data inside the structure. In order for such a structure to be beneficial, it must be the case that those orderings have a common prefix.

2. We introduce the novel concept of a *Parameterized Access Requirement Type* (PART). This concept allows us to merge physical designs in a way that saves space. We present sound and complete algorithms for merging PARTs for the various endogenous and exogenous physical design structures.

3. A space of SQL queries that can be answered in logarithmic time using a physical design that can be updated in logarithmic time in the presence of every possible critical update is enumerated. We show that this set is the biggest possible in a conjunctive dialect of SQL if we allow at most $|def(Q)|_t$ regular indices to be used to answer a Q from the set in logarithmic time. We also introduce the novel concepts of *efficiently maintainable materialized views* and describe the biggest possible set of such views in a particular design model.

1.6 Thesis Outline

In Chapter 2 of the thesis we present a formal definition of the APDECP problem. In Chapters 3 of the thesis we show how a certain set of derived attributes and materialized views could be efficiently maintained. In Chapters 4 and 5 of the thesis we present out approach to solving the APDECP problem with the different physical design models. In Chapter 6 of the theses we present analytical results which justify the values of some of the APDECP problem parameters. In particular, we describe why the presented SQL subsets for defining the set of critical queries and the set of efficiently maintainable materialized views are in some reasonable way maximal. Chapter 7 of the work presents our experimental results. Chapter 8 summarizes the thesis and outlines areas for future research.

Chapter 2

Definitions

Recall that we formulated the APDECP problem as the problem of automatically generating pseudo-code for a DBMS with QofS guarantees that is used in the scenario shown in Figure 1.2. However, this problem is too general. This is why in this thesis we only solve part of the problem. In this chapter we formally define the problem that we actually solve.

The parameters of the APDECP problem include:

- the database schema expressiveness,
- the types of QofS restrictions,
- the allowed query language for each QofS restriction type,
- the allowed update language for each QofS restriction type,
- the physical design model,
- the cost metric for calculating the size of a physical design, and
- the type of available statistical information for the database.

The expressiveness of the database schema partly depends on the available tools for reasoning with integrity constraints. In Section 1.3 of this chapter we describe the set of integrity constraints, materialized views, and derived attributes we will be interested in.

We allow arbitrary QofS restrictions for the different queries and updates in the workload. However, in this thesis we only show how to build the physical design for those queries and updates on which a logarithmic QofS requirement is imposed. We also show how queries on which looser than logarithmic time-bound is specified could access the part of the database on which critical queries and updates are specified.

We restrict the type of queries and updates on which we allow the logarithmic time-bound to be specified. In particular, in Section 1.1 of this chapter we describe the allowed SQL query language, while in Section 1.2 we describe the allowed SQL update language.

We explore two physical design models: an endogenous one and an exogenous one. They are described in Sections 2.1, 2.2 of this chapter, respectively.

The cost metric that evaluates the quality of our solution will be some approximation of the size of the constructed physical designs. This approximation is based on the available statistical information, where we assume that we have enough information to approximate the size of any of any base table or a MV. Note however that some statistical information may not be very stable, i.e. relatively small change to the database can significantly affect it. However, we define and solve the APDECP problem as a non-incremental problem, i.e. the schema of the initially created physical design does not incrementally changed as the data in it changes. It is our belief that the presence of non-stable statistical information may require the physical design schema to be periodically redesigned. We leave an algorithm for doing so as an area for future research.

Note that the APDECP problem is NP-Hard regardless of the choice for the integrity constraint languages. The reason for this computational intractability is the complexity of the PART merging algorithm (see Step 7 of the algorithm shown in Figure 1.6). In this work we present an exact exponential algorithm and polynomial time approximation algorithm for solving the problem with extended indices and approximation algorithms for solving the problem with compact structures. In Chapter 7 of this thesis we present experimental results for the compact structure design model that shows its efficiency in the case of a TPC-C workload ([34]), i.e. an OLTP like workload with lots of updates and relatively simple queries. We believe such a workload is an appropriate test case because most embedded control applications are characterized by workloads with similar query/update ratio and complexity of critical queries.

Note that we require that the database is divided into a critical and non-critical part, where critical database operations can access only the critical part, non-critical updates can access only the non-critical part and non-critical queries can access the whole database (see Figure 2.1). Note that this model does not cover super-critical operations, i.e. operations for which faster than logarithmic time is desired. The reason is that such operations could not make use of the presented in this work algorithms for executing critical queries and updates because those algorithms will be too slow for them. On the other hand, the tool that automates the physical design for the non-critical database needs to be aware that the part of the queries that access the critical part of the database have logarithmic worst-case performance.

To understand the reasoning behind Figure 2.1, consider an update. If it updates part of the database that is accessed by critical queries, then it must be critical because it locks critical data. Similarly, if it updates part of the database that is referenced by critical updates, then it must be critical for the same reason.

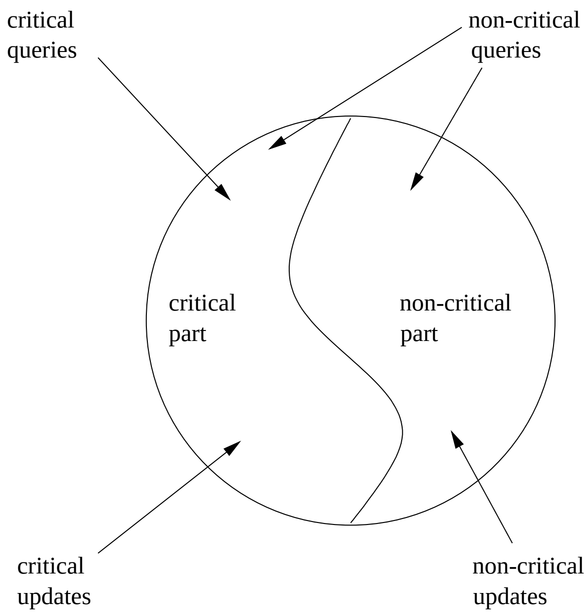

Figure 2.1: The critical and non-critical part of the control data

So far we have relied on the readers intuition when we used the terms efficient (i.e.

logarithmic) access plans for critical queries and updates. A precise definition of the two terms follows.

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

Definition 2 (efficient plan for an update) *Let U be a SQL update that updates the base table T (note that these are the only critical updates we allow). Let U_P be an access plan for U that consists of a query Q with a query plan Q_P that select the tuples to be updated. U_P also consists of a sequence of primitive updates, where each primitive updates corresponds to a resulting tuple in Q_P. Assume that the size to encode a value for each of the attributes of the database schema is a constant. The access plan U_P is efficient exactly when Q_P is an efficient plan for Q and each of the primitive updates can be performed in $O(log|T|)$ time.*

2.1 Query and Schema Languages

We use a modification of SQL to describe the database operations. A query in our modified SQL can reference the *base tables* and MVs in a database schema. Note that in this work we do not allow arbitrary MVs. In particular, we only consider a small subset of the possible MVs (see Section 1.3 of this chapter), where each MV in the subset is efficiently maintainable. Throughout this thesis we use the unqualified term table to refer to both base tables and MVs. We will use T to denote a base table, V to denote a MV, R to denote a table, and Σ to denote a database schema.

We will use the capital letters A, B, C, D, S, and M to refer to table attributes. Exactly one of the attributes of each table is the system attribute `ID`. We require that each tuple of each base table is uniquely identified by its `ID` value, i.e. the `ID` attribute is a global tuple identifier in the space of base tables.

Note that the `ID` attribute is similar to the global object identifier of the ODMG model (see [4]). The differences are that: (1) the `ID` attribute of a tuple 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, i.e. two distinct tuples that belong to base tables can not have the same `ID` value because they are never stored at the same location. However, as we will see later in this section, a tuple of a MV can share the value for the `ID` attribute with a tuple of a base table.

In order to be able to map the `ID` of a tuple to its physical address, we create an `ID` *lookup structure*. This structure consists of three look-up tables: `local look-up`, `global look-up`, and `physical design look-up`. The local look-up table has three attributes: `ID`, `physical design`, and `address`, where the `ID` and `physical design` attributes form the table's primary key. When we try to find the address of a tuple, we first check this table and see if there is an entry for the corresponding tuple and physical design. The global look-up table has two attributes: `ID` and `address`, where the `ID` attribute is the table's primary key. It stores information about the addresses of tuples that are shared between several data structures. The stored addresses are global, i.e. they are not an offset in a particular physical design. The physical design look-up table has the attributes `ID` and `address`, where the `ID` attribute is the table's primary key. It contains information about the different physical designs and their location (note that we associate an `ID` value not only with tuples, but also with physical designs). The described tables need to be updated every time a record or a physical design is inserted, deleted, or its physical address is modified. Each of the described tables can be implemented using a hash functions. The `ID` value of a tuple or a physical design cannot be modified during the tuple's or physical design's life. Once all tuples that are stored at a particular physical address are deleted, then the particular `ID` value is released for reuse. Similarly, once a physical design is deleted, its `ID` value is released for reuse.

The non-`ID` attributes of a table are either *standard* and are of one of the predefined types (e.g. integer, string) or are *reference* and contain the `ID` of a tuple in a base table. We require that all reference attributes are not `NULL` and point to an existing tuple, i.e. we impose a foreign key constraint for reference attributes.

Definition 3 (attributes of a table) *We will use* $\texttt{Attr}(R)$*,* $\texttt{Attr}^{\texttt{r}}(R)$*, and* $\texttt{Attr}^{\texttt{s}}(R)$ *to denote, respectively, all the attributes of the table* R*, the reference attributes of the table* R*, and the standard attributes of the table* R*. We will also use* $\texttt{Attr}(Q, R)$*,* $\texttt{Attr}^{\texttt{r}}(Q, R)$*, and* $\texttt{Attr}^{\texttt{s}}(Q, R)$ *to denote, respectively, the set of attributes in* $\texttt{Attr}(R)$*,* $\texttt{Attr}^{\texttt{r}}(R)$*, and* $\texttt{Attr}^{\texttt{s}}(R)$ *that are referenced in the query* Q*.*

Since the `ID` attributes of tables play an important role in reasoning about table containment, we present the following definitions. $R_1 \subseteq_{\texttt{ID}} R_2$ is true when the `ID` values in R_1 are a subset of the `ID` values in R_2, i.e. $\pi_{\texttt{ID}} R_1 \subseteq \pi_{\texttt{ID}} R_2$. Similarly, we define $R_1 \equiv_{\texttt{ID}} R_2$ to be true iff $\pi_{\texttt{ID}} R_1 \equiv \pi_{\texttt{ID}} R_2$.

Our modified SQL differs from the traditional SQL in the creation of an `ID` attribute for query results that are stored in MVs. In order to describe how the value for the `ID` attribute is calculated, we present the following intermediate definitions.

Definition 4 (join condition) *The predicate formula* θ *is a join condition for the set of tables* $\{R_i\}_{i=1}^{r}$ *iff the formula* θ *is a conjunction of atomic predicates of the form* $R_l.A_k = R_d.\texttt{ID}$*, where* $l, d \in [1, r]$*,* $l \neq d$*, and* A_k *is a reference attribute that has the property that* $\pi_{A_k} R_l \subseteq \pi_{\texttt{ID}} R_d$*.*

Definition 5 (join graph) *The graph* G *is the join graph for the set of tables* $\{R_i\}_{i=1}^{r}$ *relative to a join condition* θ *iff there is a one to one correspondence between the nodes in* G *and the tables* $\{R_i\}_{i=1}^{r}$ *and between the edges in* G *and the atomic conditions in* θ*, where there is a directed edge from* R_l *to* R_d *labeled* A_k *in* G *iff* θ *contains the atomic predicate* $R_l.A_k = R_d.\texttt{ID}$*.*

Definition 6 (join tree) *A join tree is a join graph that is a tree, where the edges are directed from a parent node to a child node.*

Note that in this work all trees that we consider are rooted.

Definition 7 (navigating a directed graph) *Let* G *be a directed graph. We introduce the following definitions.*

1. $\texttt{parent}^G(n) = \{n' : \textit{there is a directed edge from } n' \textit{ to } n\}$

2. $\texttt{ancestor}^G(n) = \{n' : \textit{there is a directed path from } n' \textit{ to } n\}$

3. $\texttt{child}^G(n) = \{n' : \textit{there is a directed edge from } n \textit{ to } n'\}$

4. $\texttt{descendent}^G(n) = \{n' : \textit{there is a directed path from } n \textit{ to } n'\}$

Suppose that we are given a MV V that stores the result of the query Q that references only base tables. If Q has the following form:

select $A_1, \ldots, A_a$
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 ... and $\gamma_r(e_r)$,

where θ is a join tree with root R_1 and $\{\gamma_i\}_{i=1}^r$ are predicates, then V will contain a subset of the tuples in R_1 joined with the other tables in the query and therefore it can inherit the values of the `ID` attribute of R_1. In all other cases, new unique values are automatically generated for the `ID` attribute of V.

2.1.1 The Different Query Types

Figure 1.3 shows four query types, where sSQL, μSQL, and eSQL are query types that have a syntactic definition, while the cSQL query set has a semantic definition.

A sSQL query is of one of the three query types shown in Table 4.3. Note that throughout this work *dir* stands for direction and is equal to either `asc` or `desc`. As well, we will use $dir^{\texttt{TRUE}}$ to denote *dir* and $dir^{\texttt{FALSE}}$ to denote the inverse of *dir*, i.e. $dir^{\texttt{FALSE}} = \texttt{desc}$ iff $dir = \texttt{asc}$ and $dir^{\texttt{FALSE}} = \texttt{asc}$ iff $dir = \texttt{desc}$. Note as well that through out this work we use $[\cdot]$ to denote an optional component. A sSQL query has the property that it can be efficiently answered using a single index. Note that queries of type 3 can be answered in constant time using the look-up structure.

Before defining the set of μSQL queries, we present the following intermediate definitions.

Definition 8 (valid join graph) *A join graph G is valid iff the graph is connected and the corresponding undirected graph is acyclic. Unless otherwise specified, throughout the*

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

$\{|A_i|\}_{i=1}^{a}$ are distinct attributes of R, $\{|D_i|\}_{i=1}^{d}$ are distinct attributes of R, $\{A_i\}_{i=l+1}^{a}$ are non-reference attributes, and $\{A_i\}_{i=1}^{l}$ are non-`ID` attributes.

Table 2.1: The three sSQL query types

paper we will use $\theta(R_1, \ldots, R_r)$, or just θ when the referenced tables are implied, to denote a valid join graph and the corresponding join condition.

Definition 9 (admissible predicate system) *A predicate system $\mathcal{S}$ is denoted by the quintuple $\langle \Delta, \mathcal{V}, \mathcal{B}, \mathcal{O}, \mathcal{C} \rangle$, where Δ is a domain of elements, $\mathcal{V}$ is a set of variables that range over subsets of Δ, $\mathcal{B}$ is a set of binary Boolean operators, $\mathcal{O}$ is a set of binary operators defined over subsets of Δ, and $\mathcal{C}$ is a set of comparison operators defined over subsets of Δ. The syntax of each predicate γ in $\mathcal{S}$ can be described using the following grammar.*

$$\gamma ::= \mathcal{F} \mid (\mathcal{F}\ \mathbf{op}\ \mathcal{F}) \ // \textit{ where } \mathbf{op} \in \mathcal{B}$$
$$\mathcal{F} ::= (\mathcal{E}\ \mathbf{comp}\ \mathcal{E}) \ // \textit{ where } \mathbf{comp} \in \mathcal{C}$$
$$\mathcal{E} ::= (\mathcal{A}\ \mathbf{op}\ \mathcal{A}) \mid (\mathcal{E}\ \mathbf{op}\ \mathcal{A}) \mid (\mathcal{A}\ \mathbf{op}\ \mathcal{E}) \ // \textit{ where } \mathbf{op} \in \mathcal{O},\ \mathcal{A} \in \mathcal{V} \cup \Delta$$

If $\mathcal{V}$ consists of k variables we will write $\gamma(c_1, \ldots, c_k)$ to denote the result of substituting *the constants $\{c_i\}_{i=1}^k$ from Δ for those variables. Such an expression is evaluated to either true of false. We will call the predicate system $\mathcal{S}$ admissible iff the following holds.*

- *$\mathcal{S}$ is closed under conjunction, i.e. $\gamma_1 \in \mathcal{S}$ and $\gamma_2 \in \mathcal{S}$ implies $\gamma_1 \wedge \gamma_2 \in \mathcal{S}$.*
- *The following problems could be solved in polynomial time:*
  1. *The problem of whether a predicate in $\mathcal{S}$ is satisfiable, and*
  2. *The predicate subsumption problem (i.e. the problem of whether for $\gamma_1, \gamma_2 \in \mathcal{S}$, every binding of the variables of $\mathcal{S}$ that makes γ_1 true also makes γ_2 true).*
- *It can be decided in $O(|\gamma| \cdot \sum_{i=1}^{k} |c_i|)$ time whether a predicate $\gamma \in \mathcal{S}$ holds for the bindings $\{v_i = c_i\}_{i=1}^k$, where $\{v_i\}_{i=1}^k$ are the variables in γ and $\{c_i\}_{i=1}^k \subseteq \Delta$.*

Unless we specify otherwise, we will use γ to denote a predicate over an admissible system with variables consisting of the attributes of a table and $\gamma(R)$ to denote the result of applying the predicated γ to the table R. As well, we will use the term efficient predicate *to refer to a predicate over an admissible predicate system.*

For example, the results published in [11] and [28] can be used to infer that $\langle \texttt{INTEGER} \cup \texttt{CURRENCY}, \{age, salary\}, \{\wedge, \vee\}, \{+, -, *\}, \{>, <, =\}\rangle$ is an admissible predicate system, where *age* is defined over the domain `INTEGER`, *salary* is defined over the domain `CURRENCY`, and the operations $\{+, -, *\}$ are defined appropriately. Let γ be the efficient predicate $((age > 30) \wedge (salary > 20K))$ over this system. Then the expression $\gamma(\texttt{EMP})$ denotes the set of employees that receive *salary* greater than 20K and are older than 30 years of age.

Definition 10 (inverse tree in a valid join graph) *The tree G^{-t} is an inverse tree for the valid join graph G iff G^{-t} is a subgraph of G and G^{-t} is a tree in which the edges are directed from a child node to a parent node.*

Definition 11 (tuple ordering) *For a table R, a tuple ordering is defined using the syntax $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle\rangle$, where $A_1, \ldots, A_a$ are distinct attributes of the table R. It denotes an ordering of the tuples in the table R, where the tuples are first ordered relative to the value of A_1 in ascending order if $dir_1 =$ `asc` and in descending order otherwise, next* *relative to the value of the attribute A_2 in direction dir_2 and s.o. Throughout the paper we will use O to denote a tuple ordering. When the table over which the tuple ordering is defined is implied, we will use the syntax $\langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle$.*

Definition 12 (valid tuple ordering in an inverse tree) *Let G^{-t} be an inverse tree with nodes $\{R_i\}_{i=1}^k$ for the valid join graph G with nodes $\{R_i\}_{i=1}^r$ and let θ be the join condition corresponding to G. We will say that 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 $R \equiv \theta(R_1, \ldots, R_k)$, iff:*

1. $\{A_i\}_{i=1}^a \subseteq \bigcup_{i=1}^{k} Attr(R_i)$.
2. *All the attributes that belong to the same table R_i are consecutive in O for every $i \in [1..k]$. We will refer to the tuple ordering defined by those attributes as O_i.*
3. *If $R_i \in \mathtt{ancestor}^\theta(R_j)$ and O_i and O_j are both not-empty, then O_j comes before O_i in O $(i, j \in [1..k])$.*
4. *For every $i \in [1..k]$, either O_i contains the system attribute* `ID` *or for every $j \in [1..k]$ s.t. $R_j \in \mathtt{ancestor}^\theta(R_i)$, O_j is the empty tuple ordering.*

A μSQL query has one of the forms shown in Table 2.2. Figure 2.2 depicts the general shape of the join condition.

An *examined critical* SQL (eSQL) query has the general syntax shown in Table 2.3

2.1.2 Critical Updates Language

An admissible critical update either adds a tuple to a base table T of the input schema Σ, or delete or modify the tuples in a specified base table T that pass the condition of a predicate γ over an admissible predicate system. If there is a modification, then the modification function should be able to compute its result in $O(|input|)$ time. Such an update can be translated into a sequence of *primitive updates* that are of one of the three types shown in Table 2.4. Note that an insertion is already a primitive update, while a deletion or a modification can be translated into a sequence of primitive updates by calculating the result of the following query.

(*type*)	(*query*)
(1)	select $A_1, \ldots, A_a$ 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 $B_1 = :P_1$ and $\ldots$ and $B_b = :P_b$] [order by B_{b+1} $dir_1, \ldots, B_{b+d}$ dir_d]
(2)	select $A_1, \ldots, A_a$ 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 $B_1 = :P_1$ and $\ldots$ and $B_b = :P_b$ and B_{b+d+1} between $:P_{b+1}$ and $:P_{b+2}$ [order by B_{b+d+1} dir_{d+1}, B_{b+1} $dir_1, \ldots, B_{b+d}$ dir_d]
(3)	select $e.A_1, \ldots, e.A_a$ from R_1 as e where $e.\mathtt{ID} = :P_1$

$a \geq 1$, $r \geq 1$, $0 \leq k \leq r$, $b \geq 0$, $d \geq 0$, $\{|A_i|\}_{i=1}^{a}$ are distinct attributes of the tables $\{R_i\}_{i=1}^{r}$, $\{|B_i|\}_{i=1}^{b+d+1}$ are distinct attributes of the tables $\{R_i\}_{i=1}^{k}$, the tables $\{R_i\}_{i=1}^{k}$ form an inverse tree in θ, B_{b+d+1} is a standard attribute that belongs to the root table of the inverse tree, $\{B_i\}_{i=1}^{b} \cup \{\}$ are non-`ID` attributes that also belong to the root table of the inverse tree, the tuple ordering in the `order by` condition, when present, is a valid tuple ordering for the inverse tree, and for every $i \in [k+1..r]$ there exists $j \in [1..k]$ s.t. there is a directed path in θ from R_j to R_i.

Table 2.2: The possible μSQL query types

(*type*)	(*query*)
(1)[†]	`select` $A_1, \ldots, A_a$ `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` $\ldots$ `and` $\gamma_k(e_k)$ `and` $B_1 = {:}P_1$ `and` $\ldots$ `and` $B_b = {:}P_b$ `and` B_{b+1} `between` ${:}P'_1$ `and` ${:}P''_1$ `and` $\ldots$ `and` B_{b+c} `between` P'_c `and` P''_c `order by` D_1 $dir_1, \ldots, D_d$ dir_d,
(2)	`select` $e.A_1, \ldots, e.A_a$ `from` R_1 `as` e `where` $e.$`ID` $= {:}P_1$

[†] θ is a valid join graph for the tables $\{R_i\}_{i=1}^{r}$, the attributes denoted with the same capital letter but different subscript are distinct, $\{B_i\}_{i=1}^{b}$ are non-`ID` attributes, $\{B_i\}_{i=b+1}^{b+c}$ are standard attributes, and $\{D_i\}_{i=1}^{d} \cap \{B_i\}_{i=1}^{b} = \emptyset$. Note as well that part of the `where` statement, the whole `where` statement, and/or the whole `order by` statement could be missing.

Table 2.3: The possible eSQL query types

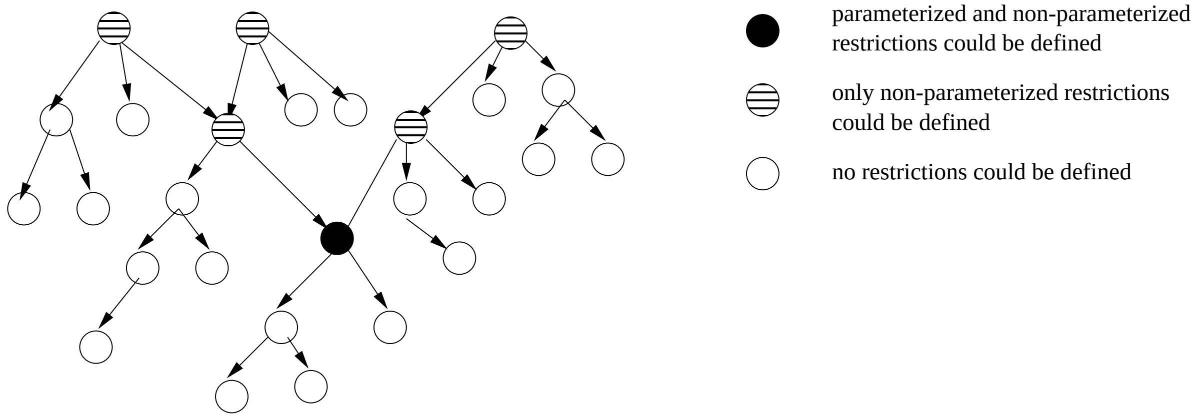

Figure 2.2: Depicts the shape of the join tree of a μSQL query

select ID
from T as e
where $\gamma(e)$

We will require that the specified critical updates are consistent with the defined integrity constraints. We will also require that the constructed low-level updates can be ordered to be also consistent. In Section 4.2.2 we examine sufficient conditions for the later to hold.

2.1.3 The Schema Model

Materialized Views

As described in the beginning of the chapter, base tables and MVs are an integral part of our schema model. One significant difference between base tables and MVs is that the content of a MV needs to be synchronized with the content of its underlying tables in order to avoid staleness. We will refer to the one time synchronization of a MV as a *refresh* and to the process of continuously synchronizing the content of a MV as *maintenance*. In this work we only consider *immediate* and *incremental* MV maintenance algorithms (see [14]). Immediate MV maintenance algorithms refresh the content of a MV before a transaction that updates one or more of the MV's underlying tables can be committed. Incremental MV maintenance algorithms refresh the content of a MV by first calculating the change to

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

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

Table 2.4: The three primitive update types

the MV and then applying this change to the view.

We next introduce several intermediate definitions that will be used in defining the concept of an *efficiently maintainable MV*.

Definition 13 (self-maintainable physical design) [1] *A physical design $\mathcal{D}$ that is part of a database $\mathcal{I}$ is self-maintainable iff the new value of $\mathcal{D}$, i.e. the value of $\mathcal{D}$ after a primitive update δ has been applied to $\mathcal{I}$, can be calculated as a function of δ and the old value of $\mathcal{D}$.*

For example, consider a linked list that stores the natural join of the tables R_1 and R_2. In the absence of schema information this physical design is not self-maintainable if all possible primitive updates are defined. For example, consider an insertion of the tuple t_1 into the table R_1 and suppose that t_1 joins with a single tuple t_2 in R_2 and before the insertion t_2 did not join with any tuples in R_1. In order to compute the new value for the linked list, we will need to know the value of the tuple t_2. However t_2 is stored neither in the old value of the linked list, nor in the primitive update and therefore the described physical design is not self-maintainable.

Definition 14 (efficiently maintainable MVs) *Let V be a MV, Q be an underlying query of V that references only base tables, and $\bar{\mathcal{D}}$ be a space of physical designs. The MV*

[1] This definition is inspired by the definition of a self-maintainable set of materialized views presented in [14].

V is efficiently maintainable relative to $\bar{\mathcal{D}}$ iff there exists a physical design $\mathcal{D}$ in $\bar{\mathcal{D}}$ that has the following properties.

- *The physical design consisting of $\mathcal{D}$ and a bag of the records corresponding to the tuples in V is self-maintainable.*
- *Any primitive update can by synchronized with the physical design consisting of $\mathcal{D}$ and a bag of the records corresponding to the tuples in V in $O(def(Q) \cdot (\sum_{i=1}^{k} log(|T_i|) + log(|V|)))$ time, where $\{T_i\}_{i=1}^{k}$ are the underlying tables of V.*

Note that we assume that a bag or records has the capabilities of inserting a record, deleting a record with given `ID`, returning the record with a given `ID`, and enumerating all the records in the bag.

In this work the space $\bar{\mathcal{D}}$ is defined as the space of possible regular indices in Σ. We assume that the initial database schema can contain MVs and that MVs can be introduced during our algorithms. However, we restrict the syntax of the underlying query of a MV to be of either a query of type 1 or a query from type 2, where the two query types are shown in Table 2.5.

In Chapter 3 we show the restriction on MVs guarantees the efficiently maintainable property of MVs with the presented syntax. In Chapter 6 we show why this restriction can not be relaxed further without significantly changing the MV definition language or relaxing certain assumptions.

Derived Attributes

Derived attributes are another element of our schema model. The value for such an attribute can be computed as a function of the values of the other attributes of the table over which the attribute is defined and, potentially, data from other tables. Similarly to MVs, derived attributes also need to be *maintained* in order to avoid staleness.

Definition 15 (efficiently maintainable derived attribute) *We will call a derived attribute A efficiently maintainable iff $O(\Sigma)$ derived attributes $\bar{A}$ can be introduced to the database schema in such a way so that the value of the derived attributes $\bar{A} \cup \{A\}$ can be refreshed in $O(|\Sigma|)$ time.*

(*type*)	(*underlying query*)
(1)	`select` $A_{l+1}, \ldots, A_a$ `from` T `as` e `where` $\gamma(e)$
(2)	`select` $[count(*)$ `as` $C,]$ $agg_{c_1}(A_1)$ `as` $C_1, \ldots, agg_{c_l}(A_l)$ `as` C_l, $A_{l+1}, \ldots, A_a$ `from` T_1 `as` $e_1, \ldots, T_r$ `as` e_r `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` $\ldots$ `and` $\gamma_r(e_r)$ `group by` $A_{l+1}, \ldots, A_a$

† $1 \le l+1 \le a$, θ in the shape of an inverse tree with root T_r, $\{|A_i|\}_{i=l+1}^{a}$ are distinct non-`ID` attributes of T_r, $\{A_i\}_{i=1}^{l} \cap \{A_i\}_{i=l+1}^{a} = \emptyset$, $\{|A_i|\}_{i=1}^{l}$ are not necessarily distinct standard attributes of the tables $\{T_i\}_{i=1}^{r}$, $agg_{c_i} \in \{\texttt{sum}, \texttt{avr}, \texttt{max}, \texttt{min}\}$ if $A_i \in \texttt{Attr}(T_r)$, and $agg_{c_i} \in \{\texttt{sum}, \texttt{avr}\}$ otherwise.

Table 2.5: Syntax of Efficiently Maintainable MVs

Our presumption is that the original database schema contains no derived attributes, but we are allowed to modify this schema and introduce efficiently maintainable derived attributes. When doing so, we will describe how the introduced derived attributes can be efficiently maintained by presenting *refresh code*.

Suppose t is a tuple of the base table R_r and A is a derived attribute of t. The value for $t.A$ is either a function of the non-derived attributes of t or a function of the whole database. In the first case, $t.A = f(t.A_1, \ldots, t.A_a)$, where $A_1, \ldots, A_a$ are standard non-derived attributes of R_r and f is a function that computes its result in $O(|input|)$ time.

In the second case, $t.A$ is defined as the result of a query, where the query can be of one of the two types shown in Table 2.6.

In Chapter 3 we show that this restriction guarantees the efficiently maintainable property of derived attributes with the presented syntax.

attribute	*value*
$t.C_r^{\dagger}$	`select count`$(*)$ `from` R_1 `as` e_1, ..., R_r `as` e_r `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` ... `and` $\gamma_{r-1}(e_{r-1})$ `and` e_r.ID= :t.ID
$t.S_{(A,r)}^{\dagger}$	`select sum`(A) `from` R_1 `as` e_1, ..., R_r `as` e_r `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` ... `and` $\gamma_{r-1}(e_{r-1})$ `and` e_r.ID= :t.ID

† θ is a valid join graph in the shape of an inverse tree with root R_r

Table 2.6: Possible values for the derived attribute of a tuple $t \in R_r$

Integrity Constraints

The integrity constraints that we will consider in this work are functional dependencies, foreign key dependencies, equality constraints, disjointness conditions, containment conditions, and covering conditions, where we are interested in integrity constraints specified both over base tables and efficiently maintainable MVs.

The syntax of a functional dependency is $R(A_1, \ldots, A_a \rightarrow B_1, \ldots, B_b)$ and it denotes the fact that if two tuples in R agree on the values for the attributes $A_1, \ldots, A_a$, then they must also agree on the values for the attributes $B_1, \ldots, B_b$.

Definition 16 (acyclic functional dependencies) *The functional dependencies in the database schema Σ are acyclic iff:*

1. $\bar{A} \rightarrow \bar{B} \rightarrow C$,
2. *there does not exist a strict subset $\bar{A}'$ of the attributes in $\bar{A}$ such that $\bar{A}' \rightarrow \bar{B}$, and*
3. *$\bar{A}$ and $\bar{B}$ do not share attributes in common*

imply that $C \notin \bar{A}$.

We require that functional dependencies are acyclic. The reason for the imposed restriction is to make the semantic PART simplification in Step 8 of our algorithm (see Section

4.2.8) deterministic. Note that the functional dependencies in most database schema that represent real world requirements are indeed acyclic.

Foreign keys are another type of integrity constraint that we will consider. In this work we consider only foreign key constraints that are expressed using derived attributes. In particular, we will write $R_1[A] \subseteq R_2[\mathtt{ID}]$ to denote the foreign key constraint $\pi_A R_1 \subseteq \pi_{\mathtt{ID}} R_2$. We will only allow such constraints when A is a reference attribute (recall that we require that reference attributes are non-`NULL`).

We will require that the database schema is also acyclic with respect to foreign key constraints. Formally, this implies that there can not exist tables $R_1, \ldots, R_r$, $r \geq 2$, such that $R_1[A_1] \subseteq R_2[\mathtt{ID}], \ldots, R_{r-1}[A_{r-1}] \subseteq R_r[\mathtt{ID}]$, and $R_r[A_r] \subseteq R_1[\mathtt{ID}]$. This constraint is necessary to guarantees that a consistent critical update that updates more than one tables can be translated into a sequence of allowed consistent critical updates. For example, since we only allow critical updates that modify only a single table, there would be no way to insert a tuple in either R_1 not R_2 if they are initially empty and there is a foreign key constraint between R_1 and R_2 and between R_2 and R_1.

A more rarely used type of an integrity constraint is the equality constraint. The syntax of an equality constraint is $\mathtt{eq}(R : \mathit{Pf} = B)$, where Pf is a *path function* and B is an attribute of the table R. The syntax of a path function is a sequence of attributes separated by dots, e.g: $A_1.A_2.A_3$. In order to describe the semantics of a path function, consider a table R and a reference attribute A. While $t.A$ denotes the value of the reference attribute, we define $t \Rightarrow A$ to denote the base table tuple with `ID` $t.A$. Then, if $\mathit{Pf} = A_1. \ldots .A_a$, we will use $t.\mathit{Pf}$ as a shorthand for $((\ldots(t \Rightarrow A_1) \Rightarrow \ldots) \Rightarrow A_{a-1}).A_a$. The stated equality constraint holds when for every tuple $t \in R$, $t.\mathit{Pf}$ and $t.B$ are both defined and have the same value.

Given two tables R_1 and R_2, we will say that the tables are disjoint iff $\pi_{\mathtt{ID}} R_1 \cap \pi_{\mathtt{ID}} R_2 = \emptyset$. The containment conditions for the tables R_1 and R_2 is $\pi_{\mathtt{ID}} R_1 \subseteq \pi_{\mathtt{ID}} R_2$. The covering condition for a table R and a set of tables $\bar{R}$ is $\pi_{\mathtt{ID}} R \subseteq \bigcup_{R' \in \bar{R}} \pi_{\mathtt{ID}} R'$.

In our algorithms each of the presented integrity constraints can be used in semantic rewriting which can potentially decrease the size of the produced physical design. Unfortunately, the decision problem of whether an integrity constraint is a consequence of a set of integrity constraints is vastly undecidable for the presented set of integrity constraints (for proof see [33] or [23]). We therefore relay on black box reasoner with the above integrity constraints which is not complete. In other words, when asked whether an integrity constraints is a consequence of a set of integrity constraints, the reasoner may answer "YES", "NO", or "I DON'T KNOW". Such a reasoner can be implemented, for example, using a description logics reasoner that incorporates concrete domains ([3]).

2.2 The Different Physical Design Models

One of the parameters of the APDECP problem is the physical design model. In this thesis we consider two different physical design models. Before describing them in details, we present the concepts of an *endogenous* and *exogenous* search structure.

Definition 17 (search attributes and search values) *Let X be an index that indexes the tuples in the table R. We will use the term search attributes to refer to those attributes of R on which range and/or partial-match queries can be efficiently answered using X. As well, we will use the term search values to denote both the duplicate preserving and the regular projection of these attributes on R. We will use the term distinct search values to denote only the duplicate preserving projection of the search attributes of R on R and the term non-distinct search values to denote the duplicate eliminating projection of the search attributes of R on R.*

Suppose that we are given a set of possibly ordered records and we need the capability of efficiently finding a record given one or more of its attributes. Such a capability can be implemented, for example, using an index in the shape of a search tree. We will call such an index endogenous if the records are stored inside the search tree (see the upper part of Figure 2.3) and exogenous if the search tree contains only the search values and the records are stored separately (see the lower part of Figure 2.3).

Another orthogonal classification of indices is *leaf indices* versus *non-leaf indices*. In a leaf index all the search values for an exogenous index and all the records for an endogenous index are stored in the leaf nodes. (A B^+ tree is an example of a leaf index.) In a non-leaf index the search values for an exogenous index and the records for an endogenous index are stored in all the nodes of the tree. (A B tree is an example of a non-leaf index.)

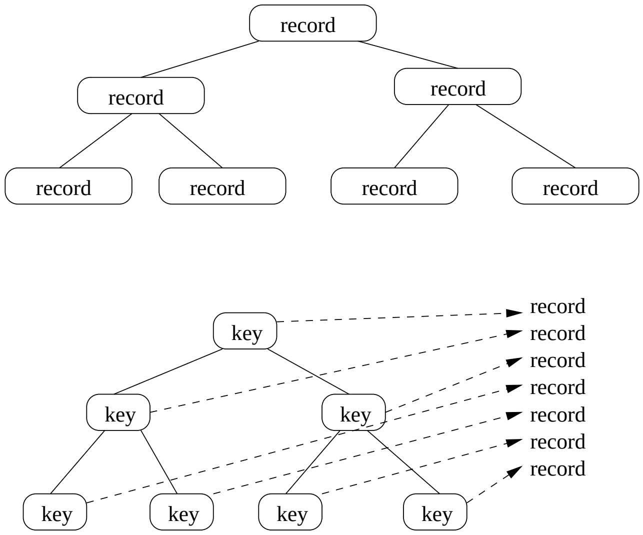

Figure 2.3: Sketch of an endogenous and exogenous index

A third classification of indices is *key* versus *total* indices. In an exogenous key index only the distinct search values are stored in the index, while in an exogenous total index the non-distinct search values are stored in the index. Note that we consider all endogenous indices to be total because they store the non-distinct search values inside the index.

We will also associate a *node capacity* with each index. This corresponds to the maximum number of records that can be stored in each node of the search tree of the index.

The *extended index* physical design model consists exclusively of endogenous indices. The *compact structure* physical design model consists of exogenous indices, hash structures, and linked lists. Note that the traditional index and linked list structures ([17]) are modified in a way that allows for more compact data representation. The actual choice of physical design depends on the ability of the query engine to utilize it, where the compact structure physical design is more complex, but also more compact than the extended index physical design. A detailed description of the physical design models follows.

2.2.1 Extended Index Design Model

This design model consists exclusively of extended indices. However, before describing the concept of an extended index we first describe the concept of a *regular index*. Note that both index types are endogenous.

Definition 18 (regular index: syntax and semantics) *The syntax of a regular index is $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle\rangle$. We will sometimes refer to this string as the description string of the index. In order for a regular index to be valid, we require that the tuple ordering $\langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle$ is a total order for the elements of R, i.e. the attributes $\{A_i\}_{i=1}^{a}$ form a key for the table R. The physical design for such an index consists of a search tree. The records in the search tree contain the tuples of R in the order $\langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle$.*

A regular index X can be implemented on top of a search tree containing the described tuples and supporting the described order. Such a search tree can be, for example, an AVL tree [1], an endogenous B trees [39, 29, 30], a CSB^+ tree [27], or a *T tree* [21]. Note that we avoid pinpointing an actual tree search structure because different circumstances call for different search structures. For example, according to Lehman and Carey (see [22]), B trees are better suited for secondary storage and T trees are better suited for main-memory applications. On the other hand, CSB^+ trees ([27]) are better suited for a cache conscious scenario. Regardless of which implementation for the search tree is chosen, a regular index supports the following operations.

Definition 19 (interface of a regular index) *Let $|X|$ be used to denote the number of tuples in the regular index X with description string $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle\rangle$. Assume that the length of the records in X and the node capacity of the search tree of X are constant. Then X has the following capabilities, where the* **retrieve** *method has constant worst-case running time and the other methods have $O(log(|X|) \cdot a)$ worst case running time.*

1. `reference insert(tuple` t`):`

   - **pre-conditions:** *t is compatible with R[2] and t is not already inserted in X.*

[2]We will say that a tuple t is compatible with a table R when the attributes of t match those of R

- **action:** *The tuple t is inserted in the index X.*
- **return value:** *This method returns the* `ID` *of the inserted tuple.*

2. `delete(reference` p`):`

- **pre-conditions:** *There exists a record with* `ID` *value p in X.*
- **action:** *Deletes the record with* `ID` *value p from X.*
- **return value:** *This method has no return value.*

3. `reference search(attr` A_1`, param` P_1`, ..., attr` A_l`, param` P_l`, direction` dir`, int` r`):`

- **pre-conditions:** $l \leq a$ *and* $r \in [l-1, l]$.
- **action:** *This method does not change X.*
- **return value:** *If $dir =$* `asc` *($dir =$* `desc`*), then this method returns the* `ID` *of the first (last), relative to the order defined by X, tuple t for which:*

  (a) $\bigwedge_{i=1}^{r} (t.A_i = P_i)$ *and*

  (b) if $r < l$, then $t.A_l > P_l$ when $dir_l = dir$ and $t.A_l < P_l$ when $dir_l \neq dir$,

  when such a tuple t exists. The method returns `NULL` *when such a tuple does not exist.*

4. `tuple retrieve(reference` p`):`

- **pre-conditions:** *There exists a tuple with* `ID` *p in X.*
- **action:** *This method does not change X.*
- **return value:** *This method returns the value of the tuple with* `ID` *p from X.*

5. `reference next(reference` p`):`

- **pre-conditions:** *There exists a record with* `ID` *value p in X.*
- **action:** *This method does not change X.*

- **return value:** *The method returns the* `ID` *of the tuple immediately after the tuple with* `ID` *p, relative to the order defined by X, when such a tuple exists. The method returns* `NULL` *when such a tuple does not exist.*

6. `reference previous(reference` p`):`

- **pre-conditions:** *There exists a record with* `ID` *value p in X.*
- **action:** *This method does not change X.*
- **return value:** *The method returns the* `ID` *of the tuple immediately before the tuple with* `ID` *p, relative to the order defined by X, when such a tuple exists. The method returns* `NULL` *when such a tuple does not exist.*

Theorem 1 (running time for the methods of a regular index) *The specified running times of the different methods of a regular index are correct.*

Proof: First, consider the `retrieve`(p) method. The operation involves locking the search tree, using the `ID` look-up structure to find the address of the record with certain `ID`, returning the found record, and unlocking the search tree. If the search tree has a node capacity of 1, then the look-up will involve performing a hash search in the `global look-up` table with value p. Otherwise, the look-up will involve performing a hash search in the `local look-up` table with physical design value of X and `ID` value of p. Note that locking the search tree guarantees that the found address in the `ID` look-up structure is valid. If the search tree was not locked, it might be the case that some procedure (e.g. balancing the search tree) could have changed the record's address. If increasing concurrent access to the search tree is important, than we could, for example, try implementing a tree locking protocol ([7]) that allows us to only lock part of the search tree. Assuming that the search in the `ID` look-up structure takes constant time, this method runs in worst case constant time.

Next, consider the `insert(`t`)` method. Executing this method is done in two fazes. The first faze involves locking the search tree, finding where the tuple is to be inserted, inserting the tuple, generating an `ID` value for t, updating the `ID` look-up structure, and unlocking the search tree. Finding where the tuple is to be inserted will involve $O(a)$ comparisons at each visited node of the search tree and at most $O(log(|X|))$ nodes are visited. In the second faze rebalancing of the search tree may be required. Note that rebalancing the search tree can be viewed as a sequence of node deletions and node insertions. The worst case running time for each of the two fazes is $O(a \cdot log(|X|))$ because each faze involves visiting at most $O(log(|X|))$ nodes in the search tree and doing at most $O(a)$ work at each visited node.

Next, consider the `delete(`p`)` method. Similarly to the `insert` method, we will describe the execution of this method in two fazes. The first faze involves locking the search tree, using the `ID` look-up structure to find the address of the record with `ID` p, deleting the record found at this address, releasing the `ID` p from the `ID` look-up structure for reuse, and unlocking the index. In the second faze rebalancing of the search tree may be required. The worst case running time for this method is $O(a \cdot log(|X|))$.

Next, consider the `search(`$A_1, P_1, \ldots, A_l, P_l, dir, r$`)` method. The method involves locking the search tree, searching for a tuple in X, and then unlocking search tree. The search visits $O(log(|X|)$ nodes and $O(l)$ comparisons are done at each node. Therefore, the worst case running time for this method is $O(a \cdot log(|X|))$.

Lastly, consider the `next(`p`)` (`previous(`p`)`) method. The method involves locking the search tree, using the `ID` look-up structure to find the address of the tuple with `ID` p, finding the tuple after (before) the tuple stored at the found location relative to the index order, and unlocking the search tree. The worst case running time for this method is $O(a \cdot log(|X|))$. ■

We next show how we can avoid specifying the direction of the attributes in a regular index and still get the interface and performance specified in Definition 19.

Theorem 2 *The regular index* $X_1 = \langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle\rangle$ *can be substituted with the regular index* $X = \langle R, \langle A_1\ \texttt{asc}, \ldots, A_a\ \texttt{asc}\rangle\rangle$ *regardless of the values for* $\{dir_i\}_{i=1}^{a}$ *and still get the interface and performance specified in Definition 19.*

Proof: Consider first the methods `insert`, `delete`, and `retrieve` for X_1. They can be executed by performing them on X without any change. The pseudo-code for X_1`.next(`p`)` is as follows.

```
(01) reference next(reference p) {
(02)     tuple t = X.retrieve(p);
```

```
    if $(dir_a = \texttt{asc})$ reference $p' = X.\texttt{next}(p)$;
    if $(dir_a = \texttt{desc})$ reference $p' = X.\texttt{previous}(p)$;
    if $(p' \neq NULL)$ {
      $t' = X$.retrieve$(p')$;
      if $(\bigwedge_{i=1}^{a-1}(t.A_i = t'.A_i))$ return $p'$;
    }
    if $(a == 1)$ return NULL;
    reference $p' = X.\texttt{search}(A_1, t.A_1, \ldots, A_{a-1}, t.A_{a-1}, dir_{a-1}, a-2)$;
    return next_up$(p, p', a-1)$;
}
```

```
reference next_up(reference $p$, reference $p'$, level $l$) {
    $t = X$.retrieve$(p)$;
    if $(p'$ ==NULL) {
      if $(l == 1)$ return NULL;
      reference $p' = X.\texttt{search}(A_1, t.A_1, \ldots, A_{l-1}, t.A_{l-1}, dir_{l-1}, l-2)$;
      next_up$(p, p', l-1)$;
    }
    $t' = X$.retrieve$(p')$;
    return next_down$(p', l, \texttt{asc})$;
}
```

```
reference next_down(reference $p$, level $l$, direction $dir$) {
    if $(p ==$ NULL$)$ return NULL;
    if $(l == a)$ return $p$;
    tuple $t = X$.retrieve$(p)$;
    reference $p' = X.\texttt{search}(A_1, t.A_1, \ldots, A_l, t.A_l, \texttt{asc}^{dir_{l+1}==dir}, l)$;
    next_down$(p', l+1, dir)$;
}
```

Consider first Lines 1-12 of the `next` method. The code first checks whether the next tuple relative to the defined order in X_1 has the same values for the attributed $A_1, \ldots, A_{a-1}$ as t. If this is the case, then only X.`next` or X.`previous` needs to be executed and the result returned. If this is not the case and $a = 1$, then a *next* tuple does not exist and the method returns NULL. If this is not the case and $a > 1$, then Line 10 of the code looks for the tuple that has the same value as t for the attributes $\{A_i\}_{i=1}^{a-2}$ and the next value for the attribute A_{a-1}, relative to the order defined in X_1, and calls the `next_up` method with the found tuple, if any.

The method `next_up` tries to find a tuple t' that has the property that $\bigwedge_{i=1}^{l-1}(t.A_i = t'.A_i)$ and the value for A_l of t' is right after the value for A_l of t relative to the order defined in X_1. When this is the case, then the `next_down` method is executed. It finds the first tuple in the index, relative to the order defined by X_1, for which $\bigwedge_{i=1}^{l-1}(t.A_i = t'.A_i)$ holds. This is indeed the correct return tuple. When t' does not have the desired property, l is decremented by 1 and `next_up` is called recursively. If l becomes 0, then a `next` tuple does not exist and the `next_up` methods returns `NULL`.

The X_1.`previous` method can be implemented using the regular index X in an analogous fashion. The X_1.`search`$(A_1, P_1, \ldots, A_l, P_l, dir, r)$ method can be implemented using the following pseudo-code.

(1) `reference search(attr` A_1, `param` $P_1, \ldots,$ `attr` A_l, `param` P_l`, direction` dir`, int` r) {
(2) $\quad$ $p = X$.`search`$(A_1, P_1, \ldots, A_l, P_l, \texttt{asc}^{dir==dir_l}, r)$;
(3) $\quad$ `return next_down`(p, l, dir)
(4) }

The pseudo-code first calls the search method for X. This method search for the corresponding tuple in X, where the search direction is set in a way to simulate a search in X_1. This can be done because the `search` method searches for a tuple with fix values for $\{A_i\}_{i=1}^{l-1}$ and therefore only the direction for A_l is important when searching for the value according to the first l attributes. After that the `next_down` method is called, which changes the search direction for the attributes $\{A_i\}_{i=l+1}^{a}$ one by one by calling the search method for X_1 $a - l$ times - ones for each attribute in the set $\{A_i\}_{i=1}^{l-1}$.

$(name)$	(MV)
V_1	`select` age `from EMP`
V_2	`select` $name$, age, $salary$ `from EMP` `where` $age < 30$
V_3	`select` $name$, age, $salary$ `from EMP` `where` $salary > 100K$
V_4	`select` age, $salary$ `from EMP` `where` $age \geq 30$ and $salary \leq 100K$

Table 2.7: Four example MVs

Note that the pseudo-code for each of the methods for X_1 has $O(a \cdot log|X_1|)$ worst case running time, except for the `retrieve` method which has constant worst case running time. ■

An extended index differs from a regular index in three ways. First, the different records inside the index can store values of different attributes. Consider for example the MVs V_3 and V_4 from Table 2.7 that are defined over our example company schema from Figure 1.5 and note that the two MVs are disjoint. An extended index can contains the records from $V_3 \cup V_4$, where the records for the MV V_3 will have the attributes *name*, *age*, and *salary*, while the records for the MV V_4 will have only the attributes *age* and *salary*.

The second novel feature of an extended index is that the order of the index could be in the form of a *branching condition*. An example of a branching condition for the table `EMP` from our schema from Figure 1.5 follows.

- First, the tuples in the table `EMP` are ordered according to their *name* in ascending order.

- Next, if two or more tuples have the same *name*, then the tuples are ordered according to the newly introduced derived attribute A in ascending order, where, for a tuple t

(*name*)	(*query*)

Q_1	`select *`
	`from EMP`
	`order by` age `asc`

Q_2	`select *`
	`from EMP`
	`where` $salary > 100\texttt{K}$ `and` $age = :P_1$

Q_3	`select *`
	`from EMP`
	`where` $name =$ “`Joe`” `and` $age = :P_2$

Table 2.8: Three example critical queries

in the table `EMP`, $t.A = 0$ if $t.age < 40$ and $t.A = 1$ otherwise.

- Next, if two tuples have the same value for the attributes *name* and A, then they are ordered according to their *age* in descending order if $A = 0$ and according to their *salary* in ascending order if $A = 1$.

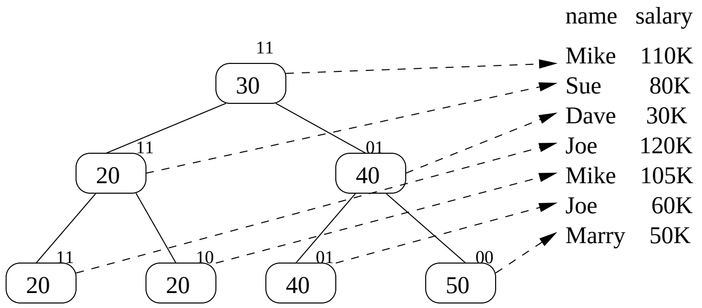

Figure 2.4: Example marked index for queries Q_1, Q_2, and Q_3 from Table 2.8

The third peculiarity of an extended index is the introduction of marking bits that allow us to efficiently find all nodes in the search tree that contain records that pass certain efficient predicate. Consider the three queries from Table 2.8 defined over out example company schema from Figure 1.5 and suppose that we want to create a single index that can answer all three queries. If we know that the employees that make more than 100K are disjoint from the employees named "Joe", then we can answer all three queries by an index on the table `EMP` and a branching condition of the form.

- First, the tuples in `EMP` are ordered according to their *age* in ascending order.
- Next, if two or more tuples have the same *age*, then the tuples are ordered according to the newly introduced derived attribute A in ascending order, where, for a tuple t in the table `EMP`, $t.A = 0$ if $t.salary > 100K$, $t.A = 1$ if $t.name =$"Joe", and $t.A = 2$ otherwise.

However, if the employees that make more than $100K$ are not disjoint from the employees that are named "Joe", then we need to use marking bits. Figure 2.4 shows an example instance of the search tree of the regular index $\langle \texttt{EMP}, \langle age\ \texttt{asc} \rangle\rangle$ in which there are two marking bits: one for the predicate $salary > 100\texttt{K}$ and one for the predicate $name =$ "*Joe*". In the figure each node contains a single entry, although in general this does not need to be the case. The value of the *age* attribute is shown inside the node, while the *name* and *salary* attribute are shown on the right side of the figure. The index is ordered according to name *asc*. Each node has two extra bits that are shown above the node. The i^{th} bit of a node is set to 1 iff the node or one of its descendants in the search tree contain a record for which γ_i is true, where $i = 1$ or $i = 2$, $\gamma_1(t)$ is the predicate $t.salary > 100K$, and $\gamma_2(t)$ is the predicate $t.name$="Joe". The introduced marking bits allow us to prune out branches of the search tree that do not contain records that contribute to the query result and, as a result, all three queries from Table 2.8 can be answered using the described index in logarithmic time relative to the size of the `EMP` table.

A formal definition of an extended index follows.

Definition 20 (syntax of an extended index) *An extended index is represented by a node labeled ordered tree G^t. The label of the root node of G^t is of the form $\langle\{\gamma_1, \ldots, \gamma_m\}, R,$ $\langle A_1, \ldots, A_a\rangle\rangle$ and the labels of the non-root nodes of G^t are of the form $\langle R, \langle A_1, \ldots, A_a\rangle\rangle$ where R is a table, $\langle A_1\ \texttt{asc}, \ldots, A_a\ \texttt{asc}\rangle$ is a valid order condition over R, and $\{\gamma_i\}_{i=1}^m$ are efficient predicates over R. In order for an extended index to be valid, we require that:*

1. *Let the node n with table R be the parent of the nodes $\langle n_1, \ldots, n_k \rangle$ with tables $\langle R_1, \ldots, R_k \rangle$, respectively, and let $\{\gamma_i\}_{i=1}^{k}$ be such that $R_i = \gamma_i(R)$, $i = 1$ to k. Then:*

   (1) $R_i \subseteq_{\mathtt{ID}} R$ *for* $i = 1$ *to* k,

   (2) $\pi_{\mathtt{ID}} R_i \cap \pi_{\mathtt{ID}} R_j = \emptyset$ *for* $i \neq j$ *and* $i, j \in [1 \ldots k]$,

   (3) $\bigcup_{i=1}^{k} \pi_{\mathtt{ID}} R_i = \pi_{\mathtt{ID}} R$,

   (4) $Attr(R) \subseteq Attr(R_i)$ *for* $i \in [1 \ldots k]$, *and*

   (5) For all $i \in [1 \ldots k]$, γ_i can be defined as an efficient predicate using the attributes of R.

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

3. *If $\langle R, \langle A_1, \ldots, A_a \rangle \rangle$ or $\langle \{\gamma_1, \ldots, \gamma_m\}, R, \langle A_1, \ldots, A_a \rangle \rangle$ is the label of a leaf node, then the attributes $\{A_i\}_{i=1}^{a}$ form a key for the table R.*

In this theses we describe several data structures using trees (an extended index is just one example). We will call such trees: *description trees*. On the other hand, we will use the term *search tree* to refer to the data structure that implements an index. Note that although the terms *description tree* and *search tree* both describe an index, they do so at different levels. While a description tree describes an index at the logical level, i.e. what records are indexed and in what order, a search tree describes the index at the physical level, i.e. it describes the nodes of the index, the records in each node, and how are the nodes connected. Note that there may be several search trees that conform to the requirements of a description tree.

Definition 21 (node label) *We use* `label`(n) *to denote the label of a node n in a node labeled tree.*

The node labeled trees examined in this work all have nodes with labels of the form $\langle \ldots \rangle$. In order to be able to compactly refer to such trees, we define the concept of a *string description* for such trees.

Definition 22 (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 $\langle \ldots \rangle$. 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 (ordered) children $n_1, \ldots, n_k$ we defined $\mathcal{L}^{\downarrow}(n) = \langle L, [\mathcal{L}^{\downarrow}(n_1), \ldots, \mathcal{L}^{\downarrow}(n_k)] \rangle$. 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. Throughout this work we will commonly use the string description of a tree to refer to the tree and the entity the tree represents.*

Note that the labeling function $\mathcal{L}^{\downarrow}$ produces information about a node and its descendants. We will also need a labeling function that produces information about a node and its predecessors.

Definition 23 (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 for 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 $\mathcal{L}^{\uparrow}(n) = \langle L', L \rangle$.*

Consider an extended index X that has a description tree G^t. Let n^r be the root of the description tree and n be a non-root node of the tree. Then the label of n^r will have the syntax $\langle \{\gamma_1, \ldots, \gamma_m\}, R, \langle A_1, \ldots, A_a \rangle \rangle$ and the label of n will have the syntax $\langle R, \langle A_1, \ldots, A_a \rangle \rangle$. For convenience, we introduce the following naming conventions.

- We will refer to R as the table of the node and write $\texttt{table}(n^r)$ and $\texttt{table}(n)$, respectively.

- We will refer to $\{\gamma_1, \ldots, \gamma_m\}$ as the γ-condition of X and write $\gamma(X)$. In order to be able to refer to a specific efficient predicate in a γ-condition, we assume that the predicates are ordered in some way. We will refer to a γ-condition as *trivial* if it is the empty set.

- We will refer to $\{\texttt{TRUE}\} \cup \{\texttt{FALSE}\} \cup \{\bigcup_{\emptyset \not\equiv \bar{\gamma} \subseteq \gamma(X)} \bigvee_{\gamma \in \bar{\gamma}} \gamma\}$ as the extended γ condition of X and write $\gamma^e(X)$.

- We will refer to $\langle A_1, \ldots, A_a \rangle$ as the *ordering label* of the node and write $\mathcal{L}(n^r)$ and $\mathcal{L}(n)$, respectively.

In order to understand the meaning of the extended γ-condition for an extended index, we need to first know the physical design that such an index represents. The semantics of an extended index in terms of the physical design that it represents follows.

Definition 24 (semantics of an extended index) *We define the semantics of an extended index X by first describing the elements that are indexed and the order in which they are indexed in the search tree of the index. If $n_1, \ldots, n_k$ are the leaf nodes of the description tree of X and they have table $\langle R_1, \ldots, R_k \rangle$, respectively, then the extended index indexes the elements $\bigcup_{i=1}^{k} R_i$. For each node n of the description tree we define an ordering function* $\texttt{Or}$ *that is a total order for the tuples in* $\texttt{table}(n)$*. If n is a leaf node and $\mathcal{L}(n) = \langle A_1, \ldots, A_a \rangle$, then we define* $\texttt{Or}(n) = \langle A_1\ \texttt{asc}, \ldots, A_a\ \texttt{asc} \rangle$*. If n is a non-leaf node with children $\langle n_1, \ldots, n_k \rangle$,* $\texttt{table}(n) = R$*, and $\mathcal{L}(n) = \langle A_1, \ldots, A_a \rangle$, then we define* $\texttt{Or}(n)$ *to be the following ordering.*

1. *The tuples are first ordered relative to the order* $\langle A_1\ \texttt{asc}, \ldots, A_a\ \texttt{asc} \rangle$*.*
2. *Next, if two ore more tuples have the same value for the attributes in $\{A_i\}_{i=1}^{a}$, then they are ordered relative to the attribute A in ascending order, where $t.A = i$ iff* $t \in \texttt{table}(n_i)$ *($i \in [1..k]$).*
3. *Finally, if two ore more tuples have the same value for the attributes $\{A_i\}_{i=1}^{a}$ and for the attribute A, then they are ordered relative to* $\texttt{Or}(n_i)$*, where i is the common value for the attribute A.*

Let n^r be the root node of the description tree of X. Then we define the ordering of the index as $\texttt{Or}(n^r)$*.*

We next define the placement of the marking bits in the index. If the index has a non-trivial γ-condition of the form $\{\gamma_1, \ldots, \gamma_m\}$, then we will associate with each node in the search tree of X m marking bits. The j^{th} bit of a node will be marked iff the node or one of its descendants in the search tree of X contains a record for which γ_j holds ($j \in [1 \ldots m]$).

The above definition described the physical design for a single extended index. However, note that we will define the physical design for a set of extended indices to be different from the union of the physical designs of the indices. The reason is that two or more extended indices may store tuples with the same `ID` values and, when possible, those tuples will be stored on top of each other in order to save space. Note that such a compression can only be achieved between extended indices that have node capacity of one. A formal definition of the physical design that we will associate with a set of extended indices follows.

Definition 25 (semantic of a set of extended indices) *Let $\bar{X}_1$ be a set of extended indices based on search trees with node capacity of one and let $\bar{X}_2$ be a set of extended indices based on search trees with node capacity greater than one. The physical design for $\bar{X}_1 \cup \bar{X}_2$ consists of the union of the physical designs for $\bar{X}_2$ union the physical designs for $\bar{X}_1$. The physical design for $\bar{X}_1$ is defined as the union of the physical design of the indices, where records with the same* `ID` *value are stored on top of each other. This implies that different records may have different attributes and therefore the information of where each attribute of each record is stored needs to be encoded inside the record. This can be done, for example, by using the data structures proposed in [42] or [43]. For each tuple in the set $\bar{X}_2$ and for each extended index it appears in, there will be an entry in the local look-up table. On the other hand, for each tuple in the set $\bar{X}_1$ there will be a single entry in the global look-up table.*

We next present a lemma that sheds light on the meaning of an extended γ-condition.

Lemma 1 *Let X be an extended index. $\gamma \in \gamma^e(X)$ implies that adding γ to the γ-condition of X will not change the set of queries that can be efficiently answered by X. The reverse is also true, i.e. if $\gamma \notin \gamma^e(X)$, then adding γ to the γ-condition of X will increase the set of queries that can be efficiently answered by X.*

Proof:

Note that $\gamma \in \gamma^e(X)$ implies that adding γ to the γ-condition of X will not change the set of queries that can be efficiently answered by X. Obviously, we do not need marking bits for the `TRUE` and `FALSE` predicates. On the other hand, if $\gamma \equiv \gamma_1 \vee \ldots \gamma_k$ and $\{\gamma_i\}_{i=1}^k$ correspond to marking bits in the index, then the marking bit for γ will be set exactly when one of the marking bits for $\{\gamma_i\}_{i=1}^k$ is set. The reason is that if one of $\{\gamma_i\}_{i=1}^k$ holds for a record in a node in the search tree of X, or for a record in one of the node's descendants, then γ will hold for a record in the node or one of its descendants. The reverse is also true, i.e. if $\gamma \notin \gamma^e(X)$, then adding γ to the γ-condition of X will increase the set of queries that can be efficiently answered by X. The reason is that $\gamma \notin \gamma^e(X)$ implies that γ is not the `TRUE` or `FALSE` predicate and it can not be expressed as a disjunction of some of the other γ-condition of the index. However, if γ_1 is in the γ-condition of the index, then we will need a separate bit for $\neg\gamma_1$. The reason is that the fact that γ_1 is `TRUE` for a record in a node in the search tree or for a record in one of the node's descendants, does not tell us whether $\neg\gamma_1$ is `TRUE` for a record in the node or for a record in one of the node's descendants. Similarly, if γ_1 and γ_2 are in the γ-condition of the index, then we will need a separate bit for $\gamma_1 \wedge \gamma_2$. The reason is that the fact that the marking bits γ_1 and γ_2 are marked in a node n in the search tree implies that γ_1 is `TRUE` for a record in n or for a record in one of the descendants of n and γ_2 is `TRUE` for a record in n or for a record in one of the descendants of n. However, it might be the case that γ_1 and γ_2 are true for records in different descendants of n, which implies that the value for the marking bit for $\gamma_1 \wedge \gamma_2$ cannot be inferred from the value of the marking bits for γ_1 and γ_2. ■

We next introduce two intermediate definitions, which will be useful in defining the interface of an extended index.

Definition 26 (rooted path in a tree) *A rooted path in a tree is any path in the tree that starts from the root or the tree. We will denote by $rp(G^t)$ the set of rooted paths in the tree G^t.*

Definition 27 (clustering property) *Let $K = \langle n_1, \ldots, n_k \rangle$ be a rooted path in the description tree of the extended index X and let the attributes in $\mathcal{L}^{\uparrow}(n_k)$ be $\langle A_1, \ldots, A_b \rangle$ in this order. Given an integer a, $0 \leq a \leq b$, we will say that K is* clustered *relative to the integer a iff one of the following conditions holds.*

1. *$k = 1$.*
2. *$\mathcal{L}(n_1)$ contains exclusively attributes from the set $\{A_i\}_{i=1}^{a}$ and if n_r is the node in K with the biggest subscript which ordering label contains exclusively attributes from the set $\{A_i\}_{i=1}^{a}$, then either $r = k$ or $k = r + 1$.*

The following lemma explains the meaning of the clustering property.

Lemma 2 *Let X be an extended index and $K = \langle n_1, \ldots, n_k \rangle$ be a rooted path in description tree of the extended index. Let* `table`$(n_k) = R_k$ *and $\langle A_1, \ldots, A_b \rangle$ be the attributes that appear in $\mathcal{L}^{\uparrow}(n_k)$ in order of appearance. Then the tuples in R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ are clustered together relative to the order defined by X (i.e. if a tuple in X is between two tuples in X, relative to the order defined by X, that belong to R_k and that have the same value for the attributes $\{A_i\}_{i=1}^{a}$, then this tuple must also belong to R_k and have these values for the attributes $\{A_i\}_{i=1}^{a}$) iff K is clustered relative to a.*

Proof: $\Leftarrow$ First, assume that K is clustered relative to a. This means that one of the two condition from Definition must hold. If $k = 1$, then the tuples in the index will be ordered relative to the order $\langle A_1$ `asc`$,\ldots, A_b$ `asc` $\rangle$ and all the tuple in the index will be in R_k. However, since $a \leq b$, the tuples in R_k will be also ordered relative to the order $\langle A_1$ `asc`$,\ldots, A_b$ `asc` $\rangle$, which implies that the records in the index that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ will be clustered together relative to the order defined by X.

Next, consider the case when Condition 2 from Definition 2.2.1 holds. The tuples in R_k will be ordered relative to the tuple ordering $\langle A_1$ `asc`$,\ldots, A_b$ `asc` $\rangle$. If $r = k$, then $a = b$. This implies that the tuples in R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ will be clustered together. Next, consider the case when $k = r+1$. From Definition 25 it follow that the tuples in R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ will be clustered together again.

$\Rightarrow$ Now assume that the tuples in R_k that have the same value for the attributes $\{A_i\}_{i=1}^{a}$ are clustered together relative to the order defined by X and that K is not clustered relative to a. This implies that $k \neq 1$. As well, if n_r is the node in K with the biggest subscript that contains exclusively attributes from the set $\{A_i\}_{i=1}^{a}$, then $r \neq k$ and $k \neq r+1$. But this implies that in the branching condition of the index order the attributes $\langle A_1, \ldots, A_{a+1}$ will appear before the branching condition for R_k, which implies that the tuples in R_k with the same value for the attribute $\{A_i\}_{i=1}^{a}$ do not need to be clustered together. ■

Definition 28 (interface of an extended index) *An extended index X has the following capabilities, where G^t is the description tree of X.*

1. `reference insert(tuple` t`):`

- **pre-conditions:** *(1) There exists a node in G^t with table R such that the tuple t is compatible with the table R, (2) t is not already inserted in X.*
- **action:** *The tuple t is inserted in the index X.*
- **return value:** *This method returns the* `ID` *of the inserted tuple.*

2. `delete(reference` p`)`

- **pre-conditions:** *There exists a record with* `ID` *value p in X.*
- **action:** *Deletes the record with* `ID` *value p from X.*
- **return value:** *This method has no return value.*

3. `reference search(table` R_r, `attr` A_1, `param` $P_1, \ldots,$ `attr` A_a, `param` P_a, `direction` $dir_1, \ldots,$ `direction` dir_b, `efficient predicate` γ, `int` m) :

- **pre-conditions:**
  (a) *There exists a node n_r in G^t with table R_r. Let $\langle n_1, \ldots, n_r\rangle$ be the path from the root of G^t to the node n_r. We will refer to this path as K.*
  (b) $\gamma \in \gamma^e(X)$.
  (c) *The attributes in $\mathcal{L}^{\uparrow}(n_r)$ are exactly $\langle A_1, \ldots, A_b\rangle$ in this order, where $1 \leq a \leq b$.*
  (d) $m = a$ *or* $m = a - 1$.
  (e) *Either K is clustered relative to m or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$* `TRUE` *iff $t \in \gamma(R_r)$.*
- **action:** *This method does not change X.*
- **return value:** *Let O be the order $\langle A_1\ dir_1, \ldots, A_b\ dir_b\rangle$. This method returns the* `ID` *of the first tuple t in X, relative to the order O, for which:*
  (a) $t \in R_r$,
  (b) $\gamma(t)$ *holds,*
  (c) $\bigwedge_{i=1}^{m}(t.A_i = P_i)$, *and*
  (d) *if $m < a$ then $t.A_a > P_a$ when $dir_a =$* `asc` *and $t.A_a < P_a$ when $dir_a =$* `desc`.

If such a tuple does not exist, then the method returns `NULL`.

4. `tuple retrieve(reference` p`):`

   - **pre-conditions:** *There exists a tuple with* `ID` p *in* X.
   - **action:** *This method does not change* X.
   - **return value:** *This method returns the value of the tuple with* `ID` p *from* X.

5. `reference next(table` R_r`, attribute` A_1`, parameter` P_1`, ..., attribute` A_a`, parameter` P_a`, direction` $dir_1, \ldots,$ `direction` dir_b`, reference` p, `efficient predicate` γ`):`

   - **pre-conditions:**
     (a) *There exists a node* n_r *in* G^t *with table* R_r. *Let* $\langle n_1, \ldots, n_r \rangle$ *be the path from the root of* G^t *to the node* n_r. *We will refer to this path as* K.
     (b) $\gamma \in \gamma^e(n_1)$.
     (c) $\mathcal{L}^{\uparrow}(n_r)$ *contains exactly the attributes* $A_1, \ldots, A_b$ *in this order and* $a \leq b$.
     (d) $p = 0$ *or there exists a tuple* t *with* `ID` *value* p *in* X *and* $t \in R_r$.
     (e) *Either* K *is clustered relative to* a *or there exists a* $\gamma' \in \gamma^e(n_1)$ *that has the property that* $\gamma'(t) =$ `TRUE` *iff* $t \in \gamma(R_r)$.
   - **action:** *This method does not change* X.
   - **return value:** *Let* O *be the order* $\langle A_1\ dir_1, \ldots, A_b\ dir_b \rangle$. *If* $p = 0$, *then the method returns the* `ID` *of the first tuple* $t' \in R_r$, *relative to the order* O, *for which* $\gamma(t')$ *and* $\bigwedge_{i=1}^{a} t'.A_i = P_i$ *both hold, when such a tuple exists. If* $p \neq 0$, *then the method returns the* `ID` *of the first tuple* $t' \in R_r$, *relative to the order* O, *after the tuple* t *for which* $\gamma(t')$ *and* $\bigwedge_{i=1}^{a} t'.A_i = P_i$ *both hold, when such a tuple exists. In both cases, when the described tuple does not exist, the method returns* `NULL`.

Note that a simplified version of an extended index in which the order condition is in the form of a path rather than a tree was first introduced by Weddell in [37].

Note that pre-condition (e) for the `search` method guarantees that either the tuples that belong to R_m are clustered together in the index or a marking bit that tells whether $\gamma(R_r)$ holds can be computed. This condition is necessary to guarantee the logarithmic running time of the method. Similarly, pre-condition (e) for the `next` method guarantees that either the tuples that belong to R_r and have the same value for the attributes $\{A_i\}_{i=1}^{a}$ are clustered together in the index or a marking bit that tells whether $\gamma(R_r)$ holds can be computed. This condition is necessary to guarantee the logarithmic running time of the method.

We next prove the running time for the methods of the interface of an extended index.

Theorem 3 (running time for the methods of an extended index) *Let the size of each of the indexed tuples and the node capacity of the search tree is constant and let $|X|$ be used to denote the number of tuples in the extended index X. Then each of the methods that comprise the interface of X has worst case running time of $O(|def(X)| \cdot log(|X|))$, except for the method* `retrieve`*, which has constant worst case running time.*

Proof: The `retrieve`(p) method can be performed the sam way it was performed on a regular index and therefore it has the same running time.

The `insert(`t`)` method for an extended index differs from the `insert` method for a regular index in two ways. The first difference relates to how the search for the tuple t is done. When performing this search for an extended index, we first need to find the table in the description tree of X to which t belongs. Then, we need to navigate in the search tree to find a place for t. Such a navigation will involve deciding whether a tuple in the search tree belongs to a certain table in the description tree. This can be done in $O(|def(X)|)$ time because it involves only checking whether tuples pass certain efficient predicated and the definition of an extended index guarantees that each tuple of an extended index contains the attributes on which the possible predicates are based.

The second difference is that the marking bits in the search tree may need to be updated after the insertion of t is performed and after a rebalancing of the search tree is performed. Next, we describe only the steps that need to be taken after the insertion of a tuple t because a rebalancing of the search tree can be described as a sequence of insertions and deletions and therefore the marking bits can be updated after each such operation. If the

γ-condition of X is not trivial, then the insertion will also involve checking whether $\gamma_j(t)$ is true for some $j = 1$ to m. If this is the case and the j^{th} bit in the node in which t is inserted is not marked, then we will mark the j^{th} bit in this node and its ancestors. The marking of an ancestor node is done by first checking the parent of the current node at each step and marking the parent's j^{th} bit if it is not marked. The process continues until the root node of the search tree is reached or until a node with marked j^{th} bit is reached (once such a node is reached, we are guaranteed that all ancestor nodes will have their j^{th} bit marked). Since the worst case running time for implementing each of the two differences is $O(|def(X)| \cdot log(|X|))$, the running time of this method is the same as the running time for `insert` method for a regular indices.

Next, consider the `delete(`p`)` method. It differs from the `delete` method for a regular index because it involved updating the marking bits of the search tree when this is necessary. Let t be the deleted tuple and suppose that the γ-condition of X is not trivial. Then the deletion would also involve checking whether the deleted tuple is the last tuple in its node in the search tree of X for which γ_j holds for some $j = 1$ to m. If this is the case, then the j^th bit of the node is unmarked. The procedure is repeated up the search tree until the root node is reached or until a node n that contains a record for which γ_j holds is reached. Assuming that the node capacity of X is a constant, each of these checks can be done in constant time. Since the worst case running time for implementing this difference is $O(|def(X)| \cdot log(|X|))$, the running time of this method is the same as the running time for the `insert` method of a regular index.

Next, consider the `search(`$R_r, A_1, P_1, \ldots, A_a, P_a, dir_1, \ldots, dir_b, \gamma, m$`)` method. Let G^t be the search tree for X in which the direction of A_i is substituted with dir_i for $i \in [1..b]$. Then this method can be executed by performing a modified version of the regular index method `closest_search(`$A_1, P_1, \ldots, A_a, P_a,$ `asc`$, m$`)` on G^t. The modification relates to the fact that the method should search for a tuple in R_r that passes the condition γ. In particular, consider the two case: when K is clustered relative to m and when it is not.

In the first case, the algorithm will look for the first tuple t in R_r, relative to the order defined by G^t, that passes the efficient predicate γ, $\bigwedge_{i=1}^{m}(t.A_i = P_i)$, and if $m < a$ then $t.A_a > P_a$ when $dir_a =$ `asc` and $t.A_a < P_a$ when $dir_a =$ `desc`. However, in this case the tuples in R_r that have the same value for the attributes $\{A_i\}_{i=1}^{m}$ will be clustered together relative to the order defined by G^t and therefore the modified version of the method would take $O(|def(X)| \cdot log(|X|))$ time.

In the second case, the algorithm will look for the first tuple t in R_r, relative to the order defined by G^t, that passes the efficient predicate γ, $\bigwedge_{i=1}^{m}(t.A_i = P_i)$, and if $m < a$ then $t.A_a > P_a$ when $dir_a =$ `asc` and $t.A_a < P_a$ when $dir_a =$ `desc`. This can be also be done in $O(|def(X)| \cdot log(|X|))$ time because $\gamma' \in \gamma^e(X)$.

Lastly, consider the `next`$(R_r, A_1, P_1, \ldots, A_a, P_a, dir_1, \ldots, dir_b, p, \gamma)$ method. First, consider the case when $p = 0$. This method can be implemented using a modified version of the `exact_search`$(A_1, P_1, \ldots, A_a, P_a,$ `asc`$)$ method for the regular index that corresponds to X in which the directions of the attributes along the path that starts at the root of the description tree of X and goes down to the node with table R_r are changed to $\langle dir_1, \ldots, dir_b\rangle$, respectively. Let G^t be the search tree for the described modified search tree of X. We will examine two subcases: when K is clustered relative to a and when it is not.

In the first subcase, the algorithm will look for the first tuple t in R_r, relative to the order defined by G^t, that passes the efficient predicate γ and for which $\bigwedge_{i=1}^{a}(t.A_i = P_i)$ holds. However, in this case the tuples in R_r with the same value for the attributes $\{A_i\}_{i=1}^{a}$ are clustered together relative to the order defined by G^t and therefore the modified version of the search can be done in $O(|def(X)| \cdot log(|X|))$ time.

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

Second, consider the case when $p \neq 0$. This method can be implemented using a modified version of the `next` method for the regular index that corresponds to X in which the directions of the attributes along the path that starts at the root of the description tree of X and goes down to the node with table R_r are changed to $\langle dir_1, \ldots, dir_b\rangle$, respectively. Let G^t be the search tree for the described modified search tree of X. We will examine two subcases: when K is clustered relative to a and when it is not.

In the first subcase, the algorithm will look for the first tuple t' in R_r after the tuple t (recall that this is the tuple with `ID` p), relative to the order defined by G^t that passes the efficient predicate γ and for which $\bigwedge_{i=1}^{a}(t'.A_i = P_i)$ holds. However, in this case the tuples in R_r with the same value for the attributes $\{A_i\}_{i=1}^a$ are clustered together relative to the order defined by G^t and therefore the modified version of the search can be done in $O(|def(X)| \cdot log(|X|))$ time.

In the second subcase, the algorithm will look for the first tuple t' in R_r after the tuple t (recall that this is the tuple with `ID` p), relative to the order defined by G^t, that passes the predicate γ' and for which $\bigwedge_{i=1}^{a}(t'.A_i = P_i)$ holds. This can be done in $O(|def(X)| \cdot log(|X|))$ time because $\gamma' \in \gamma^e(X)$.

Note that the fact that the described pseudo-codes can be transformed to run on the search tree for X with running time $O(|def(X)| \cdot log(|X|))$ follows from Theorem 2. ■

Consider our example company schema from Figure 1.5. If we are given the MVs shown in Table 2.7 and we know that V_2 and V_3 do not intersect, i.e. there is no employee that is younger than 30 years that makes more than 100K, then the extended index $\langle V_1, \langle age \rangle, [\langle V_2, \langle salary,\ \mathtt{ID}\rangle\rangle, \langle V_3, \langle name,\ \mathtt{ID}\rangle\rangle,\ \langle V_4, \langle \mathtt{ID}\rangle\rangle]\rangle$ is valid. The order is according to employees' ages first. Then, if two or more employees have the same age, then they are ordered according to the view in which they belong, where employees in V_2 come before employees in V_3 and employees in V_3 come before employees in V_4 (note that V_2, V_3 and V_4 are disjoin and their union covers all employees). Lastly, employees in V_2 with the same age are ordered according to their salary first and then according to the value of their `ID` attribute, employees in V_3 with the same age are ordered according to their *name* first and then according to the value of their `ID` attribute, and employees in V_4 with the same age are ordered according to the value of their `ID` attribute. Note that the extended index stores different attributes for different tuples, e.g. for employees in V_2 and V_3 the values for the *name*, *age*, and *salary* attributes are stored, while for employees in V_4 only the values for the attributes *age* and *salary* are stored. Using the described index one can efficiently insert and delete elements, can efficiently search for a value in V_1 or V_4 with given *age*, can efficiently search for a value in V_2 with a given *age* or given *age* and *salary*, and for a value in V_3 with given *age* or given *age* and *name*. Note that the described extended index also has the capability of efficiently answering range queries such as the one bellow.

`select` *name*
`from` V_2
`where` $age = {:}P_1$ and $salary > {:}P_2$

However, this index cannot efficiently answer the following query.

```
select name
from V_2
order by age asc, salary asc
```

Consider the scenario where each employee has a distinct *name* and none of the employees are younger than 30 years. Then we will need to scan all the tuples in the index in order to return an empty result, and therefore such a plan is not efficient. In order to be able to answer the query efficiently, we will need to add marking bits to the index. In particular, the extended index $\langle\{\gamma\}, V_1, \langle age\rangle, [\langle V_2, \langle salary, \mathtt{ID}\rangle\rangle, \langle V_3, \langle name, \mathtt{ID}\rangle\rangle, \langle V_4, \langle \mathtt{ID}\rangle\rangle]\rangle$ can efficiently answer both of the above queries, where $\gamma(t)$ holds for a tuple $t \in \mathtt{EMP}$ iff $t.age < 30$.

We next present several definitions and a theorem that describe the set of eSQL queries that can be efficiently answered using an extended index.

Definition 29 (queries of a rooted path of an extended index) *Let X be an extended index. Table 2.9 shows the set of queries that we will associate with a rooted path K of the index. We will denote this set as $Q_K(X)$. Note that we do not distinguish between two queries in which only the order in which the partial match restrictions are specified is changed, i.e. any query from Table 2.9 in which the order of $\{A_i\}_{i=1}^{l}$ is changed will also be in $Q_K(X)$.*

Definition 30 (critical queries supported by a physical design) *Let $\mathcal{D}$ be a physical design. Then the set of critical queries efficiently supported by $\mathcal{D}$, relative to the set of tables $\bar{R}$, is the set of queries that:*

- *are eSQL queries that do not contain non-parameterized restrictions , i.e. in Table 2.3 for queries of the first type $\gamma_i = \mathtt{TRUE}$ for $i = 1$ to k,*
- *reference only the tables in $\bar{R}$, and*
- *can be efficiently answered using $\mathcal{D}$.*

We will refer to this query set as $Q(\mathcal{D}, \bar{R})$.

Definition 31 (queries of an extended index) *Let X be an extended index defined over the database schema and $\bar{R}$ be the set of all tables with underlying queries of the form $\gamma_1(R_1)$* `union all` *…* `union all` *$\gamma_k(R_k)$, where $\{R_i\}_{i=1}^k$ are tables of nodes in the description tree of X and $\{\gamma_i\}_{i=1}^k$ are in the extended γ-condition of X. Then we define $Q(X)$ as $Q(X, \bar{R})$. We will refer to $Q(X)$ as the queries for the extended index X.*

Note that we define Q_1 `union all` Q_2 as the result of Q_1 union the result of Q_2, where Q_1 `union all` Q_2 is well defined iff $\pi_{\mathtt{ID}}Q_1 \cap \pi_{\mathtt{ID}}Q_2 \equiv \emptyset$ when Q_1 and Q_2 are materialized. Note as well that the we assume that the ordering of the result is not defined, i.e. any existing order in Q_1 and Q_2 is lost.

Definition 32 ($cl(\bar{Q})$) *Let $\bar{Q}$ be a set of queries. We define $cl(\bar{Q}) \equiv \{Q | \exists \{Q_1, \ldots, Q_k\} \subseteq \bar{Q}, Q \equiv \mathtt{merge\_sort}(Q_1, \ldots, Q_k, O)\}$, where* `merge_sort` *assumes that the results in $\{Q_i\}_{i=1}^k$ are ordered according to some tuple ordering O and it produces the result of the query "Q_1* `union all` *…* `union all` *Q_k" in the order O. Another pre-condition of the* `merge_sort` *method is that the results of the queries $\{Q_i\}_{i=1}^k$ are disjoint.*

Theorem 4 (efficiently supported queries by an extended index) *Let X be an extended index defined over the database schema Σ. Then $Q(X) = cl(\bigcup_{K \in rp(X)} Q_K(X))$.*

Proof: Let $\bar{R}$ be the set of tables of the form $\gamma(R)$, where γ is in the extended γ-condition of X and R is a table of a node in the description tree of X. Let $\bar{R}'$ be the set of all tables with underlying queries of the form $\gamma_1(R_1)$ `union all` … `union all` $\gamma_k(R_k)$, where $\{R_i\}_{i=1}^k$ are tables of nodes in the description tree of X and $\{\gamma_i\}_{i=1}^k$ are in the extended γ-condition of X.

$\Rightarrow$ We will first prove that $cl(\bigcup_{K \in cr(X)} Q_K(X)) \subseteq Q(X, \bar{R}')$. Indeed, let $Q \in cl(\bigcup_{K \in cr(X)} Q_K(X))$. Then there exists queries $\{Q_i\}_{i=1}^q$ s.t. $Q \equiv \mathtt{merge\_sort}(Q_1, \ldots, Q_q, O)$, $Q_i \in \bigcup_{K \in cr(X)} Q_K(X)$ for $i = 1$ to q, and Q_i is an eSQL query that does not contain non-parameterized restrictions, the result from Q_i is ordered according to O, and the query references only tables from the set $\bar{R}$. Next, note that $Q_i \in Q(X, \bar{R})$ for $i = 1$ to q implies that $Q \in Q(X, \bar{R}')$ because the plan for Q that returns the results of $\mathtt{merge\_sort}(Q_1, \ldots, Q_q, O)$ is a correct

(*type*)	(*query*)
(1)	`select` $D_1, \ldots, D_d$ `from` $\gamma(R_k)$ [`where` $A_1 = {:}P_1$ and $\ldots$ and $A_l = {:}P_l$] [`order by` $A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s$]
(2)	`select` $D_1, \ldots, D_d$ `from` $\gamma(R_k)$ `where` $A_1 =: P_1$ `and` $\ldots$ `and` $A_l = {:}P_l$ `and` A_{l+1} `between` ${:}P_{l+1}$ `and` ${:}P_{l+2}$ [`order by` $A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s$]
(3)	`select` $D_1, \ldots, D_d$ `from` $\gamma(R_k)$ `where ID` $= {:}P_1$

1. $[\cdot]$ denotes an optional component.
2. $\langle R_1, \ldots, R_k\rangle$ are the tables of $\langle n_1, \ldots, n_k\rangle$, respectively.
3. $\{A_i\}_{i=1}^{l}$ are non-`ID` attributes and A_{l+1}, if present, is a standard attribute.
4. $\{|D_i|\}_{i=1}^{d}$ are distinct attributes of R_k.
5. $\gamma \in \gamma^e(X)$.
6. $\langle A_1, \ldots, A_s\rangle$ are the first s attributes in this order from $\mathcal{L}^{\uparrow}(n_k)$.
7. Either K is clustered relative to l or there exists a $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t) =$ `TRUE` iff $t \in \gamma(R_k)$.

Table 2.9: Critical queries for a rooted path $K = \langle n_1, \ldots, n_k\rangle$ of X

and efficient plan if the query plans for $\{Q_i\}_{i=1}^{q}$ are efficient. We therefore need to show that if $Q \in Q_K(X)$ for some $K \in cr(X)$, then $Q \in Q(X, \bar{R})$.

Indeed, let $Q \in Q_K(X)$ for some rooted path K in the description tree of X. Suppose the nodes of K starting from the root node of the description tree of X are $\langle n_1, \ldots, n_k \rangle$. Let the attributes in $\mathcal{L}^{\uparrow}(n_k)$ be $\langle A_1, \ldots, A_b \rangle$ in that order.

First, consider a query of type 1 (see Table 2.9). This query can be efficiently answered by executing the following pseudo-code.

```
for (i = 1; i ≤ l; i++) dir_i = asc;
for (i = s + 1; i ≤ b; i++) dir_i = asc;
reference p = X.next(R_k, A_1, P_1, ..., A_l, P_l, dir_1, ..., dir_b, 0, γ);
while (p ≠ NULL){
    tuple t = X.retrieve(p);
    send construct_result(D_1 = t.D_1, ..., D_d = t.D_d);
    p = X.next(R_k, A_1, P_1, ..., A_l, P_l, dir_1, ..., dir_b, p, γ);
}
```

Note that the primitive `construct_result` is used to construct a resulting tuple out of the specified values for the attributes. Note as well that when Q is of the described type, condition 7 from Table 2.9 must hold and therefore the `next` method can be invoked on X.

The access plan first returns the first tuple t in the index that is in $\gamma(R_k)$ and for which $\bigwedge_{i=1}^{l} (t.A_i = P_i)$ holds. Then, it scans forward for the other tuples in the index for which the described condition holds. Theorem 3 guarantees the resulting tuples are returned in the desired time bound for efficient query plans. As well, the resulting tuples are returned in the order $\langle A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s, A_{s+1}$ `asc`, …, A_b `asc`$\rangle$, i.e. the access plan is correct.

Next, consider a query of type 2 (see Table 2.9). If the query does not contain an `order by` condition, then add `order by` A_{l+1} `asc` to the query. Note that this query modification will not change the semantics of the query. The query can be efficiently answered using the following pseudo-code.

```
(01) for (i = 1; i <= l; i++) dir_i = asc;
(02) for (i = s + 1; i <= b; i++) dir_i = asc;
(03) if (dir_{l+1} == asc) P = P_{l+1} else P = P_{l+2};
(04) reference p = X.next(R_k, A_1, P_1, ..., A_l, P_l, A_{l+1}, P, dir_1, ..., dir_b, 0, γ);
(05) if (p ≠ NULL) t = X.retrieve(p);
(06) if (p == NULL) {
(07)     reference p = X.search(R_k, A_1, P_1, ..., A_l, P_l, A_{l+1}, P, dir_1, ..., dir_b, γ, l);
(08)     if (p == NULL) return;
(09)     t = X.retrieve(p);
(10) }
(11) while ((t.A_{l+1} >= P_{l+1}) && (t.A_{l+1} <= P_{l+2})) {
(12)     send construct_result(D_1 = t.D_1, ..., D_d = t.D_d);
(13)     p = X.next(R_k, A_1, P_1, ..., A_l, P_l, dir_1, ..., dir_b, p, γ);
(14)     if (p == NULL) return;
(15)     t = X.retrieve(p);
(16) }
```

The query plans assumes that $P_{l+1} \leq P_{l+2}$. Note as well that when Q is of the described type, condition 7 from Table 2.9 must hold and therefore the `next` and `search` can be invoked.

When the scan on A_l is forward, i.e. $dir_{l+1} = \texttt{asc}$, the pseudo-code sets $P = P_{l+1}$ (i.e. the scan will be from $A_{l+1} = P_{l+1}$ to $A_{l+1} = P_{l+2}$). When the scan on A_l is backwards, i.e. $dir_{l+1} = \texttt{desc}$, the pseudo-code sets $P = P_{l+2}$ (i.e. the scan will be from $A_{l+1} = P_{l+2}$ to $A_{l+1} = P_{l+1}$) (see Line 3 of the pseudo-code). Line 5 of the code covers the case when there exists a tuple $t \in R_k$ in X that has the property that $\bigwedge_{i=1}^{l}(t.A_i = P_i)$, $t.A_{l+1} = P$, and $\gamma(t)$ holds. If such a tuple exists, then the first such tuple relative to the defined order will be the first tuple from the query result. If such a tuple does not exist (Lines 7-9), then the pseudo-code finds the first tuple $t \in R_r$, relative to the desired order, that passed the condition γ, for which $\bigwedge_{i=1}^{l}(t.A_i = P_i)$ holds, and for which $t.A_{l+1} > P$ when $dir_{l+1} = \texttt{asc}$ and $t.A_{l+1} < P$ when $dir_{l+1} = \texttt{desc}$. If such a tuple exists, then the first such tuple relative to the defined order will be the first tuple from the query result. If such a tuple does not exist, then the method returns because the query result will be empty.

Lines 11-16 of the code return tuples sequentially until the `next` method returns `NULL` (i.e. there are no more tuples with the desired properties) or until a tuple t is reached that has a value for the attribute A_{l+1} outside the range $[P_{l+1}, P_{l+2}]$. Theorem 3 guarantees the tuples are returned in the desired time bound for efficient query plans. As well, the resulting tuples are returned in the order $\langle A_{l+1}\ dir_{l+1}, \ldots, A_s\ dir_s, A_{s+1}, \texttt{asc}, \ldots, A_b\ \texttt{asc}\rangle$, i.e. the access plan is correct.

Finally, consider a query of type 3 (see Table 2.9). Such a query can be answered in constant time using the following access plan.

(1) $\texttt{tuple}\ t = X.\texttt{retrieve}(P_1)$;
(2) $\texttt{if}\ \gamma(t)\ \texttt{send}\ \text{construct\_result}(D_1 = t.D_1, \ldots, D_d = t.D_d)$;

It is easy to see that the query plan is correct and efficient because it returns the result in constant time.

$\Leftarrow$ We will next prove that $Q(X, \bar{R}') \subseteq cl(\bigcup_{K \in cr(X)} Q_K(X))$. Indeed, let $Q \in Q(X, \bar{R}')$.

First, suppose that Q reference more than one table (i.e. Q is an eSQL query of type 1 and $r > 1$ - see Table 2.3) Since there are foreign key dependencies between the tables referenced by Q and cyclic foreign key dependencies are not allowed, the tables referenced by Q must be distinct and they should have distinct values for the `ID` attribute (e.g. if Q references R_1 and R_2, then $\pi_{\texttt{ID}} R_1 \cap \pi_{\texttt{ID}} R_2 = \emptyset$). Let R_1 and R_2 be two tables referenced by Q that have a foreign key dependency defined between them (since Q is an eSQL query such two tables must exist). Let R be the table of the root node of the description tree of X. Then, since X is a valid extended index, it must be the case that $R_1 \subset_{\texttt{ID}} R$ and $R_2 \subset_{\texttt{ID}} R$. But this implies that there is a cyclic foreign key dependency between R and itself, which is a contradiction.

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

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

Indeed, assume that $\texttt{table}(n) = R_i$ for some n in the description tree of X. Note that Q_i is an efficient query iff $\gamma_i \in \gamma^e(X)$.

If Q_i is of the form:

> `select` $\ldots$
> `from` $\gamma_i(R_i)$
> `where ID` $= :P_1$,

then, by the definition of $Q_{K_i}(X)$ if follows that $Q \in Q_{K_i}(X)$. Next, suppose that Q_i is not of that form. Note that Q_i can not be of the following form.

> `select` $\ldots$
> `from` $\gamma_i(R_i)$
> `where` $A_1 = :P_1$ `and` $\ldots$ $A_l = P_l$ `and`
> C_1 `between` $:P_1$ `and` $:P_2$ `and` C_2 `between` $:P_3$ `and` $:P_4$

The reason is that X can not be used to answer such a query efficiently. As well, if Q_i contains a "C `between` $:P_1$ `and` $:P_2$" condition and an ordering condition, then the ordering condition has to be first relative to the attribute C. If not, then Q_i again will not be efficiently answerable using X.

Therefore, Q_i is of the following form.

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

The attributes $A_1, \ldots, A_s$ must be the first s attributes in $\mathcal{L}^{\uparrow}(n)$. Otherwise, again Q_i will not be efficiently answerable using X.

Recall that K_i is clustered relative to l iff the tuples in X with the same values for the attributes $\{A_i\}_{i=1}^{l}$ that are in R_i are consecutive relative to the order defined by X (see Lemma 2). If K_i is clustered relative to l, then condition 7 of Table 2.9 holds. If K_i is not clustered relative to l, then the tuples of R_i in X with the same value for the attributes $\{A_i\}_{i=1}^{l}$ will be scattered in X and therefore it must be the case that there exists $\gamma' \in \gamma^e(X)$ that has the property that $\gamma'(t)$ is true iff $t \in \gamma_i(R_i)$. Therefore $Q_i \in Q_{K_i}(X)$. ■

2.2.2 Compact Structures Design Model

The compact structures physical design model consists of *overlapping indices*, hash structures and linked lists. Unlike the previous extended index physical design model, this one is exogenous in the sense that the search structures are stored in overlapping indices and hash structures, while the actual records are stored in overlapping linked lists.

Hash Structures

The hash structure that we will consider is the traditional hash structure (e.g. extendible hashing [10], linear hashing [20], or dynamic hashing [19]) and has the following capabilities.

Definition 33 (syntax and semantics of a hash structure) *A hash structure H has the following general syntax $\langle R, \{A_1, \ldots, A_a\}\rangle$. This syntax represents a hash structure on the attributes $\{A_i\}_{i=1}^{a}$ of R.*

For a hash structure $\langle R, \{A_1, \ldots, A_a\}\rangle$ we only consider the case where $\{A_i\}_{i=1}^{a}$ form a candidate key for the table R. Such a hash structures take as input the values for the attributes $\{A_i\}_{i=1}^{a}$ and returns the `ID` of the tuple with the given values for the attributes $\{A_i\}_{i=1}^{a}$. The `ID` look-up structure can be used to find the address of the tuple with given `ID`. Whether we use the global or local look-up table to perform this operation depends on the structure in which the returning tuple is stored.

Definition 34 (interface of a hash structure) *A hash structure* $H = \langle R, \{A_1, \ldots, A_a\}\rangle$ *has the following capabilities:*

1. `reference insert(tuple` t`):`
   - **pre-conditions:** *t is compatible with R.*
   - **action:** *Inserts in H the tuple t.*
   - **return value:** *Returns the* `ID` *of the inserted tuple.*
   - **running time:** *The worst case running time for this operation is amortized constant time.*
2. `status delete(reference` p`):`
   - **pre-conditions:** *There exist in H an element with* `ID` *p.*
   - **action:** *Deletes from H the element with* `ID` *p.*
   - **return value:** *This procedure has no return value.*
   - **running time:** *The worst case running time for this operation is amortized constant time.*
3. `reference search(Attr` A_1`, Param` $P_1, \ldots,$ `Attr` A_a`, Param` P_a`):`
   - **pre-conditions:** *H contains a tuple t for which $t.A_i = P_i$ for $i \in [1..a]$.*
   - **action:** *No actions.*
   - **return value:** *Returns the* `ID` *of the tuple for which $t.A_i = {:}P_i$ for $i \in [1..a]$ if one exists. Returns* `NULL` *otherwise.*
   - **running time:** *The worst case running time for this operation is amortized constant time.*
4. `tuple retrieve(reference` p`):`
   - **pre-conditions:** *A tuple with* `ID` *p exists in H.*
   - **action:** *No actions.*

- **return value:** *Returns the tuple with* `ID` p *from* H.
- **running time:** *The worst case running time for this operation is constant.*

5. `getAll()`
   - **pre-conditions:** *No pre-conditions*
   - **action:** *No actions.*
   - **return value:** *Returns the* `ID`*s of the tuples in the hash structure.*
   - **running time:** *The worst case running time for this operation is order the size of the result.*

It usually takes less time to retrieve an element from a hash structure than from an index. However, note that a hash structure has less capabilities than an index, in particular a hash structure does not support a total order.

Linked Lists

We next explore two linked list types: *ordinary linked lists* and *doubly linked lists*. Note that both linked list types allow an element to be inserted immediately after a given element in constant time. However, only a doubly linked list allows for a given element to be deleted in constant time. A description of the syntax and semantics of a linked list follows.

Definition 35 (syntax and semantics of a linked list) *A linked list W has the following syntax $\langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle, type\rangle$, where type $\in \{\texttt{ll}, \texttt{dll}\}$, $\{A_i\}_{i=1}^{a} \in Attr(R)$, and $\langle A_1,\ dir_1, \ldots, A_a\ dir_a\rangle$ is a total order. If type=*`ll`*, then its an ordinary linked list and if type=*`dll`*, then its a doubly linked list. The list contains the elements of the table R in the order $\langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle$.*

The interface of a linked list follows.

Definition 36 (interface of a linked list) *A linked list $W = \langle R, \langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle, type\rangle$ has the following capabilities:*

1. `reference insert`*(tuple t, reference p):*

- **pre-conditions:** *t is compatible with R and either $p = 0$ or there exists a record in W with* `ID` *p*
- **action:** *Inserts in W the tuple t. If $p = 0$, then the tuple is inserted at the beginning of the list. If $p \neq 0$, then the tuple is inserted immediately after the tuple with* `ID` *p.*
- **return value:** *Returns the* `ID` *of the inserted tuple.*
- **running time:** *The worst case running time for this operation is constant.*

2. `status delete(reference` p`)`*:*

- **pre-conditions:** *type* $=$ `dll`*, there exist in W an element with* `ID` *p.*
- **action:** *Deletes from W the element with* `ID` *p.*
- **return value:** *This method has no return value.*
- **running time:** *The worst case running time for this method is constant.*

3. `tuple retrieve(reference` p`)`*:*

- **pre-conditions:** *A tuple with* `ID` *p exists in W.*
- **action:** *No actions.*
- **return value:** *Returns the tuple with* `ID` *p from W.*
- **running time:** *The worst case running time for this operation is constant.*

4. `reference next(reference` p`)`*:*

- **pre-conditions:** *Either $p = 0$ or W contains a tuple t with* `ID` *p.*
- **action:** *No actions.*
- **return value:** *If $p = 0$, then the method returns the* `ID` *of the first tuple of the linked list. If $p > 0$, then the method returns the* `ID` *of the tuple in the linked list that is after the tuple with* `ID` *p. If such a tuple does not exists, then the method returns* `NULL`*.*
- **running time:** *The worst case running time for this operation is constant.*

5. `reference previous(reference` p`)`*:*

   - **pre-conditions:** *type* $=$ `dll` *and either* $p = 0$ *or* W *contains a tuple* t *with* `ID` p.
   - **action:** *No actions.*
   - **return value:** *If* $p = 0$*, then the method returns the* `ID` *of the last tuple in the linked list. If* $p > 0$*, then the method returns the* `ID` *of the tuple immediately before the tuple with* `ID` p *in the linked list. If such a tuple does not exists, then the method returns* `NULL`*.*
   - **running time:** *The worst case running time for this method is constant.*

First, note that in order to be able to execute `previous`(0) efficiently, we need to store the reference of the last element in the index. Second, note that several linked lists can be stored on top of each other. This can be achieved by storing the `ID`s of the records of the linked lists in the global look-up table.

Compact Structures

A compact structure consists of a number of extended indices, hash structures, and overlapping linked and doubly linked lists. A formal definition of a compact structure follows.

Definition 37 (complete path in a tree) *Let* G^t *be a tree. A complete path in* G^t *is a path that starts from 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 38 (syntax of a compact structure) *The description tree of a compact structure is unordered node labeled tree, which we will refer to as* G^t*. The general syntax of the label of a non-leaf node of* G^t *is either* $\langle \bar{\gamma}, R, \langle A_1, \ldots, A_a \rangle \rangle$ *or* $\langle R, \langle \{A_1, \ldots, A_a\} \rangle \rangle$*, where* R *is a table,* $\{A_i\}_{i=1}^{a} \subseteq Attr(R)$*,* $\bar{\gamma}$ *is a set of efficient predicates,* $a \geq 0$*, and* $b > 0$*. We will refer to nodes of the first type as index nodes and to nodes of the second type as hash nodes. Note that when* $\bar{\gamma}$ *contains only the predicate* `TRUE`*, we will use the syntax* $\langle R, \langle A_1, \ldots, A_a \rangle \rangle$ *to represent an index node. Given a non-leaf node* n*, we will refer to* R *as the node's table* *and write* $\texttt{table}(n)$*, to* $\bar{\gamma}$ *as the node's* γ*-condition and write* $\gamma(n)$*, and to* $\langle A_1 \ldots, A_a \rangle$ *and* $\{A_1, \ldots, A_a\}$ *as the node's ordering label and write* $\mathcal{L}(n)$*.*

The general syntax of a leaf node is $\langle R, \{C_1, \ldots, C_c\}, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle, type \rangle$*, where* $\{A_i\}_{i=1}^{a} \cup \{C_i\}_{i=1}^{c} \subseteq Attr(R)$ *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 the type part 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 structure to be valid, we impose the following restrictions.*

1. *If the node* n' *with table* R' *is the parent of the node* n *with label* R *in* G^t*, then* $R \subseteq R'$*.*

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

3. *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 table* R *s.t.* $\gamma(R') = R$*. Moreover, for any two such node* n *and* n'*, all the intermediate nodes along the path from* n' *to* n*, if any, are either based on the table* R *or contain the efficient predicate* γ *in their extended* γ*-condition.*

4. *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.*

Definition 39 (semantics of a compact structure) *We will define the semantics for a set of compact structures because two or more such structure may share parts of their physical designs. Each node in the description tree of a compact structure corresponds to a physical design. Let* J *be a compact structure and let* G^t *be the corresponding description tree.*

Let n *be a leaf node of* G^t *with label* $\langle R, \{C_1, \ldots, C_c\}, \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle, type \rangle$*. First, consider the case where* n *is the root node, i.e.* G^t *consists of a single node. If* n*'s ordering label is empty and* $type(n) = \texttt{ll}$ *(*$type(n) = \texttt{dll}$*), then* n *will correspond to a linked list* *(doubly linked list) of the records* $\pi^d_{C_1,\ldots,C_c}(R)$ *in some order, where* π^d *denotes duplicate preserving projection. If* n*'s ordering label is not empty and* $type(n) = \texttt{ll}$ *(*$type(n) = \texttt{dll}$*), then* n *will correspond a linked list (doubly linked list) of the records* $\pi^d_{C_1,\ldots,C_c}(R)$ *that is in the order* $\langle A_1\ dir_1, \ldots, A_a\ dir_a\rangle$*.*

Second, consider the case where n *is not the root node of* G^t*, but* n *is still a leaf node of* G^t *and let* n' *be the parent of* n *in* G^t*. Suppose that* $\{D_i\}_{i=1}^d$ *are the attributes in* $\mathcal{L}^{\uparrow}(n')$*. Then* n *will correspond to a set of linked lists if* $type = \texttt{ll}$ *and to a set of doubly linked lists if* $type = \texttt{dll}$*. 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* $\pi^d_{C_1,\ldots,C_c}\sigma_{D_1=c_1\wedge\ldots\wedge D_d=c_d}(R)$ *ordered by the ordering condition in the ordering label of* n *when present.*

Note that the leaf nodes in a set of compact structures correspond to a set of linked lists. We will allow those lists to overlap, i.e. when these linked lists refer to the same tuple more than once, we will store only one instance on the tuple. This implies that different records can have different number of next/previous references and the offset for an attribute may change during a linked list traversal. This is why we also store information on where each attribute of each record is located. This can be achieved, for example, by using the data structure proposed by Zibin and Gil in [42]. Note as well, that if a record has several pointers with the same value, then the value of the pointer could be stored only once and the look-up structure will return the same address when asked to return the value of the different pointers.

Next, consider the case where n *is a non-leaf node and the physical design for its descendent nodes has already been defined. Let* $\{n_i\}_{i=1}^u$ *be the child nodes of* n *in* G^t *and let* R *be the table for* n*.*

First, suppose that n *is the root of* G^t*. Let* $D_1, \ldots, D_d$ *be the attributes of the ordering label of* n*. We define* R' *to be the table* $\pi_{D_1,\ldots,D_d}(R)$ *with* u *derived attribute, where the* i^{th} *derived attribute,* $i = 1$ *to* u*, is the* `ID` *of the physical design of* n_i *for the corresponding values of* $D_1, \ldots, D_d$*. If* n *is of the form* $\langle\{\gamma_1, \ldots, \gamma_m\}, R, \langle A_1, \ldots, A_a\rangle\rangle$*, then let* $\{\gamma'_i\}_{i=1}^m$ *be new predicates that have the property that* $\gamma'_i(t')$ *holds for a tuple* $t' \in R'$*,* $i = 1$ *to* m*, iff* R *contains a tuple* t *such that* $\gamma_i(t)$ *holds and* $t.A_j = t'.A_j$ *for* $j = 1$ *to* a*. In this case* n *will correspond to the extended index* $\langle\{\gamma'_1, \ldots, \gamma'_m\}, R', \langle A_1, \ldots, A_a\rangle\rangle$*. On the other hand,* *if n is still the root node of G^t, but its label is of the form $\langle R, \{A_1, \ldots, A_a\}\rangle$, then n will correspond to the hash structure $\langle R', \{A_1, \ldots, A_a\}\rangle$.*

Finally, consider the case where n is a non-root and a non-leaf node and let $D_1, \ldots, D_d$ be the attributes of the ordering label of n. 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 correspond to a number of a structures, one for each distinct value of the attributes $\{M_i\}_{i=1}^m$ in R. For $\{M_i = v_i\}_{i=1}^m$ we define the table $R'_{v_1,\ldots,v_m}$ as $\pi_{D_1,\ldots,D_d}(\sigma_{M_1=v_1\wedge\ldots\wedge M_m=v_m} R)$ with u derived attributes, where the i^{th} derived attribute, $i = 1$ to u, is the ID *of the physical design of n_i for the corresponding values of $M_1, \ldots, M_m$ and $D_1, \ldots, D_d$. If n is of the form $\langle\{\gamma_1, \ldots, \gamma_m\}, R, \langle A_1, \ldots, A_a\rangle\rangle$, then let $\{\gamma'_{i,v_1,\ldots,v_m}\}_{i=1}^m$ be new predicates that have the property that $\gamma'_{i,v_1,\ldots,v_m}(t')$ holds for a tuple $t' \in R'_{v_1,\ldots,v_m}$, $i = 1$ to m, only when R contains a tuple t such that $\gamma_i(t)$, $t.M_j = t'.M_j$ for $j = 1$ to m, and $t.A_j = t'.A_j$ for $j \in [1\ldots a]$ all hold. In this case the physical design of n for $M_i = v_i$, $i = 1$ to m, will correspond to the extended index $\langle\{\gamma'_{1,v_1,\ldots,v_m}, \ldots, \gamma'_{m,v_1,\ldots,v_m}\}, R'_{v_1,\ldots,v_m}, \langle A_1, \ldots, A_a\rangle\rangle$. On the other hand, if n's label is of the form $\langle R, \{A_1, \ldots, A_a\}\rangle$, then the physical design of n for $M_i = v_i$, $i = 1$ to m, will correspond to the hash structure $\langle R'_{v_1,\ldots,v_m}, \{A_1, \ldots, A_b\}\rangle$.*

Next, we give an example of a compact structure. Consider the Tables 1.6 and 1.7 from the motivating example in Section 1.2. The physical design shown in the tables corresponds to the compact structure: $\langle\{\gamma\}, \texttt{COMP\_REMP\_FSTATS}, \langle loc\rangle, [\langle \texttt{COMP\_YREMP\_FSTATS}, \langle rank\rangle, [\langle \texttt{COMP\_YREMP\_FSTATS}, \{*\}, \langle\rangle, \texttt{ll}\rangle]\rangle, \langle \texttt{COMP\_REMP\_FSTATS}, \langle budget\rangle, [\langle \texttt{COMP\_REMP\_FSTATS}, \{*\}, \langle\rangle, \texttt{ll}\rangle]\rangle]\rangle$, where γ is such that $\gamma(t)$ is **TRUE** iff $t \in$ COMP_YREMP_FSTATS.

Before describing the set of critical queries that we will use a compact structure to answer efficiently, we present the following intermediate definitions.

Definition 40 (permutation of a complete path) *Let K be a complete path in the description tree of the compact structure J and let $\langle n_1, \ldots, n_k\rangle$ be the nodes of K. Then a permutation Π converts K into a list of attributes. This list contains the attributes in the ordering label of $n_1, \ldots, n_{k-1}$ in that order followed by the attributes of the ordering condition of n_k, where the order of the attributes is preserved for index and leaf nodes, but not for hash nodes (i.e. the attributes in the ordering label of a hash node can be permutated).*

Definition 41 (queries of a complete path of a compact structure) *Let J be a compact structure 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 $k \geq 0$. Then Table 2.10 shows the set of queries that we will associate with the path K. We will refer to this set as $Q_K(J)$.*

Definition 42 (queries of a compact structure) *Let J be a compact structure over the database schema Σ and let $\bar{R}$ be the set of tables of the leaf nodes of the description tree of J. Then we define $Q(J)$ as $Q(J, \bar{R})$. We will refer to $Q(J)$ as the queries of the compact structure J.*

Theorem 5 (efficiently supported queries by a compact structure) *Let J be a compact structure over the database schema Σ. Then $\bigcup_{K \in \mathtt{cp}(J)} Q_K(J) = Q(J)$.*

Proof: Let $\bar{R}$ be the set of tables referenced in the leaf nodes of the description tree for J.

$\Rightarrow$ We will first prove that $\bigcup_{K \in cp(J)} Q_K(J) \subseteq Q(J, \bar{R})$. Indeed, let $Q \in Q_K(J)$ for some complete path K in J. Suppose that the nodes along K starting from the root of the description tree of J are $\langle n_1, \ldots, n_{k+1} \rangle$ in this order. Let the table of n_i be R_i for $i \in [1..k]$.

Consider first the case when Q a query of type 1 (see Table 2.10). Let r be the subscript of the node with the biggest subscript in K that has an ordering label and contains exclusively attributes from the set $\{A_i\}_{i=1}^{l}$ in the ordering label. We set $r = 0$ when such a node does not exists. Let $A_1, \ldots, A_{m_1}$ be the attributes in n_1 and $A_{m_{i-1}+1}, \ldots, A_{m_i}$ be the attributes of n_i for $i = 2$ to k. For consistency, we define $m_0 = 0$ and $dir_j =$ `asc` for $j \in [1 \ldots l] \cup [a+1 \ldots m_k]$. Our claim is that Q can be efficiently answered by running the following pseudo-code with parameters $i = 1$ and $\mathcal{D}$ equal to the physical design for the node n_1.

```
follow(int i, physical design D){
    reference p;
    tuple t;
    physical structure D_1;
```

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

1. R_{k+1} is the table of n_{k+1};
2. $\{D_i\}_{i=1}^{d}$ are distinct stored attributes of n_{k+1} and $\{|A_i|\}_{i=1}^{a}$ are distinct attributes;
3. $\{A_i\}_{i=1}^{l}$ are non-`ID` attributes and $\{A_i\}_{i=l+1}^{a}$ are non-reference attributes.
4. If `type`$(n_{k+1} =$ `dll`$)$, then $w =$ `TRUE` or `FALSE`, else $w =$ `TRUE`.
5. There exists a permutation Π of $\langle n_1, \ldots, n_{k+1}\rangle$ s.t. $A_1, \ldots, A_a$ is a prefix of $\Pi(\langle n_1, \ldots, n_{k+1}\rangle)$ and if A_i appears in the ordering label of the node n_{k+1}, then it appears there with direction dir_i;
6. Either K is clustered relative to l of 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 its table is R_{k+1} or that its an index node with table R and extended γ-condition that contains the predicate γ that has the property that $\gamma(R) = R_{k+1}$; and
7. If A_l does not belong to the ordering condition of n_{k+1} and r is such that A_l is an attribute in the search condition of the node n_r, then either n_r is an index node or it is a hash node and the attributes in the ordering label of n_r belong to the set $\{A_i\}_{i=1}^{l}$.

Table 2.10: Critical queries for a complete path $K = \langle n_1, \ldots, n_{k+1}\rangle$

`if` $(i == k+1)\{$
  `if` $(w ==$ `TRUE`$)$ $p = \mathcal{D}$.`next`(0) `else` $p = \mathcal{D}$.`previous`(0);
  `while` $(p \neq$ `NULL`$)\{$
    $t = \mathcal{D}$.`retrieve`(p);
    `send construct_result`$(D_1 = t.D_1, \ldots, D_d = t.D_d)$;
    `if` $(w ==$ `TRUE`$)$ $p = \mathcal{D}$.`next`(p) `else` $p = \mathcal{D}$.`previous`(p);
  $\}$
$\}$
`else` $\{$
  `if` $(i \leq r)\{$
    `if` $($`type`$(\mathcal{D}) ==$ `hash`$)$
      $p = \mathcal{D}$.`search`$(A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_{m_i}, P_{m_i})$
    `else`
      $p = \mathcal{D}$.`next`$($`table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_{m_i}, P_{m_i}, dir_{m_{i-1}+1}, \ldots, dir_{m_i}, 0,$ `TRUE`$)$;
    `if` $(p \neq$ `NULL`$)$ $t = \mathcal{D}$.`retrieve`(p) `else` halt;
    $\mathcal{D} =$ `find_structure`(t, n_{i+1});
    `follow`$(i+1, \mathcal{D})$;
  $\}$
  `else` $\{$
    `if` $(i == r+1$ `&&` $non\_empty)\{$
      $p = \mathcal{D}$.`next`$($`table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_l, P_l, dir_{m_{i-1}+1}, \ldots, dir_l, dir^w_{l+1}, \ldots, dir^w_{m_i}, 0, \gamma_i)$
      `if` $(p \neq$ `NULL`$)$ $t = \mathcal{D}$.`retrieve`(p) `else` halt;
      `while` $(p \neq$ `NULL`$)$
        $\mathcal{D}_1 =$ `find_structure`(t, n_{i+1});
        `follow_structure`$(i+1, \mathcal{D}_1)$;
        $p = \mathcal{D}$.`next`$($`table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_l, P_l, dir_{m_{i-1}+1}, \ldots, dir_l, dir^w_{l+1}, \ldots, dir^w_{m_i}, p, \gamma_i)$
        $t = \mathcal{D}$.`retrieve`(p);
      $\}$

```
32            }
33            else {
34              if (type(𝒟) = hash)
35                for p ∈ 𝒟.getAll() {
36                  t = 𝒟.retrieve(p);
37                  𝒟_1 = find_structure(t, n_{i+1});
38                  follow(i + 1, D_1);
39                }
40              else {
41                p = 𝒟.next(table(root(𝒟)), dir^w_{m_{i-1}+1}, ..., dir^w_{m_i}, 0, γ_i);
42                while(p ≠ NULL){
43                  t = 𝒟.retrieve(p);
44                  𝒟_1 = find_structure(t, n_{i+1});
45                  follow(i + 1, D_1);
46                  p = 𝒟.next(table(root(𝒟)), dir^w_{m_{i-1}+1}, ..., dir^w_{m_i}, p, γ_i);
47                }
48              }
49            }
50          }
51        }
```

The presented procedure traverses the physical designs $\mathcal{D}$ that is of the physical designs for n_i and recursively calls itself to traverse the required physical designs for n_{i+1}.

Lines 4–11 of the code cover the base case. In those lines the tuples in a linked list are scanned an their attributes returned using the primitive `send`.

Lines 13–21 cover the case when $i \leq r$. In this scenario only a single look-up needs to be done in the corresponding index or hash structure. The method `find_structure`(t, n) is used to denote the function that finds the physical structure for the node n that has an `ID` valued stored in the corresponding attribute of t.

Lines 23–31 cover the case when partial match conditions are specified on a subset of the search attributes of n_i. We define the variable *non_empty* to be true only when the ordering label of n_{r+1} contains at least one attribute from the set $\{A_i\}_{i=1}^{l}$. The pseudo-code scans in for tuples that have the requested values for the search attributes in n_{r+1} and that pass the predicate γ_{r+1}. We define $\gamma_i(t)$ to be true for a tuple t, $i =\in [r+1, \ldots k]$, only when there exists a tuple $t' \in R_{k+1}$ such that t' and t have the same value for the search attributes of n_i.

Finally, Lines 34-48 of the pseudo-code cover the case where all the values of the physical design $\mathcal{D}$ that pass the condition γ_i need to be retrieved. Note that when $\mathcal{D}$ is a hash structure then, by the definition of a compact structure, we are guaranteed that the tables of n_i and n_{k+1} are the same and therefore no γ condition needs to be specified.

First, note that the presented access plan returns the desired values. It first consecutively probes the necessary data structures for the nodes $n_1, \ldots, n_r$ with the given values for the parameters. Then it scans the required physical designs for the nodes $n_{r+1}, \ldots, n_{k+1}$. It is also efficient because only values that contribute to the query result are returned from each scan of a search and hash structure. Finally, note that the order in which the tuples are returned coincides with the order specified by the query.

Next, consider the case when Q is of type 2 (see Table 2.10). If the query does not contain an `order by` condition, then we will add to the query the expression: `order by` A_{l+1} `asc`. We can do this because any answer to the new query will also be an answer to the old one. The query can be efficiently answered by executing the following pseudo-code with parameters $i = 1$ and $\mathcal{D}$ equal to the physical design for the node n_1.

```
follow(int i, physical design D){
  reference p;
  tuple t;
  physical structure D1;
  if (i == k + 1){
    if (w == TRUE) p = D.next(0) else p = D.previous(0);
    while (p ≠ NULL){
      t = D.retrieve(p);
      send construct_result(D1 = t.D1, ..., Dd = t.Dd);
      if (w == TRUE) p = D.next(p) else p = D.previous(p);
    }
```

}
`else` {
`if` $(i \leq r)\{$
`if` $($`type`$(\mathcal{D}) ==$ `hash`$)$
$p = \mathcal{D}.$`search`$(A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_{m_i}, P_{m_i})$
`else`
$p = \mathcal{D}.$`next`$($`table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_{m_i}, P_{m_i},$ $dir_{m_{i-1}+1}, \ldots, dir_{m_i}, 0,$ `TRUE`$);$
`if` $(p \neq$ `NULL`$)$ $t = \mathcal{D}.$`retrieve`(p) `else` halt;
$\mathcal{D} =$ `find_structure`$(t, n_{i+1});$
`follow`$(i+1, \mathcal{D});$
}
`else` {
`if` $(i == r+1)\{$
`if` $dir^w_{l+1} =$ `asc` $P = P_{l+1}$ else $P = P_{l+2};$
$p = \mathcal{D}.$`next`$($`Table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_l, P_l, A_{l+1}, P$ $dir_{m_{i-1}+1}, \ldots, dir_l, dir^w_{l+1}, \ldots, dir^w_{m_i}, 0, \gamma_i);$
`if` $(p \neq$ `NULL`$)$ $t = \mathcal{D}.$`retrieve`$(p);$
`if` $(p ==$ `NULL`$)$
`reference` $p = \mathcal{D}.$`search`$($`Table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots,$ $A_l, P_l, A_{l+1}, P, dir_{m_{i-1}+1}, \ldots, dir_l, dir^w_{l+1}, \ldots, dir^w_{m_i}, \gamma_i, l - m_{i-1} - 1);$
`if` $(p ==$ `NULL`$)$ `return`;
$t = \mathcal{D}.$`retrieve`$(p);$
}
`while` $((t.A_{l+1} \geq P_{l+1})$ `&&` $(t.A_{l+1} \leq P_{l+2}))$ {
$\mathcal{D}_1 =$ `find_structure`$(t, n_{i+1});$
`follow_structure`$(i+1, \mathcal{D}_1);$
$p = \mathcal{D}.$`next`$($`Table`$($`root`$(\mathcal{D})), A_{m_{i-1}+1}, P_{m_{i-1}+1}, \ldots, A_l, P_l,$ $dir_{m_{i-1}+1}, \ldots, dir_l, dir^w_{l+1}, \ldots, dir^w_{m_i}, p, \gamma_i);$
`if` $(p \neq$ `NULL`$)$ return;
$t = \mathcal{D}.$`retrieve`$(p);$

```
38            }
39         }
40         else {
41            if (type(D) = hash)
42               for p ∈ D.getAll() {
43                  t = D.retrieve(p);
44                  D_1 = find_structure(t, n_{i+1});
45                  follow(i + 1, D_1);
46               }
47            else {
48               p = D.next(table(root(D)), dir^w_{m_{i-1}+1}, ..., dir^w_{m_i}, 0, γ_i);
49               while(p ≠ NULL){
50                  t = D.retrieve(p);
51                  D_1 = find_structure(t, n_{i+1});
52                  follow(i + 1, D_1);
53                  p = D.next(table(root(D)), dir^w_{m_{i-1}+1}, ..., dir^w_{m_i}, p, γ_i);
54               }
55            }
56         }
57      }
58   }
```

The code is similar to the code for queries of type 1, where the only difference is the pseudo-code when $i = r + 1$ (Lines 26-39). Note that n_{r+1} is the node that contains in its ordering label zero or more attributes partial match attributes of the query Q and it also contains the attribute A_{l+1} on which the range predicate is defined. In the code a scan of each physical structure for n_{r+1} is done. The scan first tries to find a tuple t in the structure for which $t.A_i = P_i$ for $i \in [m_{r-1} + 1, \ldots, l]$ and for which $t.M_{l+1} = P$, where P is defined in Line 24 of the code. If such a tuple exists, then this tuple will be the first selected tuple from the current node. If it does not, then the pseudo-code tries to find the first tuple t for which $t.A_i = P_i$ for $i \in [m_{r-1} + 1, \ldots, l]$ and $t.A_{l+1}$ is in the defined range.

Then the scan continues until a tuple t for which $t.A_{l+1}$ is out of the range $[P_{l+1} \ldots P_{l+2}]$ is reached. The reasoning why the pseudo-code returns the correct tuples in the right order and in the desired time-bound is similar to the reasoning for queries of type 1.

Finally, consider a query of type 3 (see Table 2.10). Such a query can be efficiently answered using the following query plan.

(01) $\texttt{tuple}\ t = \texttt{retrieve}(P_1)$;
(02) $\texttt{send construct\_result}(D_1 = t.D_1, \ldots, D_d = t.D_d)$;

Note that line uses the global `ID` look-up table to find the `ID` of the record where the tuple with `ID` P_1 is stored. It is easy to see that the query plan is correct and efficient because it returns the result in constant time.

$\Leftarrow$ We will next prove that $Q(J, \bar{R}) \subseteq \bigcup_{K \in cr(J)} Q_K(J)$, i.e. that for every eSQL Q that references only the tables from the leaf nodes of the description tree of J and that can be efficiently answered using J, there exists a complete path K in J such that $Q \in Q_K(J)$.

First, suppose that Q reference more than one table. Since there are foreign key dependencies between the tables referenced by Q and cyclic foreign key dependencies are not allowed, the tables referenced by Q must be distinct. Let R_1 and R_2 be two tables referenced by Q that have a foreign key dependency defined between them (since Q is an eSQL query such two tables must exist). Let $\langle R, \ldots \rangle$ be the label of the root node of J. Then, since J is a valid compact structure, it must be the case that $R_1 \subseteq R$ and $R_2 \subseteq R$. But this implies that there is a cyclic foreign key dependency between R and itself, which is a contradiction.

Therefore Q must reference table that belongs to a leaf node of the description tree of J. Let K be the complete path in the description tree of J that ends at the leaf node that references the table over which Q is defined. Let $\langle n_1, \ldots, n_{k+1} \rangle$ be the nodes along the path K and let $\langle R_1, \ldots, R_{k+1} \rangle$ be their respective tables.

If Q is of the form:

$\texttt{select}\ D_1, \ldots, D_d$
$\texttt{from}\ R_{k+1}$
$\texttt{where ID} = {:}P_1,$

then, by the definition of $Q_K(J)$ if follows that $Q \in Q_K(J)$. Next, suppose that Q is not of the form. Note that Q can not be of the following form.

select ID
from R_{k+1}
where $M_1 =$:P_1 and ... $M_l = P_l$ and
C_1 between :P_{l+1} and :P_{l+2} and C_2 between :P_{l+3} and :P_{l+4}

The reason is that J can not be used to answer such a query efficiently. As well, if Q contains a "C between :P_1 and :P_2" condition and an ordering condition, then the order has to be first relative to the attribute C. If not, then Q again will not be efficiently answerable using X.

Therefore, Q is of the following form.

select $D_1, \ldots, D_d$
from R_{k+1}
where $A_1 =$:P_1 and ... $A_l = P_l$ and A_{l+1} between :P_{l+1} and :P_{l+2}
order by A_{l+1} $dir_{l+1}, \ldots, A_s$ dir_s

In order for Q to be efficiently answerable the attributes $D_1 \ldots, D_d$ and $A_1, \ldots, A_s$ must have the properties described in Table 2.10 and therefore $Q \in Q_K(X)$. ■

We will next show how a compact structure can be updated.

Definition 43 (update interface of a compact structure) *Let J be a compact structure over the database schema Σ and let R be the table of the root node of the description tree of J. Then J supports the following update primitives.*

1. `reference` **insert**(`tuple` t)*:*
   - **pre-conditions:** *t is compatible with R*
   - **action:** *The tuple t is inserted in J.*
   - **return value:** *This method returns the* `ID` *of the inserted tuple.*
2. **delete**(`reference` p)*:*

- **pre-conditions:** *There exists a tuple with* `ID` p *in* J.
- **action:** *Deletes the record with* `ID` p *from* R.
- **return value:** *This method has no return value.*

Theorem 6 (running time for the update methods of a compact structure) *Let J is a compact structure and let the branching factor of the description tree of J and the branching factor of all other defined compact structures be constant. Suppose as well that the number of compact structures is constant and the size of each record in each compact structure is constant. Then the following is true.*

1. *The* `insert(tuple` t`)` *method can be performed in $O(log(|J|) \cdot |def(J)|)$ time as long as every leaf node of the description tree of J that is based on a table R such that $t \in R$ contains an empty ordering condition.*

2. *The* `delete(reference` p`)` *method can be performed in $O(log(|J|) \cdot |def(J)|)$ time as long as every leaf node of the description tree of J that is based on a table R such that R contains a tuple with* `ID` p *is of type doubly linked list.*

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

```
reference result;
insert(tuple t, physical design D) {
   node n = node(D);
   if t ∉ table(D) exit;
   if (type(D) == list) {
      if (D.next(0) ≠ t.ID) result = D.insert(t, 0);
   }
   else {
      node {n_i}_{i=1}^k = children(n);
      reference p = D.search(t.Attr(n));
      if (p == NULL) {
```

```
10          for(i = 1; i ≤ k; i++) D_i = new(n_i);
11          D.insert(construct_tuple(t.Attr(n), D_1, ..., D_k));
12        }
13        else {
14          for(i = 1; i ≤ k; i++) D_i = find_structure(t, n_i);
15        }
16        for(i = 1; i ≤ k; i++) insert(t, D_i);
17     }
18 }
```

Line 1 of the code sets n to be the node in the description tree of J of which $\mathcal{D}$ is an instance. Line 2 checks whether n belongs to the table for the current node. If this is not the case, then t should not be inserted into $\mathcal{D}$ and the procedure exists. Lines 3-5 cover the case when $\mathcal{D}$ is a list. The tuple t is inserted in the begging of the list if this has not been already done. Note that this is a correct behavior for this case because we know that no order is defined on the list. Note as well that we need the check whether the tuple has not already been inserted because the `insert` method can try to insert the same tuple more than once because the physical designs for the leaf node of the description tree of a compact structure can overlap. We have used the variable `result` to stored the result of executing the procedure.

Lines 7-16 cover the case when n is not a leaf-node in the description tree of J. Line 7 initializes $\{n_i\}_{i=1}^{k}$ to be the children of n in the description tree for J. In Line 8 we use the new primitive *search* to refer to a search for an element in both an extended index and a hash structure, where the result of executing the primitive is `NULL` when an element is not found. The `search` method specified all directions as `asc` when searching in a extended index. When $p =$ `NULL`, we use the primitive `new` to create new physical designs for $\mathcal{D}_i$ as an instance of the node n_i, $i = 1$ to k. The new tuple that we insert in $\mathcal{D}$ points to these newly created designs. Note that we use the $t.\texttt{Attr}(n)$ primitive to denote the attributes of t referenced in the ordering label of the node n. As well, we use the `construct_tuple` primite to denote the operation of constructing a tuple from the given values for the attributes. Line 14 covers the case when a node for t already exists in $\mathcal{D}$.

In this case $\{\mathcal{D}_i\}_{i=1}^k$ are set to the physical designs pointed to be the found tuple. Finally, line 16 recursively calls the `insert` procedure with the physical designs for $\{\mathcal{D}_i\}_{i=1}^k$. Note that the presented pseud-code exhibits the desired behavior.

Next, note that the insertion pseudo-code visits $O(|def(J)|)$ complete paths in J and it takes $O(log(|J|))$ time to visit each path, i.e. the pseudo-code runs in the required time-bound.

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

```
reference p;
tuple t;
int delete(physical design D) {
   node n = node(D);
   if (t ∉ table(D)) return !empty(D);
   if (type(D) == list) {
      if (D.retrieve(p) ≠ NULL) D.delete(p);
      return !empty(D);
   }
   else {
      node {n_i}_{i=1}^k = children(n);
      Boolean deletion_required = FALSE;
      for(i = 1; i ≤ k; i++) {
         D_i = find_structure(t, n_i);
         if (delete(D_i) == TRUE) deletion_required = FALSE;
      }
      if (deletion_required == TRUE) D.delete(search(t.Attr(n)));
      return !empty(D);
   }
}
```

Line 2 of the code checks whether the tuple t actually belongs to the table for the physical design $\mathcal{D}$. If this is not the case, then no deletion needs to be performed and !`empty`($\mathcal{D}$) is returned. Note that the we require from the `delete` pseudo-code to return `FALSE` when $\mathcal{D}$ is empty after the necessary operations on $\mathcal{D}$ are performed and return `TRUE` otherwise. Lines 4-5 cover the case when $\mathcal{D}$ is a linked list. It checks that t has not already been deleted, and if it is not, the procedure deletes t from the current linked list and all linked lists it appears in. Since all such linked lists are doubly linked list, the deletion can be performed in the constant time. Line 8-15 cover the case when $\mathcal{D}$ is not a linked list. First, all the physical designs for the descendants of n that contain the values for the corresponding attributes of t are found. The deletion is performed recursively on them. The variable *deletion_required* is initially set to `TRUE`. However, if any of the visited descendent physical designs is not empty, then the *deletion_required* variable is set to `FALSE`, because in this case the tuple t for the current physical design will not have to be deleted.

Next, note that the deletion pseudo-code visits $O(|def(J)|)$ complete paths in J and it takes $O(log(|J|))$ time to visit each path, i.e. the pseudo-code runs in the required time-bound. ■

Chapter 3

Efficiently Maintaining MVs and Derived Attributes

3.1 Maintaining Derived Attributes

Recall that if t is a tuple of the base table R_r and A is a derived attribute of t, then the value for $t.A$ is either a function of the non-derived attributes of t or a function of the whole database.

In the first case, $t.A = f(t.A_1, \ldots, t.A_a)$, where $A_1, \ldots, A_a$ are standard non-derived attributes of R_r and f is a function that works in $O(|input|)$ time. In this case no special actions need to performed for adding or deleting a tuple. When the value of a tuple's attribute is modified, then the affected derived attributes should be recalculated, which takes $O(|\Sigma|)$ time.

In the second case, if $t \in R_r$, then $t.A$ is defined as the result of a query Q that is of one of the three types shown in Table 2.6.

We will first present an algorithm for efficiently maintaining the derived attribute C_r. 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` $count(*)$

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

where $\{R'_q\}_{q=1}^m$ is the set of tables for which there is a directed path from R_q to R_j, γ'_q is the predicate condition for R_q in Q for $q \in [1..m]$, and θ' is the part of θ that applies to the tables in the set $\{R_j\} \cup \bigcup_{q=1}^{m} R'_q$. To each table R_i, $i \in [1..r]$, we also add the derived attribute C_i, where $C_i = 1$ when $\texttt{parent}^\theta(R_i)$ is empty and $C_i = \prod_{R \in \texttt{parent}^\theta(R_i)} C_{(R_i,R)}$ otherwise. Now, note that $t.C_{(R_i,R_j)}$ for $t \in R_i$ can also be expressed as follows.

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

We will refer to this 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 that order.

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 $\texttt{parent}^\theta(R_y)$ is empty, then $t.C_y$ will be set to 1. Otherwise, $t.C_y$ will be set to 0 because the defined integrity constraints will guarantee that t can not 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 ($1 \leq i, j \leq r$ 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 $\texttt{parent}^\theta(R_j)$ is empty. 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 $\texttt{parent}^\theta(R_j)$ is empty. 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 $t^{\texttt{old}}.S_j$. As well, 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 $t^{\texttt{new}}.S_j$. 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 $O(|\Sigma|)$ time to perform a refresh on the attribute C_i or the attribute $C_{(R_i,R_j)}$.

Next, we will show how the derived attribute $t.S_{(A,r)}$ can be maintained. W.l.o.g. assume that $A \in R_s$ and $\langle R_s, \ldots, R_r\rangle$ is the directed path in θ from R_s to R_r. We will use the already defined attributes $C_{(R_i,R_j)}$ in θ and introduce the new derived attribute $t.S_{(A,i))}$ for $t \in R_i$ ($i \in [s..r]$) using the following expression.

select $sum(A)$
from R_1 as e_1, ..., R_i as e_i
where $\theta'(e_1, \ldots, e_i)$ and $\gamma_1(e_1)$ and ... and $\gamma_{i-1}(e_{i-1})$ and e_i.ID= :t.ID,

where θ' is the part of θ that applies to the tables in the set $\bigcup_{q=1}^{i} R_q$. We will also introduce to each table in the set $\{R_i\}_{i=s+1}^{r}$ the derived attribute $t.B_{(A,i)}$, where for $t \in R_i$, $t.B_{(A,i)}$ is equal to the result of the following expression.

select $sum(S_{(A,i-1)})$
from R_{i-1} as e
where $\gamma_{i-1}(e)$ and $e.Pf^{\theta}_{(i-1,i)} = t.$ID

Next, note that $t.S_{(A,i)} = t.A$ when $\texttt{parent}^{\theta}(R_i)$ is empty, $t.S_{(A,i)} = t.C_i \cdot t.A$ when $i = s$, and $t.S_{(A,i)}$ is equal to $t.B_{(A,i)} \cdot \prod_{R_q \in \texttt{parent}^{\theta}(R_i), q \neq i-1} t.C_{(R_i,R_q)}$ otherwise (Note that throughout this work we assume that the product of a 0 number of elements is 1).

We will next show how the newly introduced derived attributes can be maintained. Note that we will apply the superscript `new` and `old` to attributes and path functions to denote the new and old value of an attribute. When it is clear from the context which value we are refereing to, the superscript is not written.

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 that order.

First, we will examine the affect of the update on R_y. If $y \notin [s..r]$, then the value of the derived attributes will not change. Therefore, assume that $y \in [s..r]$. A deletion will also not affect the derived attributes. A modification will have an effect only is $y = s$. In this case $t.S_{(A,y)}$ will be recalculated as $t.C_s^{\texttt{new}} \cdot t.A^{\texttt{new}}$. A primitive insertion will introduce a new tuple t to R_y. We will set $t.S_{(A,y)}$ to $t.C_y \cdot t.A$.

Second, consider how the derived attribute $t.B_{(A,i)}$, $i \in [s..r]$ can be refreshed after a primitive update. Such an update will affect at most two tuples in at most one table in $\texttt{parent}^{\theta}(R_i)$. If 0 tuples are affected, then $t.B_{(A,i)}$ will not be affected. Therefore, assume that there is a change to a tuple t' in $R_j \in \texttt{parent}^{\theta}(R_i)$. First, assume that $j \in [s..r]$. If t' is deleted from R_j and $\gamma_j(t')$ holds, then we will decrease $t'.Pf_{(R_j,R_i)}.B_{(A,i)}$ by $t'.S_{(A,j)}$. Similarly, if t' is inserted into R_j and $\gamma_j(t')$ holds, then we will increase $t'.Pf_{(R_j,R_i)}.B_{(A,i)}$ by $t'.S_{(A,j)}$. Lastly, if $t' \in R_j$ is modified from $t'^{\texttt{old}}$ to $t'^{\texttt{new}}$, then we will perform the following actions.

1. If $\gamma_j(t'^{\texttt{old}})$ holds, then we will decrease $t'^{\texttt{old}}.Pf_{(R_j,R_i)}.B_{(A,i)}$ by $t'^{\texttt{old}}.S_{(A,j)}$.

2. If $\gamma_j(t'^{\texttt{new}})$ holds, then we will increase $t'^{\texttt{new}}.Pf_{(R_j,R_i)}.B_{(A,i)}$ by $t'^{\texttt{new}}.S_{(A,j)}$.

In all other cases, no action needs to be performed.

Next, assume that $j \notin [s..r]$. We will examine two subcases: when $i = s$ and when $i \neq s$. If $i = s$, then $S_{(A,i)} = t.C_i \cdot t.A$. If t' is deleted from R_j or inserted into R_j, then we will recalculate $t'.Pf_{(R_j,R_i)}.S_{(A,i)}$. If a tuple in R_j is modified from $t'^{\texttt{old}}$ to $t'^{\texttt{new}}$, then we will recalculate $t'^{\texttt{old}}.Pf_{(R_j,R_i)}.S_{(A,i)}$ and $t'^{\texttt{new}}.Pf_{(R_j,R_i)}.S_{(A,i)}$. The final case is when $i \neq s$. However, in this case $B_{(A,i)}$ will not be affected.

In all cases in which we did not explain how to compute $C_{(A,i)}$, the derived attribute will be refreshed using the formula $t.S_{(A,i)} = t.B_{(A,i)} \cdot \prod_{R_q \in \texttt{parent}^{\theta}(R_i), q \neq i-1} t.C_{(R_i,R_q)}$. In particular, if t' is deleted from R_j or inserted into R_j, then we will recalculate $t'.Pf_{(R_j,R_i)}.S_{(A,i)}$. If a tuple in R_j is modified from $t'^{\texttt{old}}$ to $t'^{\texttt{new}}$, then we will recalculate $t'^{\texttt{old}}.Pf_{(R_j,R_i)}.S_{(A,i)}$ and $t'^{\texttt{new}}.Pf_{(R_j,R_i)}.S_{(A,i)}$.

3.2 Maintaining Materialized Views

Recall that we allow MVs of two types - see Table 2.5.

3.2.1 Materialized Views of the First Type

Consider first a MV V of the first type, i.e with the following underlying query.

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

An insertion of a tuple t in T will also involve an insertion to the MV iff $\gamma(t)$ holds. A deletion of a tuple from T will also involve a deletion from the MV iff $\gamma(t)$ holds. A modification of a tuple t from $t^{\texttt{old}}$ to $t^{\texttt{new}}$ in T_r will involve:

1. an insertion of $t^{\texttt{new}}$ in the MV iff $\gamma(t^{\texttt{old}})$ does not hold, but $\gamma(t^{\texttt{new}})$ does,
2. a deletion of t from the MV iff $\gamma(t^{\texttt{old}})$ holds, but $\gamma(t^{\texttt{new}})$ does not, and
3. a modification of $t^{\texttt{old}}$ to $t^{\texttt{new}}$ in the MV iff $\gamma(t^{\texttt{old}})$ and $\gamma(t^{\texttt{new}})$ both hold.

Note that for each MV V of the first type we will add all possible primitive updates to V to the set of primitive updates.

3.2.2 Materialized Views of the Second Type

Next, consider a MV of the second type (see Table 2.5). We will express the underlying query V as follows:

> select $C_1, \ldots, C_l, A_{l+1}, \ldots, A_a, C$
> from T_r as e
> where $\gamma_r(e)$ and $C > 0$
> group by $A_{l+1}, \ldots, A_a$,

where C and $C_1, \ldots, C_l$ are derived attributes.

For a tuple $t \in V$, $t.C$ is equal to the result of the following query.

select count($*$)
from T_1 as $e_1, \ldots, T_r$ as e_r
where $\theta(e_1, \ldots, e_r)$ and $\gamma_1(e_1)$ and $\ldots$ and $\gamma_r(e_r)$ and $e_r.A_{l+1} = :t.A_{l+1}$ and $\ldots$
and $e_r.A_a = :t.A_a$

For a tuple $t \in V$ the value of $t.C_i$ is equal to the value of the following expression, where $i \in [1..l]$.

select $agg_{c_i}(A_i)$
from T_1 as $e_1, \ldots, T_r$ as e_r
where $\theta(e_1, \ldots, e_r)$ and $\gamma_1(e_1)$ and $\ldots$ and $\gamma_r(e_r)$ and $e_r.A_{l+1} = :t.A_{l+1}$ and $\ldots$
and $e_r.A_a = :t.A_a$

However, the previous subsection does not show how the derived attributes $C, C_1, \ldots, C_l$ can be directly maintained. The reason is that maintaining these attributes may require evaluating critical queries, i.e. maintaining some of the attributes may require logarithmic time.

We will first introduce the derived attribute $C^{\texttt{count}}$ for a tuple $t \in T_r$. We define $t.C^{\texttt{count}}$ as follows.

select count($*$)
from T_1 as $e_1, \ldots, T_r$ as e_r
where $\theta(e_1, \ldots, e_r)$ and $\gamma_1(e_1)$ and $\ldots$ and $\gamma_{r-1}(e_{r-1})$ and $e_r.\texttt{ID} = :t.\texttt{ID}$

Note that Section 3.1.1 show how $C^{\texttt{count}}$ can be efficiently maintained.

Consider next how the derived attribute C can be maintained. We will add the query Q_1 from Table 3.1 to the set of simple critical queries. Note that any primitive update can affect at most two tuples in T_r. First, consider the case when a tuple t is added to T_r as a result of the update. If $\gamma_r(t)$ and $t.C^{\texttt{count}} > 0$, then set $t' = Q_1(t.A_{l+1}, \ldots, t.A_a)$ and increase $t'.C$ by $t.C^{\texttt{count}}$. Similarly, if a tuple t is deleted from T_r as a result of an update, $\gamma_r(t)$ and $t.C^{\texttt{count}} > 0$ both hold, and $Q_1^{\texttt{old}}(t.A_{l+1}, \ldots, t.A_a)$ is not empty, then set $t' = Q_1(t.A_{l+1}, \ldots, t.A_a)$ and decrease $t'.C$ by $t.C^{\texttt{count}}$. Lastly, if a tuple $t^{\texttt{old}} \in T_r$ is modified to $t^{\texttt{new}} \in T_r$, then consider the following actions.

1. If $\gamma_r(t^{\texttt{old}})$ and $t^{\texttt{old}}.C^{\texttt{count}} > 0$ both hold and $Q_1(t^{\texttt{old}}.A_{l+1}, \ldots, t^{\texttt{old}}.A_a)$ is not empty, then set $t' = Q_1(t^{\texttt{old}}.A_{l+1}, \ldots, t^{\texttt{old}}.A_a)$ and decrease $t'.C$ by $t^{\texttt{old}}.C^{\texttt{count}}$.

2. If $\gamma_r(t^{\texttt{new}})$ and$t^{\texttt{new}}.C^{\texttt{count}} > 0$ both hold, then set $t' = Q_1(t^{\texttt{new}}.A_{l+1}, \ldots, t^{\texttt{new}}.A_a)$ and increase $t'.C$ by $t^{\texttt{old}}.C^{\texttt{count}}$.

Note that when a tuple is inserted in V, its initial value for C is set to 0.

Next, consider the case when $agg_{c_i} = \texttt{sum}$ and the derived attribute C_i that needs to be maintained. We will introduce the derived attribute $C_i^{\texttt{sum}}$ to T_r, where for $t \in T_r$, $t.C_i^{\texttt{sum}}$ is equal to the result of the following query.

> `select` $sum(A_i)$
> `from` T_1 `as` $e_1, \ldots, T_r$ `as` e_r
> `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` $\ldots$ `and` $\gamma_{r-1}(e_{r-1})$ `and` $e_r.\texttt{ID} = :t.\texttt{ID}$

Section 3.1.1 shows how the attribute $C_i^{\texttt{sum}}$ can be efficiently maintained. In this case we will also add the query Q_1 from Table 3.1 to the set of simple critical queries. Note that any primitive update can affect at most two tuples in T_r. First, consider the case when a tuple t is added to T_r as a result of the update. If $\gamma_r(t)$ and $t.C^{\texttt{count}} > 0$, then set $t' = Q_1(t.A_{l+1}, \ldots, t.A_a)$ and increase $t'.C_i$ by $t.C_i^{\texttt{sum}}$. Similarly, if a tuple t is deleted from T_r as a result of an update, $\gamma_r(t)$ and $t.C^{\texttt{count}} > 0$ both hold, and $Q_1(t.A_{l+1}, \ldots, t.A_a)$ is not empty, then set $t' = Q_1(t.A_{l+1}, \ldots, t.A_a)$ and decrease $t'.C_i$ by $t.C_i^{\texttt{sum}}$. Lastly, if a tuple $t^{\texttt{old}} \in T_r$ is modified to $t^{\texttt{new}} \in T_r$, then consider the following actions.

1. If $\gamma_r(t^{\texttt{old}})$ and $t^{\texttt{old}}.C^{\texttt{count}} > 0$ both hold and $Q_1(t^{\texttt{old}}.A_{l+1}, \ldots, t^{\texttt{old}}.A_a)$ is not empty, then set $t' = Q_1(t^{\texttt{old}}.A_{l+1}, \ldots, t^{\texttt{old}}.A_a)$ and decrease $t'.C_i$ by $t^{\texttt{old}}.C_i^{\texttt{sum}}$.

2. If $\gamma_r(t^{\texttt{new}})$ and $t^{\texttt{new}}.C^{\texttt{count}} > 0$ both hold, then set $t' = Q_1(t^{\texttt{new}}.A_{l+1}, \ldots, t^{\texttt{new}}.A_a)$ and increase $t'.C_i$ by $t^{\texttt{old}}.C_i^{\texttt{sum}}$.

Note that when a tuple is inserted in V, its initial value for C_i is set to 0.

Next, consider the case when $agg_{c_i} = \texttt{avr}$ and the derived attribute C_i that needs to be maintained. Note that C_i can be computed as a derived attribute with value $C_i^{\texttt{sum}}/C$. Note that this expression is well defined because V stores only tuple for which $C > 0$.

query/update	*value*
query Q_1	select $*$ from V as e where $e.A_{l+1}$= :P_{l+1} and ... and $e.A_a$= :P_a,
query Q_2^i	select A_i from T_1 as e_1, ..., T_r as e_r where $\theta(e_1,\ldots,e_r)$ and $\gamma_1(e_1)$ and ... and $\gamma_r(e_r)$ and $e.A_{l+1}$= :P_{l+1} and ... and $e.A_a$= :P_a order by A_i asc
query Q_3^i	select A_i from T_1 as e_1, ..., T_r as e_r where $\theta(e_1,\ldots,e_r)$ and $\gamma_1(e_1)$ and ... and $\gamma_r(e_r)$ and e_r.ID = :P_1 order by A_i desc
update U_1	insert into $V(A_{l+1},\ldots,A_a)$ values (:$P_{l+1},\ldots,$:P_l)
update U_2	delete from V as e where e.ID = :P_1
update U_3	update V as e set $e.A_{l+1}$ = :P_{l+1}, ..., $e.A_a$ = :P_a where e.ID = :P_{a+1}

Table 3.1: Additional critical queries and updates

Lastly, consider the case when agg_{c_i} = max (agg_{c_i} = min). We will add the query Q_2^i (Q_3^i) from Table 3.1 to the set of critical queries. If a tuple t in V could be affected as a result of a primitive update, then we will recalculate $t.C_i$ as $Q_2^i(t.A_{l+1},\ldots,t.A_s)$ ($Q_3^i(t.A_{l+1},\ldots,t.A_s)$).

We next show how the MV V can be efficiently maintained. The presented refresh calls relies on the code for maintaining the different derived attributes of V. Note however that the refresh code for the different derived attributes of V is executed after executing the described code for efficiently maintaining V. The reason is that the refresh code for the derived attributes needs to see the new state of the database, i.e. the state of the database when the non-derived attributes of V have been updated. Note as well that the following pseudo-code for maintaining V sees the old values for the derived attributes of V, but the new values for the derived attributes of T_r.

If a tuple t is inserted into T_r, then the following refresh code should be executed.

$\texttt{if}\ ((\gamma_r(t))\ \texttt{\&\&}\ (t.C^{\texttt{count}} > 0)\ \texttt{\&\&}\ (Q_1(t.A_{l+1}, \ldots, t.A_a) \equiv \emptyset))$

$\quad U_1(t.A_{l+1}, \ldots, t.A_a)\texttt{;}$

If a tuple t is deleted from T_r, then the following refresh code should be executed.

$\texttt{if}\ ((!\gamma_r(t))\ \texttt{||}\ (Q_1(t.A_{l+1}, \ldots, t.A_a) \equiv \emptyset))\ \texttt{return;}$

$t' = Q_1(t.A_{l+1}, \ldots, t.A_a)\texttt{;}$

$\texttt{if}\ (t'.C == t.C^{\texttt{count}})\ U_2(t'.\texttt{ID})\texttt{;}$

If a tuple in T_r is modified from $t^{\texttt{old}}$ to $t^{\texttt{new}}$, then the following refresh code should be performed.

$\texttt{if}\ ((\gamma_r(t^{\texttt{old}}))\ \texttt{\&\&}\ (Q_1(t^{\texttt{old}}.A_{l+1}, \ldots, t^{\texttt{old}}.A_a) \not\equiv \emptyset))\ \{$

$\quad t' = Q_1(t^{\texttt{old}}.A_{l+1}, \ldots, t^{\texttt{old}}.A_a)\texttt{;}$

$\quad \texttt{if}\ (t'.C == t.C^{\texttt{count}})\ \{$

$\qquad \texttt{if}\ ((!\gamma_r(t^{\texttt{new}}))\ \texttt{||}\ (t^{\texttt{new}}.C^{\texttt{count}} == 0)\ \texttt{||}\ (Q_1(t^{\texttt{new}}.A_{l+1}, \ldots, t^{\texttt{new}}.A_a) \not\equiv$
$\emptyset))$

$\qquad\quad \{\ U_2(t'.\texttt{ID});\ \texttt{return;}\ \}$

$\qquad U_3(t^{\texttt{new}}.A_{l+1}, \ldots, t^{\texttt{new}}.A_a, t'.\texttt{ID})\texttt{;}$

$\qquad \texttt{return;}$

$\quad \}$

$\}$

$\texttt{if}\ ((\gamma_r(t^{\texttt{new}}))\ \texttt{\&\&}\ (t^{\texttt{new}}.C^{\texttt{count}} > 0)\ \texttt{\&\&}\ (Q_1(t^{\texttt{new}}.A_{l+1}, \ldots, t^{\texttt{new}}.A_a) \equiv \emptyset))$

$\quad U_1(t^{\texttt{new}}.A_{l+1}, \ldots, t^{\texttt{new}}.A_a)\texttt{;}$

Note that the insert pseudo-code first checks that the inserted tuple t will contribute to V, i.e. it passes the γ_r predicate and $t.C^{\texttt{count}} > 0$. Then it checks that a tuple in V with the same values for $\{A_i\}_{i=l+1}^{a}$ as t does not already exist. If both conditions pass, then a new tuple is inserted in V. If the first condition does not pass, then t will not affect V. If the second condition does not pass, then a tuple for t is already inserted in V and therefore again nothing needs to be done.

The delete pseudo-code first checks wether the deleted tuple t will require the corresponding tuple t' from V to be deleted. This is the case only when t passes the γ_r, the tuple t' exists and $t.C^{\texttt{count}} = t'.C$.

The update pseudo-code basically performs the delete pseudo-code followed by the insert pseudo-code. The only modification is that when a deletion and an insertion are both required the pseudo-code performs a modification instead.

Note that the described pseudo-codes are efficient as long as the access plans for the operation in Table 3.1 are efficient. We can guarantee that by adding the operations to the set of critical queries and to the set of primitive updates, respectively.

Chapter 4

Finding an Encoding with Endogenous Indices

In this chapter we describe how to solve the problem at hand using endogenous indices. In particular, in the first part of this chapter we define the PARTs for endogenous indices and prove some of their properties. In the second part of the chapter we show how the steps from Figure 1.6 can be applied for endogenous indices. Note however that some of the step are general and apply to both endogenous indices and compact structures.

4.1 ePARTs

In this section we solve the problem of creating the smallest possible physical design that can answer a set of critical queries, where we require that the physical design consists exclusively of extended indices that can be efficiently maintained.

Note that for every sSQL there exist a number of extended indices that can be used to efficiently answer it. For example, consider the following query over our example company schema from 1.5.

select $*$
from EMP
where $age >$:P

The extended index $\langle R, \langle age, A\rangle\rangle$ can be used to efficiently answer the query regardless of the value for the attribute A. Note as well that every extended index can be used to efficiently answer a number of critical queries - see Theorem 4.

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

We will use *extended index Parameterized Access Requirement Types* (PARTs) to describe a set of indices. The definition of the syntax of an ePART follows.

Definition 44 (ePART syntax) *An ePART $\mathcal{P}$ is an unordered node-labeled tree, where the syntax of each node label is of the following form.*

$$
\begin{aligned}
\textit{node label} &::= \langle\{\gamma_1,\ldots,\gamma_m\}, R, \langle L\rangle\rangle \mid \langle\{\gamma_1,\ldots,\gamma_m\}, R, \langle\rangle\rangle \mid \langle R, \langle L\rangle\rangle \mid \langle R, \langle\rangle\rangle \\
L &::= E \mid F \mid E,\ L \mid F,\ L \\
E &::= \{A_1,\ldots,A_a\} \\
F &::= A_1,\ldots,A_a
\end{aligned}
$$

Note that in the above grammar we require that $a > 0$ and $m > 0$.

Given a node label $\langle\{\gamma_1,\ldots,\gamma_m\}, R, \langle L\rangle\rangle$, we will refer to the E parts of L as E-components and to the F parts - as F-components. We will refer to R as the node's table and write $\texttt{table}(n)$ *to denote it, to $\{\gamma_1,\ldots,\gamma_m\}$ as the node's γ-condition and write $\gamma(n)$ to denote it, and to L as the node's ordering label and write $\mathcal{L}(n)$ to denote it. Note that we require that a γ-condition can be defined only for the root node of $\mathcal{P}$ and we will therefore also refer to it as the γ-condition of the ePART and write $\gamma(\mathcal{P})$. We will use $|L|$ to denote the number of attributes referenced in a ordering label.*

We next present several intermediate definitions that will help us explain the additional requirements that we impose on valid ePARTs.

Definition 45 (concatenating ordering labels) *If L_1 and L_2 are ordering labels, then we define their concatenation as follows.*

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

Definition 46 (the $\lfloor\cdot\rfloor$ operation) *The following function converts a set of attributes into either the empty string an E-component.*

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

Definition 47 (permutations for an ePART ordering label) *Let L be an ePART ordering label. We will use $\Pi(L)$ to denote result of applying the κ operations to the E-components of the ordering label one or more times. Let E be the E-component $\{A_1,\ldots,A_a\}$ and let $\langle A'_1,\ldots,A'_a\rangle$ be a permutation of $\langle A_1,\ldots,A_a\rangle$. Then the κ operations converts E to one of the following expressions:*

- $\{A'_1,\ldots,A'_a\}$,
- $A \circ \lfloor E - \{A\}\rfloor$,
- $\lfloor E - \{A\}\rfloor \circ A$, *or*
- $\{A'_1,\ldots,A'_k\}$,$\{A_{k+1},\ldots,A_a\}$,

where $1 \leq k < a$. We will call Π a complete permutation if it converts L into an expression in which all E-components are of size at most 1. We will use Π^c to denote complete permutations.

Definition 48 (permutations for an ePART) *A* permutation Π *for an ePART $\mathcal{P}$ applies some permutation to some of the E-components of some of the ordering labels of the ePART. A* complete permutation Π^c *for an ePART $\mathcal{P}$ applies complete permutations to all ordering labels.*

Definition 49 (fixed ePART) *A fixed ePART is and ePART that has no E-components of size greater than 1 in its ordering labels.*

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

Definition 50 (converting a fixed ePART into an extended index) *Let $\mathcal{P}$ be a fixed ePART. We will convert it into an extended index by performing the following actions:*

1. *Fix the three order.*
2. *If $\{n_i\}_{i=1}^{k}$ are the children of the node n and $\bigcup_{i=1}^{k} \pi_{\mathtt{ID}}\mathtt{table}(n_i) \subset \pi_{\mathtt{ID}}\mathtt{table}(n)$, then add a new child node to n with table R that has the attributes of $table(n)$ such that $\pi_{\mathtt{ID}}R$ includes exactly the values that are in $\pi_{\mathtt{ID}}table(n)$ but not in $\bigcup_{i=1}^{k} \pi_{\mathtt{ID}}\mathtt{table}(n_i)$.*
3. *Examining the ordering labels of the leaf nodes of $\mathcal{P}$. For each ordering label that does not contain the attribute* `ID`*, append the* `ID` *attribute to the end of the ordering label.*

We required that the described procedure produces a valid artificial index.

Definition 51 (valid ePART) *An ePART $\mathcal{P}$ is valid iff for every complete permutation Π^c of $\mathcal{P}$, $\Pi^c(\mathcal{P})$ can be converted into a valid extended index.*

Definition 52 (cover of a set of indices) *Let $\bar{X}$ be a set of extended indices. Then* $\mathtt{cover}(\bar{X}) = \{X | \exists X' \in \bar{X}, Q(X') \subseteq Q(X)\}$.

Definition 53 (extended indices represented by a fixed ePART) *Let $\mathcal{P}$ be a fixed ePART. Then we will use $X(\mathcal{P})$ to denote the set of extended indices that $\mathcal{P}$ can be converted into. We will also use $X^c(\mathcal{P})$ to denote the set* $\mathtt{cover}(X(\mathcal{P}))$.

(*name*)	(*query*)	(*ePART*)

Q_1	`select *`
	`from` V_1
	`order by` age `asc`
		$\mathcal{P}_1 = \langle V_1, \langle age \rangle \rangle$

Q_2	`select *`
	`from` V_2
	`order by` age `asc`, $salary$ `asc`
		$\mathcal{P}_2 = \langle V_2, \langle age, salary \rangle \rangle$

Q_3	`select *` `from` V_3 `where` $age = {:}P_1$ `and` $name = {:}P_2$	$\mathcal{P}_3 = \langle V_3, \langle \{age, name\} \rangle \rangle$

Table 4.1: Three example critical queries and the corresponding ePARTs

Definition 54 (extended indices represented by an ePART) *Let $\mathcal{P}$ be an ePART. Then we will use $X(\mathcal{P})$ to denote the set of extended indices $\bigcup_{\Pi^c} Y(\Pi^c(\mathcal{P}))$, where Π^c varies over all valid complete permutations for $\mathcal{P}$. We will also use $X^c(\mathcal{P})$ to denote the set* `cover`$(X(\mathcal{P}))$. *We will refer to $X(\mathcal{P})$ as the* underlying *set of extended indices for $\mathcal{P}$ and to $X^c(\mathcal{P})$ as the set of extended indices for $\mathcal{P}$.*

Consider our example company schema from Figure 1.5 and the MVs from Table 2.7. Suppose that we are given the critical queries shown in Table 4.1 and suppose that V_2 and V_3 are disjoint. When solving the problem at hand, we will first build the ePARTs shown in Table 4.1. Note that for an ePART $\mathcal{P}$, $X^c(\mathcal{P})$ represents exactly the set of indices that can be used to efficiently answer the critical query that generated the ePART. Note as well that $\mathcal{P}_1$ and $\mathcal{P}_2$ are fixed ePART. On the other hand, there are two valid complete permutations for $\mathcal{P}_3$: one will convert it into the fixed ePART $\langle V_3, \langle age, name \rangle \rangle$ and the other into the fixed ePART $\langle V_3, \langle name, age \rangle \rangle$.

The next step of our solution is ePART merging. For example, the presented in Table 4.1 ePART could be merged into the ePART $\mathcal{P} = \langle \{\gamma\}, V_1, \langle age \rangle, [\langle V_2, \langle salary \rangle \rangle, \langle V_3, \langle \{name\} \rangle \rangle] \rangle$, where $\gamma(t)$ holds for a tuple $t \in V_1$ iff $t.age < 30$ (i.e. γ selects the tuples in V_1 that are also in V_2). Note that we do not need to add to the ePART a γ-condition that selects the tuples in V_3 because the query Q_3 contains a partial match predicate on the attribute *age*. Note that $X^c(\mathcal{P}) = \bigcap_{i=1}^{3} X^c(\mathcal{P}_i)$, i.e. the set of indices of the created ePART is the intersection of the extended indices of the old ePARTs.

The final step of our algorithm involves converting the created ePART into an extended index. The extended index $X = \langle\{\gamma\}, V_1, \langle age\rangle, [\langle V_2, \langle salary, \mathtt{ID}\rangle\rangle, \langle V_3, \langle name, \mathtt{ID}\rangle\rangle, \langle V_4, \langle \mathtt{ID}\rangle\rangle]\rangle$ is one of the smallest possible indices, where V_4 is defined in Table 2.7. Note that $X \in X(\mathcal{P})$ and therefore $X \in X^c(\mathcal{P})$.

What we have done in this example is to first represent the set of extended indices that can be used to efficiently answer each of the original queries (those are the sets $\{X^c(\mathcal{P}_i)\}_{i=1}^{3}$). It turns out that the intersection of three sets (i.e. $X^c(\mathcal{P})$) is not empty. As a final step, we choose the index X from the set $X^c(\mathcal{P})$ as one of the representative of the set with the smallest size.

4.2 The Solution Steps

In this section we explain the steps in Figure 1.6 for extended indices. However, some of the steps also apply for compact structures.

4.2.1 Step 1 - Handling Non-critical Queries and Updates

Consider a non-critical query Q. We can use an off-the-shelf query optimizer to produce an access plan for Q. We assume that the query plan is in the shape of a tree, where the leaf nodes are sequential scans or index scans of tables and the non-leaf nodes are database operations (e.g. selection, projection, union, sort, loop-join, merge-join, etc.). Our first job will be to rewrite the query plan by substituting each leaf-node that references both part of the critical and non-critical part of the database with the union operations on two nodes, where one of the nodes references exclusively the critical part of the database and the other node references exclusively the non-critical part of the database.

We next examine the part of the query plan that access the critical part of the database. In particular, we identify nodes n in the query plan that have the following properties.

1. n represents only data from the critical part of the database.

2. The data represented by n could expressed using a μSQL query.

3. The parent of n in the query plan fails to satisfy at least one of the above two properties.

For each such plan we add the identified μSQL query to the set of critical queries. It is easy to see that the so identified μSQL queries cover the critical part of the database accessed by the query. Note that a created μSQL query could also reference a newly introduced MV, in which case we would add the view to the database schema.

We next examine the part of the query plan that accesses the non-critical part of the database. In particular, consider all the nodes n in the query plan that have the following properties.

1. n represents only data from the non-critical part of the database.
2. The parent of n in the query plan does not represent only data from the non-critical part of the database.

For each such node n that has the desired property, we will add the query that corresponds to n to the non-critical set of queries.

4.2.2 Step 2 - Handling Critical Updates

Recall that a critical update either adds a tuple to a base table R of the input schema Σ, or delete or modify the tuples in a specified base table R that pass the condition of a predicate γ over an admissible predicate system. If there is a modification, then the modification function f should be able to compute $f(x)$ in $O(|x|)$ time. Given such a critical update, we will construct an access plan that references a critical query and a sequence of primitive updates. In order for the constructed sequence of critical updates to be consistent with the database schema, we require that the modification function f is monotonic in certain cases.

Note that an insertion is already a primitive update. A modification has the following syntax.

`update` R `as` e
`set` $e.A = f(e.A)$
`where` $\gamma(e)$

Let Q be the query:

```
select e.ID
from R as e
where γ(e)
```

and U be the primitive update:

```
update R as e
set e.A = f(e.A)
where e.ID = :P_1
```

Then, the presented modification can be executed using the following access plan.

```
for t ∈ Q
  execute U(t.ID);
```

However, such a query plan may not be consistent. For example, consider our example schema from Figure 1.5 and the functional dependency EMP($name \to salary$). Now, an update that increases everyone's salary by 10% will not violate the functional dependency at the end of the update, but it may violate it after the application of the first primitive update. We will ensure that our update decomposition algorithm is consistency preserving by requiring that the function $f(\cdot)$ for all updates that update attributes that appear in a functional dependency are monotonic. For example, if f is non-decreasing, then we can make the above query plan consistency preserving relative to any set of allowed integrity constraints by changing Q to:

```
select ID
from R
where γ(x)
order by A desc.
```

Similarly, if f is non-increasing, then we can make the above query plan consistency preserving relative to any set of allowed integrity constraints by changing Q to:

```
select ID
from R
where γ(x)
order by A asc.
```

Note that the condition that f be monotonic is a sufficient, but not necessary, condition for guaranteing that critical updates are broken into a sequence of consistency preserving primitive updates in the presence of integrity constraints. The constructed query Q is added to the set of critical queries and the primitive update U is added to the set of primitive updates.

4.2.3 Step 3 - Semantic Query Rewriting

The third step of the algorithm is semantic query rewriting of the critical queries. In particular, we will be interested in join elimination transformations (see [15]), although other semantic optimization rewrites may be advantageous. Note that in this work we also consider an order optimization technique along the lines presented in [8]. However, this optimization is done in Step 8 because the technique is used to simplify a query plan rather than a query.

We next present a modification of the join elimination technique from [15] and an extension of it that is novel to the best of our knowledge.

Consider a valid critical query Q that is based on the tables $\{R_i\}_{i=1}^{k}$, $k > 2$. Let θ be the join condition of the query and γ_i be the predicate in Q for R_i, $i = 1$ to k.

A simplified version of the rewrite suggested in [15] adopted to our query and schema model goes as follow:

1. Try to find a table R_i, $1 \leq i \leq k$, that has the following properties:
    - $\gamma_i =$ **TRUE**,
    - no parameterized restrictions are defined on R_i,
    - no attributes of R_i appear in the `select` part of Q, and
    - in the graph θ there are no outgoing edges from R_i and there is a single incoming edge.

2. If such a table is found, then remove R_i from the `from` condition of Q and remove the join condition between R_i and the single table it joins with from the `where` condition of Q. Then go to Step 1.

The rewrite is correct because the eliminated tables do not contribute to the query result. One possible extension of this rewrite is described next.

1. Try to find tables R_x, R_y, and R_z such that:
   - there is a directed edge between R_x and R_y and w.l.o.g. we assume that it is labeled A;
   - there is a directed edge between R_y and R_z and w.l.o.g. we assume that it is labeled B;
   - the attributes of R_y do not appear in the `select` part of Q;
   - $\gamma_y = \texttt{TRUE}$;
   - no parameterized restrictions are defined on R_y; and
   - the integrity constraint $\texttt{eq}(R_x : A.B = C)$ holds for some attribute C of R_x.
2. If such tables are found, then replace $R_x.A = R_y \wedge R_y.B = R_z$ with $R_x.C = R_z$ in θ, delete R_y from the `from` part of Q, and then goto Step 1.

It is easy to show that the modified query is equivalent to the original one. In particular, the join condition between R_x and R_z was not changed, while the table R_y was deleted because none of its attributes are projected out in the query and its tuples are not filtered by the `where` condition of the query.

In order to demonstrate the second rewrite, consider the following query over our example company schema from Figure 1.5.

`select` e.*
`from` EMP `as` e, DEP `as` d, COMP `as` c
`where` $e.dep = d.$ID `and` $d.com = c.$ID `and` $c.rank > 5$

If we know that the integrity constraint $\mathtt{eq}(\mathtt{EMP} : \mathit{dep.comp} = \mathit{company})$ holds, then we can rewrite the query as follows.

```
select e.*
from EMP as e, COMP as c
where e.company = c.ID and c.rank > 5
```

4.2.4 Step 4 - μSQL Query Breakup

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

Before presenting our algorithm for decomposing μSQL queries, we examine how a query Q of an intermediate query type can be broken down into simpler queries. The examined query has the following syntax:

```
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 a valid join graph in the shape of a tree with root R_1.

We will first create r queries, one for each table in the set $\{R_i\}_{i=1}^{r}$. The query Q_{R_i} that corresponding to R_i is:

```
select x.B_1, ..., x.B_b, x.ID
from R_i as x
where x.ID = :P,
```

where $\{B_i\}_{i=1}^{b} = \mathtt{Attr}(Q, R_i)$. Next, consider the following pseudo-code.

```
(1) tuple extract_attributes(table R, param P, join tree θ){
(2)     tuple t = Q_R(P);
(3)     vector x=attributes(t);
(4)     for R' ∈ child^θ(R)
(5)       param 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 $|def(Q)|$ time by executing `extract_attributes`$(\mathtt{R}_1, \mathtt{P}_1, \theta)$. In line 3 the function `attributes` extracts the non `ID` attributes of t into x. In line 4 the function $\mathtt{child}^{\theta}(R)$ returns the tables that are direct children of the table R_i 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 those values to x. First, note that the presented pseudo-code returns the result of Q because it traverses the join tree defined by θ to collect the values of all required attributes. Second, note that each query Q_i, $i = 1$ to r, can be executed in constant time. Calling 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. Note that from now on we will use `extract_attributes(`P_1, θ`)` to refer to `extract_attributes(1,`P_1, θ`)`.

We next present our algorithm for decomposing μSQL into simple queries. Recall that a μSQL query has the following general syntax.

`select` $A_1, \ldots, A_a$
`from` R_1 `as` $x_1, \ldots, R_r$ `as` x_r
$[^1$`where` $\theta(x_1, \ldots, x_r)$ `and` $\gamma_1(x_1)$ `and` $\ldots$ $\gamma_k(x_k)$ `and`
$B_1 = {:}P_1$ `and` $\ldots$ `and` $B_b = {:}P_b$ `and` $[^2 B_{b+1}$ `between` ${:}P_{b+1}$ `and` ${:}P_{b+2}]^2]^1$
$[^3$`order by` $[^4 B_{d+1}\ dir_{b+1},]^4\ D_1\ dir_1, \ldots, D_d\ dir_d]^3$,

where θ is a valid join graph for the tables $\{R_i\}_{i=1}^{r}$, $1 \leq k \leq r$, $B_i \in Attr(R_1)$, the subgraph induced by the nodes $\{R_i\}_{i=1}^{k}$, which we will call G^{-t}, is an inverse tree in θ with root R_1, "$[B_{b+1}\ dir_{d+1},]\ D_1\ dir_1, \ldots, D_d\ dir_d$" is a valid order for G^{-t} and $[^i\cdot]^i$ denotes an optional component for $i = 1$ to 4. We require that if an optional component is present and it is contained in another optional component, then the outer optional component is also present. We also require that if optional component 2 is present, then so is optional component 4. Finally, we impose the following restrictions on the tables $\{R_i\}_{i=k+1}^{r}$:

1. For all x, $k < x \leq r$, there does not exist a directed path from R_x to a table R_y in θ s.t. $1 \leq y \leq k$.

2. For all x, $k < x \leq r$, there exist a unique y s.t. $1 \leq y \leq k$ and there exists a directed path from R_y to R_x in θ that has the property that R_y is the only table in the path with subscript in the range $[1, k]$.

3. Let $k < x \leq r$ and y be corresponding unique number described in condition 2. Then there exists a unique directed path from R_y to R_x.

Our first step will be to add to the table R the derived attribute S_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 $t.S_R^Q$ is equal to the result of the following query for $t \in R$.

> $\texttt{select}$ count(*)
> $\texttt{from}\ R'_1\ \texttt{as}\ x_1, \ldots, R'_m\ \texttt{as}\ x_m$
> $\texttt{where}\ \theta(x_1, \ldots, x_m)\ \texttt{and}\ \gamma'_1(x_1)\ \texttt{and} \ldots \texttt{and}\ \gamma'_m(x_m)$
> $\texttt{and}\ x_1.Pf^{\theta}_{(R'_1,R)} = :t.\texttt{ID}\ \texttt{and} \ldots \texttt{and}\ x_s.Pf^{\theta}_{(R'_s,R)} = :t.\texttt{ID}$

In the above query $\{R'_i\}_{i=1}^m = \texttt{ancestor}^{G^{-t}}(R)$, $\{R'_i\}_{i=1}^s = \texttt{parent}^{G^{-t}}(R)$, and γ'_i is the predicate in Q for R'_i for $i = 1$ to k.

W.l.o.g. we assume that the tables $\{R_i\}_{i=1}^k$ are ordered in such a way so that:

- R_1 is the root of G^{-t},
- if $R_x \in \texttt{parent}^{G^{-t}}(R_y)$, $1 \leq x, y \leq k$, 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 for which $\texttt{parent}^{G^{-t}}(R_i) = \emptyset$ we define V_i using the underlying query:

> $\texttt{select}\ x.*$
> $\texttt{from}\ R_i\ \texttt{as}\ x$
> $\texttt{where}\ \gamma_i(x)$.

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

$\texttt{select}\ x.*$
$\texttt{from}\ R_i\ \texttt{as}\ x$
$\texttt{where}\ \gamma_i(x)\ \texttt{and}\ x.S^Q_{R_i} > 0.$

Let Q_1 be the query:

$\texttt{select}\ \mathit{Attr}(Q, V_1)$
$\texttt{from}\ V_1\ \texttt{as}\ x$
$[^1\texttt{where}\ B_1 = {:}P_1\ \texttt{and}\ \ldots\ \texttt{and}\ B_b = {:}P_b\ \texttt{and}\ [^2 B_{b+1}\ \texttt{between}\ {:}P_{b+1}\ \texttt{and}\ {:}P_{b+2}]^2]^1$
$[^3\texttt{order by}\ [^4 B_{d+1}\ \mathit{dir}_{b+1},]^4\ D_1\ \mathit{dir}_1, \ldots, D_d\ \mathit{dir}_d]^3,$

where the optional components are present exactly when they are present in Q, and Q_i, $2 \leq i \leq k$ be the query:

$\texttt{select}\ \mathit{Attr}(Q, V_i)$
$\texttt{from}\ V_i\ \texttt{as}\ x$
$\texttt{where}\ x.\mathit{Pf}^{\theta}_{(R_i, \texttt{child}^{G^{-t}}(R_i))} = {:}P$
$\texttt{order by}\ O_i.$

Note that we have used O_i to refer to the part of the ordering condition in Q that references attributes from R_i, $1 \leq i \leq k$. As well, note that $\mathit{Pf}^{\theta}_{(R_i, \texttt{child}^{G^{-t}}(R_i))}$ is well defined because $\texttt{child}^{G^{-t}}$ is always a single node since G^{-t} is an inverse tree.

Then Q can be answered using the following query plan.

$\texttt{for tuple}\ t_1 \in Q_1(P_1, \ldots, P_b, [P_{b+1}, P_{b+2}])\{$
$\quad \texttt{for tuple}\ t_2 \in Q_2(t_{\texttt{index}(\texttt{child}^{G^{-t}}(R_2))}.\texttt{ID})\{$
$\quad \ldots$
$\quad\quad \texttt{for tuple}\ t_k \in Q_k(t_{\texttt{index}(\texttt{child}^{G^{-t}}(R_k))}.\texttt{ID})\{$
$\quad\quad\quad \texttt{tuple}\ t = \emptyset;$
$\quad\quad\quad \texttt{for}\ (\texttt{int}\ i = 1; i \leq k; i++)$
$\quad\quad\quad\quad t += \texttt{extract\_attributes}(t_i.\texttt{ID}, \theta_i);$
$\quad\quad\quad \texttt{send}\ t\ ;$
$\quad\quad \}$

```
(10)       ...
(11)     }
(12) }
```

Note that we have used the `index` function to return the subscript of a relation, i.e. $\mathtt{index}(R_i) = i$ for $i = 1$ to m. As well, 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 $[1..k]$ in the path$\}$.

First, note that the query plan returns exactly the tuples in Q. The loops in Lines 1-4 traverse G^{-t} and scan only tuples that pass the `where` condition of Q Lines 5-8 then traverse the subtrees rooted at each of the nodes in which all nodes except for the root nodes are based on tables with subscripts in the range $[k+1, r]$. The required attributes of the scanned tuples are stored in t and Line 8 of the pseudo-code a tuple from the query result is returned.

Second, note that the presented 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 either the empty ordering or total ordering 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 $\{R_i\}_{i=1}^{k}$, and all the `extract_attributes` queries that need to be executed for a single tuple of the query result will take $O(r)$ time. Since for each tuple of the query result k queries of the first type need to be executed, the total worst case execution time to retrieve a tuple from Q will be $O(log(x) \cdot k + r) = O(log(x) \cdot |def(Q)|)$.

4.2.5 Step 5 - Converting sSQL Queries into ePARTs

Table 4.2 shows the ePARTs that we will produce for each given sSQL query type (see Table 4.3). Note that for a sSQL query of type 3 we create the ePART $\langle \pi^{d}_{D_1,\ldots,D_d}(R), \langle\rangle\rangle$

(*query type*)	(*PART*)
(1)	$\langle \pi^d_{D_1,\ldots,D_d}(R), \langle \{A_1,\ldots,A_l\}, \langle A_{l+1},\ldots,A_a\rangle\rangle\rangle$
(2)	$\langle \pi^d_{D_1,\ldots,D_d}(R), \langle \{A_1,\ldots,A_l\}, \langle A_{l+1},\ldots,A_a\rangle\rangle\rangle$
(3)	$\langle \pi^d_{D_1,\ldots,D_d}(R), \langle\rangle\rangle$

Table 4.2: The PARTs for the three sSQL query types

and not the ePART $\langle \pi^d_{D_1,\ldots,D_d}(R), \langle\{\mathtt{ID}\}\rangle\rangle$ because every extended index has the capability of returning a tuple from the index given the `ID` of the tuple.

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

Theorem 7 (correctness of ePART creation) *If the sSQL query Q generates the ePART $\mathcal{P}$ in this step, then:*

1. *$X^c(\mathcal{P})$ contains exactly the set of extended indices that can be used to efficiently answer Q.*

Proof: 1. First, suppose that Q is an sSQL query of third type (see Table 4.3), i.e. $\mathcal{P} = \langle \pi^d_{D_1,\ldots,D_d}(R), \langle\rangle\rangle$. Note that $X(\mathcal{P})$ contains the single extended index $X = \langle \pi^d_{D_1,\ldots,D_d}(R), \langle \mathtt{ID}\rangle\rangle$.

$\Leftarrow$ If $X_1 \in X^c(\mathcal{P})$, then by Definitions 54 and 52 it follows that $Q(X) \subseteq Q(X_1)$, which is what we wanted to prove.

$\Rightarrow$ Suppose that X_1 can be used to efficiently answer Q, i.e. $Q \in Q(X_1)$. This implies that X_1 can be used to efficiently answer the queries:

`select` $\bar{A}$
`from` R,

where $\bar{A} \subseteq \{D_i\}_{i=1}^d$. But those are exactly the set of queries that X can be used to efficiently answer. This implies that $Q(X) \subseteq Q(X_1)$, i.e. $X_1 \in \mathtt{cover}(\{X\})$ or $X_1 \in X^c(\mathcal{P})$.

2. Second, suppose that Q is an sSQL query of type 1 or 2 (see Table 4.3), i.e. $\mathcal{P} = \langle \pi^d_{D_1,\ldots,D_d}(R), \langle \{A_1,\ldots,A_l\}, A_{l+1},\ldots,A_a\rangle\rangle$. Then $X(\mathcal{P})$ contains the extended indices of the form: $\langle \pi^d_{D_1,\ldots,D_d}(R), \langle A'_1,\ldots,A'_l, A_{l+1},\ldots,A_a, \mathtt{ID}\rangle\rangle$, where $\langle A'_1,\ldots,A'_l\rangle$ is a permutation of the attributes $\{A_i\}_{i=1}^a$. We will denote this set as $\bar{X}$.

(*type*)	(*query*)
(1)	`select` $D_1, \ldots, D_b$ `from` R [`where` $A_1 =$:P_1 `and` ... `and` $A_k =$:P_k]
(2)	`select` $D_1, \ldots, D_b$ `from` R `where` $A_1 =$:P_1 `and` ... `and` $A_l =$:P_l `and` A_{l+1} `between` :P_{l+1} and :P_{l+2} [`order by` A_{l+1} dir_{l+1} ..., A_m dir_m]
(3)	`select` $D_1, \ldots, D_b$ `from` R `where` $A_1 =$:P_1 `and` ... `and` $A_l =$:P_l [`order by` A_{l+1} dir_{l+1} ..., A_m dir_m]

$1 \leq b \leq d$, $1 \leq k \leq a$, $l+1 \leq m \leq a$

Table 4.3: A set of SQL queries

$\Leftarrow$ Suppose that $X_1 \in X^c(\mathcal{P})$. Then by Definitions 54 and 52 it follows that if $X \in \bar{X}$, then $Q(X) \subseteq Q(X_1)$. But the extended indices in $\bar{X}$ can be used to efficiently answer Q and therefore X_1 can efficiently answer Q.

$\Rightarrow$ Suppose that X_1 can be used to efficiently answer Q, i.e. $Q \in Q(X_1)$. This implies that X_1 can be used to efficiently answer the queries shown in Table **??**. But if in extended index X_2 can be used to efficiently answer the queries shown in Table **??**, then it must be the case $Q(X) \subseteq Q(X_2)$ for some $X \in \bar{X}$ (see Theorem **??**). This implies that $X_1 \in$ `cover`$(\bar{X})$ or $X_1 \in X^c(\mathcal{P})$.

■

4.2.6 Step 6 - Making PARTs Efficiently Maintainable

This step applies only to cPARTs because ePARTs are based on base tables or MVs that are efficiently maintainable.

4.2.7 Step 7 - PART Merging

In this section we present a procedure for merging ePARTs (i.e. set of extended indices). In order for a set of ePARTs to be mergable, it must be the case the intersection of the extended indices of the ePARTs is not empty. If the presented procedure merges a set of ePARTs into a single ePART, then the extended indices of the resulting ePART is the intersection of the extended indices of the initial ePARTs.

Merging Ordering Labels

A procedure for merging ePARTs required a way for merging the ordering labels inside them. We next present an algorithm that does this and we use $\oplus$ to denote it. Informally, the method succeeds when one of the labels is a prefix of the other under some permutation of the attributes in the E-components of the ordering labels. The actual definition of the $\oplus$ operation follows.

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

Note that in the above pseudo-code we have used E to denote a non-empty set, i.e. a string of type $\{\ldots\}$ and A to denote a single attribute. Note as well that the "$\oplus$" function is partial, i.e. not every two ordering labels are mergable. Since "$\oplus$" is a commutative and an associative operation, we will use $\bigoplus_{i=1}^{k} L_i$ for $L_1 \oplus (L_2 \oplus (\ldots (L_{k-1} \oplus L_k) \ldots))$.

Definition 55 (compatible attribute orderings) *Let $\{L_i\}_{i=1}^{k}$ be k ordering labels that* *are F-components. We will say that the attribute ordering defined by $\{L_i\}_{i=1}^k$ are compatible iff for $i, j \in [1..k]$ either L_i is a prefix of L_j or L_j is a prefix of L_i.*

Theorem 8 *1. If $\bigoplus_{i=1}^{k} L_i$ returns $L_{k+1} \neq$ `UNDEFINED`, then:*

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

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

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

Proof: 1.1 Suppose that $\{\Pi_i^c\}_{i=1}^k$ are complete permutations that convert $\{L_i\}_{i=1}^k$ into the F-components $\{L_i'\}_{i=1}^k$ and the attribute orderings defined by $\{L_i'\}_{i=1}^k$ are compatible. We will prove the there exists a complete permutation Π_{k+1}^c that converts L_{k+1} into L_{k+1}' and for which the attribute orderings defined by $\{L_i'\}_{i=1}^{k+1}$ are compatible.

Consider the j^{th} attribute of $\{L_i'\}_{i=1}^k$. For the ordering labels in the set $\{L_i'\}_{i=1}^k$ for which $|L_i| \geq j$, this attribute will be the same and let it be equal to A_j. Let $|L_{k+1} = l|$ and let Π_{k+1}^c be the permutation that converts L_{k+1} into $A_1, \ldots, A_l$. Then for this permutation the attribute orderings $\{L_i'\}_{i=1}^{k+1}$ will be compatible.

1.2 Suppose that Π_{k+1}^c is a complete permutation that converts L_{k+1} into $L_{k+1}' = A_1, \ldots, A_l$. Let $\{\Pi^c\}_{i=1}^k$ be the complete permutations that convert L_i into $L_i' = A_1, \ldots A_{|L_i|}$. Then the attribute ordering defined by $\{L_i'\}_{i=1}^k$ are compatible.

2. Next, we will prove the second part of the theorem. If $\bigoplus_{i=1}^{k} L_i$ returns `UNDEFINED`, then suppose that this happens after k recursive calls of the "$\oplus$" operation. Let L' be the prefix of result generated during those k calls ($|L'| = k$) and suppose that the $\oplus$ method returned

`UNDEFINED` when trying to merge L^1 and L^2, i.e.: $\Pi_1(L_1) = L' \circ L^1$, $\Pi_2(L_2) = L' \circ L^2$, $|L^1|, |L^2| > 0$, Π_1 and Π_2 are valid permutations for L_1 and L_2, respectively, and one of the following is true for L^1 and L^2.

1. L^1 starts with the E-component E_1, L^2 starts with the E-component E_2, $E_1 \nsubseteq E_2$, and $E_2 \nsubseteq E_1$.

2. One of the ordering labels starts with the E-component E, the other starts with the attribute A, and $A \notin E$.

3. L^1 starts with the attribute A, L^2 starts with the attribute B, and $A \neq B$.

We will prove that in all three cases there does not an exists complete permutations $\{\Pi_i^c\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into two ordering labels which attribute orderings are compatible.

Consider first case 1 and suppose that there exist complete permutations $\{\Pi_i^c\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into two ordering labels which attribute orderings are compatible. However, the permutation will convert E_1 and E_2 into two ordering labels with attribute orderings are compatible, which implies that $E_1 \subseteq E_2$ or $E_2 \subseteq E_1$ - contradiction.

Consider next case 2 and suppose that there exist complete permutations $\{\Pi_i^c\}_{i=1}^2$ that convert $\{L^i\}_{i=1}^2$ into two ordering labels which attribute orderings are compatible. W.l.o.g. we assume that L^1 starts with A and L^2 starts with E. Therefore, Π^c will convert L^2 into an ordering label that starts with an A and therefore $A \in E$ - contradiction.

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

Complexity of ePART merging

Recall that Step 5 of the algorithm returns a set of *simple ePARTs*, where a formal definition of a simple ePART follows.

Definition 56 (simple ePART) *An ePART is* simple *if it contains a single node without a γ-condition.*

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

Table 4.4: Three example sSQL queries

Suppose that $R_1 \subseteq R$, $R_2 \subseteq R$, $R_3 \subseteq R$, $\pi_{\mathtt{ID}} R_2 \cap \pi_{\mathtt{ID}} R_3 = \emptyset$, $\pi_{\mathtt{ID}} R_1 \cap \pi_{\mathtt{ID}} R_2 \neq \emptyset$, and $\pi_{\mathtt{ID}} R_1 \cap \pi_{\mathtt{ID}} R_3 \neq \emptyset$. Consider the ePARTs $\mathcal{P} = \langle R, \langle \{A\} \rangle \rangle$ and $\mathcal{P}_i = \langle R_i, \langle \{A, B_i\} \rangle \rangle$ for $i = 1$ to 3. If $\mathcal{P}$ and $\mathcal{P}_1$ are merged together in the ePART $\mathcal{P}^1$, then $\mathcal{P}^1$ and $\mathcal{P}_2$ will not be mergable, $\mathcal{P}^1$ and $\mathcal{P}_3$ will not be mergable, and $\mathcal{P}_2$ and $\mathcal{P}_3$ will not be mergable.

First, suppose that $\mathcal{P}^1$ and $\mathcal{P}_2$ are mergable. This implies that there exists an extended index X that can be used to efficiently answer the queries shown in Table 4.4. Suppose that X cannot be simplified further and still be used to answer the three queries efficiently. Let t be a tuple in R_1 and R_2 (recall that $\pi_{\mathtt{ID}} R_1 \cap \pi_{\mathtt{ID}} R_2 \neq \emptyset$). Then X must contain a table a node n and a table R' such that $t \in R'$ and the attributes A, B_1, and B_2 must appear in the ordering labels of the nodes in the rooted path in X that ends at n. However, from Theorem 4 it follows that either the appearance of B_1 or B_2 can be removed from the path without changing the capability of the index to be used to answer the three queries - contradiction. The ePARTs $\mathcal{P}^1$ and $\mathcal{P}_3$ cannot be merged for the same reason. Lastly, the ePARTs $\mathcal{P}_2$ and $\mathcal{P}_3$ cannot be merged because the fact that they are based on disjoint tables implies that $X^c(\mathcal{P}_2) \cap X^c(\mathcal{P}_3) = \emptyset$.

A different alternative is creating the ePART $\langle R, \langle \{A\} \rangle, [\langle R_2, \langle \{B_2\} \rangle, \langle R_3, \langle \{B_3\} \rangle]\rangle$ as the result of merging $\mathcal{P}$, $\mathcal{P}_2$ and $\mathcal{P}_3$. If the size of $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ is all the same and equal to one unit, then merging $\mathcal{P}$ with $\mathcal{P}_1$ will result in a saving of at most one unit, while the presented alternative solution (i.e. merging $\mathcal{P}$ with $\mathcal{P}_2$ and $\mathcal{P}_3$) could save two units.

Therefore, just because two ePART are mergable does not mean that merging them will produce a solution of the smallest size.

The presented example could be extended to prove that ePART merging in NP-Hard. The proof will use a reduction from the set packing problem, which is presented in [13] as follows.

Definition 57 (set packing problem) `INSTANCE` *: A collection of finite sets and a positive integer N.*
`QUESTION` *: Does the collection contain at least N mutually disjoint sets.*

Theorem 9 (ePART merging is NP-Hard) *Let $\{\mathcal{P}_i\}_{i=1}^{k}$ are k simple ePARTs. Then the problem of merging those ePARTs in such a way so that the size of the resulting ePARTs is as small as possible is NP-Hard.*

Proof: Suppose that we are given an instance of the set packing problems. Let the given sets be $\{\mathcal{M}_i\}_{i=1}^{m}$. Let $\mathcal{M}$ be the set $\bigcup\limits_{i=1}^{m} \mathcal{M}_i$ and f be a one-to-one function that maps the elements of $\mathcal{M}$ into the tuples of the table R with attributes $A, B_1, \ldots, B_m$. Let R_i be the tuples that f maps $\mathcal{M}_i$ to, where $i = 1$ to m. We next create the ePARTs $\mathcal{P} = \langle R, \langle \{A\} \rangle \rangle$ and $\{\mathcal{P}_i = \langle R_i, \langle \{A, B_i\} \rangle \rangle\}_{i=1}^{m}$. W.l.o.g. suppose that $\mathcal{P}_i$ is of size one unit for $i \in [1 \ldots m]$.

Let $\{\mathcal{P}'_i\}_{i=1}^{k}$ be the solution to the ePART merging problem for the ePARTs $\mathcal{P}$ and $\{\mathcal{P}_i\}_{i=1}^{m}$. As shown in the example preceding this theorem, the ePARTs $\{\mathcal{P}_i\}_{i=1}^{m}$ cannot be merged between themselves, i.e. $\{\mathcal{P}'_i\}_{i=1}^{k}$ consists of the ePART $\mathcal{P}'$ and a subset of the ePARTs $\{\mathcal{P}_i\}_{i=1}^{m}$. The ePARTs in the set $\{\mathcal{P}_i\}_{i=1}^{m}$ that are used in creating the ePART $\mathcal{P}'$ are based on disjoint tables and therefore if this set contains at least N elements, then the answer to the particular instance of set packing problem is yes. If this set does not contain N elements, then the answer to the particular instance of the set packing problem is no. ■

We will next show the ePART merging is in DEXPTIME by presenting an exponential algorithm that solves it. Note that this time-bound reflects the worst case time of the algorithm without accounting for the time needed to infer integrity constraints, i.e. the problem is in DEXPTIME even if reasoning with integrity constraints takes constant time.

Before presenting the ePART merging algorithm, we present several examples that shed more light on the complexity ePART merging.

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

Another possibility is to create the ePART $\mathcal{P}^1 = \langle\{\gamma\}, R_1, \langle\{A, B\}\rangle\rangle$ as the result of merging $\mathcal{P}_1$ and $\mathcal{P}_2$, where $\gamma(t)$ holds iff $t \in R_2$. The problem with this merge is that if there exists an ePART $\mathcal{P}_3 = \langle R_3, \langle\{A, C\}\rangle\rangle$ such that $R_3 \subseteq R_1$ and $\pi_{\mathtt{ID}} R_2 \cap \pi_{\mathtt{ID}} R_3 = \emptyset$, then there is not an easy way to merge $\mathcal{P}^1$ and $\mathcal{P}_3$. In particular, the γ condition from $\mathcal{P}^1$ may have to be removed and two new nodes with tables R_2 and R_3 may have to be added to the ePART.

The presented example show that there maybe more than one way of merging two ePARTs and it is important to choose the correct way. As well, the presented examples may give the impression that when merging two ePARTs no new tables need to be introduced in the resulting ePART. The example we present next shows that this is not the case.

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

The presented examples show that in solving the ePART merge problem not every two ePART that can be merged should be merged, there maybe more than one way to merge two ePARTs and selecting the right way is important, and that when two ePARTs are merged new tables in the resulting ePART may need to be introduced.

Since not any two ePARTs that are mergable are merged in the optimal solution, we use an Oracle to decide how the initial simple ePARTs are to be merged. We do so by enumerates the possible partitionings of the simple ePARTs, where we require that all the ePARTs from each partition are merged into a single ePART and no two ePARTs from distinct partitions are merged together. There are $\mathcal{B}_k$ ways to partition k elements, where

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

is the k^{th} Bell number [5]. Note that $2^k < \mathcal{B}_k < 2^{k \cdot log(k)}$, i.e. $\mathcal{B}_k$ grows exponentially.

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

If the first bit of an ePART is marked, then the table of the ePART will be in the resulting ePART. More precisely, if the simple ePARTs that have the first bit marked are based on the tables $\{\bar{R}\}$, then we will find the equivalence classes of $\{\bar{R}\}$ relative to the relationship $=_{\texttt{ID}}$ and for each equivalence class create a table for the resulting ePART that is the natural join of the tables in the equivalence class.

All the ePARTs in a partition that have the second bit marked will be used to create a set of new tables for the resulting ePART. In particular, if the simple ePARTs that have the second bit marked are based on the tables $\bar{R}$, then we will first find the equivalence classes of $\{\bar{R}\}$ relative to the relationship $=_{\texttt{ID}}$ and for each equivalence class create the natural join of the tables in the equivalence class. Let the result of the described procedure be the tables $\{R_i\}_{i=1}^{r}$. We will create all tables of the form $R_1^{j_1}$`interesection` $\ldots$ `interesection` $R_r^{j_r}$, where j_i is 1 ore -1 for $i \in [1 \ldots r]$ and $\{j_i\}_{i=1}^{r}$ are not all -1, where R^1 is the table R and R^{-1} is the set of all tuples that are not in R. Note that the so produced tables are disjoint and partition the set of the tuples in the tables $\{R_i\}_{i=1}^{r}$ into at most $2^r - 1$ parts. Note that when we apply a `union` or an `intersection` on two tables, we do not require that the tables share the same set of attributes, but the resulting table will have all the attributes from both tables. When taking the `difference` of two tables (e.g. $R_1 - R_2$), we also do not require that the two table have the same set of attributes, but the resulting table will have only the attributes of R_1. Note as well that the operations are performed based on the value of the `ID` attribute, e.g. in order to calculate the union of two tables we first calculate the duplicate eliminating union of the `ID` values from the two tables and then we add all required attributes to the resulting `ID` values.

All the ePARTs in a partition that have the third bit marked will be used to create a single new table for the resulting ePART. In particular, if the simple ePARTs that have the third bit marked are based on the tables $\{\bar{R}\}$, then we will first find the equivalence classes of $\{\bar{R}\}$ relative to the relationship $=_{\mathtt{ID}}$ and for each equivalence class create the natural join of the tables in the equivalence class. Then we will create a table for the union of the resulting tables and use it as one of the tables for the resulting ePART.

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

The pseudo code `merge_ePARTs`, that merges the ePARTs $\{\mathcal{P}_i\}_{i=1}^m$ that are from the same partition follows. The presented method returns the result of merging the ePARTs when they are mergable into an ePART $\mathcal{P}$ that contains only the tables request by the Oracle. Otherwise, the method returns `UNDEFINED`.

`merge_ePARTs(ePART` $\mathcal{P}_1, \ldots,$ `ePART` $\mathcal{P}_m$`)` $\{$
    $\bar{R}_1 = \mathtt{norm}\{R | R \in \mathtt{table}(\mathcal{P}_i)$ `for` $i \in [1..m]$ `and` $\mathcal{P}_i.\mathtt{bit}_1 = 1\}$;
    $\bar{R}_2 = \mathtt{norm}\{R | R \in \mathtt{table}(\mathcal{P}_i)$ `for` $i \in [1..m]$ `and` $\mathcal{P}_i.\mathtt{bit}_2 = 1\}$;
    $\bar{R}_3 = \mathtt{norm}\{R | R \in \mathtt{table}(\mathcal{P}_i)$ `for` $i \in [1..m]$ `and` $\mathcal{P}_i.\mathtt{bit}_3 = 1\}$;
    $\bar{R} = \bar{R}_1 \cup \mathtt{intersections}(\bar{R}_2) \cup \mathtt{union}(\bar{R}_3)$;
    `if` $(\mathtt{build\_tree}(\bar{R}) == \mathtt{UNDEFINED})$ `return UNDEFINED;`
    $G^t = \mathtt{build\_tree}(\bar{R})$;
    $\langle n_1, \ldots, n_r \rangle =$ `breath first traversal of` G^t;
    `for` $(i = 1; i \leq r; i++)$ $\{$
      $\bar{\mathcal{P}}_i = \{\mathcal{P}_j |\ j \in [1..m]$, $\mathcal{P}_j.\mathtt{bit}_1 = 1$ `or` $\mathcal{P}_j.\mathtt{bit}_2 = 1$, `and` $\mathcal{P}_j$ `contribute to the creation of` $\mathtt{table}(n_i)$ $\}$;
      $L_i = \bigoplus_{L \in \mathcal{L}(\mathcal{P}), \mathcal{P} \in \bar{\mathcal{P}}_i} L$;
      $\mathcal{P}_i = \langle \mathtt{table}(n_i), \langle L_i \rangle \rangle$;
      `if` $(i == 1)$ $\mathcal{P} = \mathcal{P}_1$ `else` $\mathcal{P} = \mathtt{table\_ePART\_merge}(\mathcal{P}, \mathcal{P}_i)$;
      `if` $(\mathcal{P} == \mathtt{UNDEFINED})$ `return UNDEFINED;`
    $\}$
    $\bar{\mathcal{P}} = \{\mathcal{P}_j \mid j \in [1 \ldots m]$ `and` $\mathcal{P}_j.\mathtt{bit}_1 = 0$ `and` $\mathcal{P}_j.\mathtt{bit}_2 = 0\}$
    `for` $\mathcal{P}' \in \bar{\mathcal{P}}$ $\{$
      $\mathcal{P} = \mathtt{gamma\_ePART\_merge}(\mathcal{P}, \mathcal{P}')$;

```
(18)      if (𝒫 == UNDEFINED) return UNDEFINED;
(19)    }
(20)    return 𝒫;
(21) }
```

The method $\texttt{norm}(\bar{R})$ constructs a single table from each equivalence class relative to relationship $=_{\texttt{ID}}$. It does so by doing the natural join of the tables in each equivalence class. The method $\texttt{intersections}(\bar{R})$ returns the result of applying all possible boolean operations on the tables $\bar{R}$ except for the boolean operation that computes the union of all tables in the set $\bar{R}$. Note that the operations are done on the projection on the `ID` attributes and the remaining attributes are added later. For example, if the union or the intersection of R_1 and R_2 is performed, then the resulting table will have the attributes of R_1 and R_2. On the other hand, if $R_1 - R_2$ is performed, then the resulting table will have only the attributes from R_1. The method $\texttt{build\_tree}(\bar{R})$ builds a node labeled tree with labels the tables in the set $\bar{R}$. The tree has the properties:

1. $\texttt{table}(n_1) \subseteq_{\texttt{ID}} \texttt{table}(n_2)$ iff n_1 is an ancestor of n_2, and

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

When such a tree does not exist, then the `build_tree` method returns `UNDEFINED`. Note that we use node functions as `table` and $\mathcal{L}$ on simple ePARTs because those ePARTs contain a single node.

The `merge_ePARTs` method first create a tree from the suggested by the two Oracles tables (Lines 1-6 of the method). The created tree will be the tree for the resulting ePART, and when this tree does not have the desired properties to be the tree for a valid ePART, then an exception is raised. We start with an empty ePART and we do a breath first traversal of the created tree. If an ePART has its 1^{st} bit marked, then we use the method `table_ePART_merge` to add it to the resulting ePART, where this merge will involve the introduction of a new node to the ePART and possibly modifying existing ordering labels. Similarly, if an ePART has its 2^{nd} bit marked, then we use the method `table_ePART_merge` to add it to the resulting ePART in the same way. Note that all the ePART that have their $3^r d$ bit marked, if any, will contribute to creating the table for the root node of the ePART and therefore there labels should be created after the original population of the tree with ordering labels. Then all the ePARTs that have note been added to the resulting ePART are added to it using the `gamma_ePART_merge` method. Note that those are the ePARTs that do not have their first 2 bits marked. The `gamma_ePART_merge` is performed in a such a way that no new tables are added to the current ePART, the existing ePART tables are not modified, and only new γ conditions are added or existing ordering labels are modified.

The `table_ePART_merge`$(\mathcal{P}_1, \mathcal{P}_2)$ method merges an arbitrary ePART $\mathcal{P}_1$ with a simple ePART $\mathcal{P}_2 = \langle R, \langle L \rangle\rangle$. The pre-condition for executing this method is the existence of a node n_k in $\mathcal{P}_1$ with table R_k such that $R \subset_{\mathtt{ID}} R_k$ and for every R' that is a table of a child node of n in $\mathcal{P}_1$ it is the case that $\pi_{\mathtt{ID}} R' \cap \pi_{\mathtt{ID}} R = \emptyset$. The method adds a new node with table R to $\mathcal{P}_1$ when possible and returns `UNDEFINED` otherwise. If the insertion can be done in such a way so that the label of the inserted node is empty, then the method also returns `UNDEFINED`. The reason is that the created ePART will be equivalent to an ePART without the inserted node in which a γ condition is added to the node's parent in the ePART. W.l.o.g. assume that $\langle n_1, \ldots, n_k\rangle$ is the rooted path in $\mathcal{P}_1$ that ends at n_k. Let $n_i = \langle \bar{\gamma}_i, R_i, \langle L_i \rangle\rangle$ for $i = 1$ to k. The pseudo-code for `table_ePART_merge` method follows.

ePART table_ePART_merge(ePART $\mathcal{P}_1$, ePART $\mathcal{P}_2$)
    compute $n_i = \langle \bar{\gamma}_i, R_i, \langle L_i \rangle\rangle$, $i = 1$ to k;
    $\mathcal{P}_2 = \langle R, \langle L \rangle\rangle$;
    $L' = L_1 \circ \cdots \circ L_k$;
    if $(L \oplus L' ==$ UNDEFINED$)$ return UNDEFINED;
    $L'' = L \oplus L'$;
    γ' : $\gamma'(t)$ holds iff $t \in R$;
    $L'' = L'_1 \circ \cdots \circ L'_k \circ L_{k+1}$, where $|L'_i| = |L_i|$ for $i = 1$ to k;
    if $(|L| \leq |L'|)$ return UNDEFINED;
    $L = L^1 \circ L^2$, where $|L^1| = |L'|$;
    if ((is_E_component(L^1) || $(|L^1| == 0))$
      $\gamma =$ TRUE;
    else
      $\gamma = \gamma'$;
    return substitute$(\mathcal{P}_1, n_1 = \langle \bar{\gamma}_1 \cup \gamma, R_1, \langle L'_1 \rangle\rangle, n_2 = \langle \bar{\gamma}_2, R_2, \langle L'_2 \rangle\rangle, \ldots,$

$n_k = \langle \bar{\gamma}_k, R_k, \langle L'_k \rangle, [\langle \texttt{add\_gamma}(R + \texttt{Attr}(R_k), R_k), \langle L_{k+1} \rangle \rangle] \rangle$);

(15) }

The referenced substitute($\mathcal{P}, n_1 = \mathcal{P}_1, \ldots, n_k = \mathcal{P}_k$) method returns the result of substituting the node n_i in $\mathcal{P}$ with the tree for $\mathcal{P}_i$, $i \in [1..k]$. In order for this method to be well defined, it must be the case that for $i \in [1..k]$ either n_i is a leaf node in the description tree of $\mathcal{P}$ or the tree of $\mathcal{P}_i$ is in the form of a directed path. The referenced is_E_component(L) method returns TRUE iff L is an E-component. The referenced add_gamma(R_1, R_2) returns R_1 to which an additional binary attribute that shows whether for a tuple $t \in R_2$, $\pi_{\texttt{ID}} t \in R_1$ also holds. Such an attribute is only added when this cannot be determined from the existing attributes of R_1 in constant time. In the code we have used $R_1 + Attr(R_2)$ to denote the table R_1 in which the attributes of R_2 are added to it. This operation is well defined exactly when $R_1 \subseteq_{\texttt{ID}} R_2$.

The table_ePART_merge method add a new node in $\mathcal{P}_1$ with table R_1. When the conditions in Line 10 of the pseudo-code is TRUE, then it is the case that the rooted path in the new ePART that ends at the inserted node, which we will denote as K' is clustered relative to the integer l, where l is the position of the last attribute in L that is part of the E-component of L. In this case a γ-condition does not need to be added to the new ePART because the artificial indices it represents will be able to efficiently answer the query that generated $\mathcal{P}_2$ without the γ-condition. The labels along the path K' are also modified to reflect the result of applying the "$\oplus$" operation between the ordering label constructed from K with the ordering label L.

The pseudo-code for the gamma_ePART_merge method follows. The code merges an arbitrary ePART $\mathcal{P}_1$ with a simple ePART $\mathcal{P}_2$. The code tries to do so without introducing new nodes in $\mathcal{P}_1$ and without changing the tables of the nodes in $\mathcal{P}_1$. If this can not be done, then the method returns UNDEFINED.

(00) ePART $\mathcal{P}$;

(00) set of Tables $\bar{R} = \emptyset$;

(00) gamma_ePART_merge(ePART $\mathcal{P}_1$, ePART $\mathcal{P}_2$) {

(01) $\mathcal{P}_2 = \langle R, \langle L \rangle \rangle$;

```
    $L = \{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a$;
    Boolean $mergable$=TRUE;
    $n_1 = \texttt{root}(\mathcal{P}_1) = \langle \bar{\gamma}_1, R_1, \langle L_1 \rangle \rangle$;
    if !$(\pi_{\texttt{ID}} R \subseteq \pi_{\texttt{ID}} R_1)$ return UNDEFINED;
    if $(L \oplus L_1 ==$ UNDEFINED$)$ return UNDEFINED;
    $L' = L \oplus L_1$;
    if $((\texttt{is\_leaf}(n_1))\ ||\ (|L| \leq |L_1|))$ {
      if $\langle L' \rangle$ is clustered relative to $l$
        $\gamma =$ TRUE;
      else
        $\gamma$ :  $\gamma(t)$ iff $t \in R$;
      $\mathcal{P} = \texttt{substitute}(\mathcal{P}_1, n_1 = \langle \bar{\gamma}_1 \cup \{\gamma\}, R_1 + \texttt{Attr}(R), \langle L' \rangle \rangle)$;
      return $\mathcal{P}$;
    }
    $L' = L^1 \circ L^2$ where $|L^1| = |L_1|$;
    $\mathcal{P} = \texttt{substitute}(\mathcal{P}_1, n_1 = \langle \bar{\gamma}_1, R_1, \langle L^1 \rangle \rangle)$;
    for $n' \in \texttt{children}(n_1, \mathcal{P}_1)$
      $mergable = mergable \wedge \texttt{recursive\_ePART\_merge}(n', R, L^2, l)$;
    if $((mergable)$ && $(\pi_{\texttt{ID}} R \subseteq \bigcup_{R' \in \bar{R}} \pi_{\texttt{ID}} R'))$ return $\mathcal{P}$ else return UNDEFINED;
}
```

```
Boolean recursive_ePART_merge(node $n$, table $R$, label $L$, int $l$){
    $n = \langle R_1, \langle L_1 \rangle \rangle$;
    if $(\pi_{\texttt{ID}} R \cap \pi_{\texttt{ID}} R_1 == \emptyset)$ return FALSE;
    Boolean $mergable$=TRUE;
    $\gamma$ :  $\gamma(t)$ iff $t \in R$;
    if $(L \oplus L_1 ==$ UNDEFINED$)$ return UNDEFINED;
    $L' = L \oplus L_1$;
    if $((\texttt{is\_leaf}(n))\ ||\ (|L| \leq |L_1|))$ {
      $\mathcal{P} = \texttt{substitute}(\mathcal{P}, n = \langle R_1 + \texttt{Attr}(R), \langle L' \rangle \rangle)$;
      $\bar{R} = \bar{R} \cup R_1$;
```

(10) `Let` K `be the rooted path in` $\mathcal{P}$ `that ends at` n`;`
(11) `Let` n_1 `the root node of` $\mathcal{P}$`;`
(12) `if` K `is not clustered relative to` l
(13) $\mathcal{P} = \texttt{substitute}(\mathcal{P}, n_1 = \langle \gamma(n_1) \cup \{\gamma\}, \texttt{table}(n_1), \mathcal{L}(n_1)\rangle)$`;`
(14) $\}$
(15) `else` $\{$
(16) $L' = L^1 \circ L^2$ `where` $|L^1| = |L_1|$`;`
(17) $\mathcal{P} = \texttt{substitute}(\mathcal{P}_1, n = \langle R_1, \langle L^1 \rangle\rangle)$`;`
(18) `for` $n' \in \texttt{children}(n)$
(19) $mergable = mergable \wedge \texttt{recursive\_ePART\_merge}(n', R, L^2, l)$`;`
(20) $\}$
(21) `return` $mergable$`;`
(22) $\}$

The `gamma_ePART_merge` method tries to merge $\mathcal{P}_1$ with the simple ePART $\mathcal{P}_2 = \langle R, \langle L \rangle\rangle$ without creating a new nodes in $\mathcal{P}_1$. In particular it scans the tree of $\mathcal{P}_1$ in a depth first search manner, where the non-root nodes of subtree are pruned out when the root n is such that $|L| \leq |L_1|$ (see Line 6 of the `recursive_ePART_merge` procedure. The resulting ePART that is constructed throughout the method invocation is stored in $\mathcal{P}$. The *mergable* variable is used to denote whether a condition that will be prevent the two ePARTs from being mergable is reached. In particular, such a condition will be when a node n is reached during the traversal of $\mathcal{P}_1$ that has a table R_1 and $\pi_{\texttt{ID}} R_1 \cap \pi_{\texttt{ID}} R = \emptyset$.

We will next prove a theorem that shows that the presented ePART merging algorithm is correct. The theorem uses the following definition.

Definition 58 (simplifying an extended index) *Let X be an extended index. X can be simplified by performing one or more of the following rules one or more times.*

- *a table R of a node in X converted into a table R' s.t. $Attr(R') \subset Attr(R)$ or $\pi_{\texttt{ID}} R' \subseteq \pi_{\texttt{ID}} R$,*
- *a γ condition removed from X,*

- *an ordering label L of a node of X being simplified by deleting an attribute from its end,*
- *a leaf node of X being removed.*

Note the if an extended index X is simplified into an extended X', then the physical encoding of X will be greater or equal to that of X'.

Definition 59 (similar extended indices) *We will say that the extended index X is similar to the extended index X' iff X can be converted into X' by applying on or more times the following procedure: If $\bigcup_{i=1}^{k} R_i = R$, then split a non-root node in X with label $\langle R, \langle L \rangle\rangle$ into k nodes with tables $\{R_i\}_{i=1}^{k}$, ordering label L, and same parent and children.*

Theorem 10 *1. Suppose that the sSQL queries $\{Q_i\}_{i=1}^{k}$ generate the simple ePARTs $\{\mathcal{P}\}_{i=1}^{k}$ and for some of the possibilities generated by the second Oracle,* `merge_ePARTs` *produces an answer different than* `UNDEFINED`*. Let $\bar{\mathcal{P}}$ be the set of ePARTs (i.e. answers different than* `NULL`*) produced by* `merge_ePARTs` *under the different possibilities suggest by the second Oracle, then:*

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

1.2 $\bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$ is not empty.

1.3 In presence of only disjointness, containment, and covering integrity constraint conditions, if X can be used to efficiently answer the queries $\{Q_i\}_{i=1}^{k}$ and all extended indices X' that are simplified from X cannot be used to efficiently answer the enumerated queries, then X is similar to some extended index X' for which there exists $\mathcal{P} \in \bar{\mathcal{P}}$ such that $X' \in X(P)$.

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

Proof: 1. Suppose that $\{\mathcal{P}\}_{i=1}^{k}$ are simple ePARTs and $\bigcap_{i=1}^{k} X^c(\mathcal{P}_i) \neq \emptyset$.

1.1 We will first prove that if $\mathcal{P} \in \bar{\mathcal{P}}$, then $X^c(\mathcal{P}) = \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$.

$\Leftarrow$ Suppose that $X \in \bigcap_{i=1}^{k} X^c(\mathcal{P}_i)$, i.e $X \in X^c(\mathcal{P}_i)$ for all $i \in [1..k]$. Let $\bar{X}_i$ be the set of extended indices that $\mathcal{P}_i$ convert to, i.e. $\bar{X}_i = X(\mathcal{P}_i)$ for $i = 1$ to k. Therefore, it must be the case that $X \in \texttt{closure}(\bar{X}_i)$ for $i = 1$ to k.

Consider the ePART $\mathcal{P}_i$ for some $i \in [1..k]$. Note that $\mathcal{P}_i$ will have the form $\langle R, \langle \{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a \rangle\rangle$. Since $X \in \texttt{closure}(\bar{X}_i)$, there must exist $X_i \in \bar{X}_i$ $(i \in [1 \ldots k])$ such that $Q(X_i) \subseteq Q(X)$ for $i \in [1 \ldots k]$. Note that $X_i = \langle R, \langle A'_1, \ldots, A'_l, A_{l+1}, \ldots, A_a, \texttt{ID}\rangle\rangle$, where $\langle A'_1, \ldots, A'_l\rangle$ is a permutation of the attributes $\{A_i\}_{i=1}^{l}$. Let $\bar{X}$ be the set of extended indices in $\mathcal{P}$ that respect the permutation decisions of $\{X_{i=1}\}^k$. In other words, for $i \in [1..k]$, if the attribute A comes before the attribute B in the concatenation of the ordering labels along a rooted path K in X_i, then the attribute A comes before the attribute B in the rooted path(s) that K maps to in the extended indices in $\bar{X}$. Note that the `table_ePART_merge` and `gamma_ePART_merge` methods are such that $P_i = \langle R, \langle \{A_1, \ldots, A_l\}, A_{l+1}, \ldots, A_a\rangle\rangle$ maps to one or more rooted paths in $\mathcal{P}$, which we will denote as $\bar{K}_i$ that have the following properties:

1. If the end nodes of the paths $\bar{K}_i$ are the tables $\bar{n}_i$, then:
   (a) $\pi_{\texttt{ID}} R = \bigcup_{R' \in \gamma(table(n))}^{n \in \bar{n}_i, \gamma \in \gamma^e(n)} \pi_{\texttt{ID}} R'$.
   (b) If $n \in \bar{n}_i$ and $R' \in \texttt{table}(n)$, then $Attr(R) \subseteq Attr(R')$
2. The concatenation of the attributes of the ordering labels of each path $\bar{K}_i$ starts with a permutation of the attributes $\{A_i\}_{i=1}^{l}$ followed by the attributes $\langle A_{l+1}, \ldots, A_a\rangle$ in this order.
3. Each path in the set $\bar{K}_i$ is clustered relative to l.

This implies that for $X' \in \bar{X}$ $Q(X') \subseteq Q(X)$ and therefore $X \in \texttt{closure}(\bar{X})$, which implies that $X \in X^c(\mathcal{P})$.

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

1. If the end nodes of the paths $\bar{K}_i$ are the tables $\bar{n}_i$, then:
   (a) $\pi_{\texttt{ID}} R = \bigcup_{R' \in \gamma(table(n))}^{n \in \bar{n}_i, \gamma \in \gamma^e(n)} \pi_{\texttt{ID}} R'$.
   (b) If $n \in \bar{n}_i$ and $R' \in \texttt{table}(n)$, then $Attr(R) \subseteq Attr(R')$
2. The concatenation of the attributes of the ordering labels of each path $\bar{K}_i$ starts with a permutation of the attributes $\{A_i\}_{i=1}^{l}$ followed by the attributes $\langle A_{l+1}, \ldots, A_a \rangle$ in this order.
3. Each path in the set $\bar{K}_i$ is clustered relative to l.

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

1.2 Direct consequence of 1.1.

1.3 Suppose that the extended index X can be used to efficiently answer the queries $\{Q_i\}_{i=1}^{k}$ and that X cannot be simplified further and still be used to efficiently answer those queries. Let the query Q_i translate into the ePART $\mathcal{P}_i = \langle R_i, A_1, \ldots, A_l, A_{l+1}, \ldots, A_a \rangle$ for $i \in [1 \ldots k]$. Therefore, from Theorem 2 it follows that X must contain the nodes $\bar{n}_i$ that have the following properties.

1. If the end nodes of the paths $\bar{K}_i$ are the tables $\bar{n}_i$, then:
   (a) $\pi_{\texttt{ID}} R = \bigcup_{R' \in \gamma(table(n))}^{n \in \bar{n}_i, \gamma \in \gamma^e(n)} \pi_{\texttt{ID}} R'$.
   (b) If $n \in \bar{n}_i$ and $R' \in \texttt{table}(n)$, then $Attr(R) \subseteq Attr(R')$

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

3. Each path in the set $\bar{K}_i$ is clustered relative to l.

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

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

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

1. on tables that are referenced in the set of queries $\{Q_i\}_{i=1}^{k}$ or

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

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

Note that if a table R is created by the `table_ePART_merge`, it must be the case that the ordering label of the node for R will not be empty. W.l.o.g. suppose that $\mathcal{P}$ can not be rewritten to an equivalent ePART $\mathcal{P}'$ in which the table of the node for R will be empty or not present. If this could not be done, then another possibility suggested by the second Oracle will generate the described ePART. Recall that our algorithm for creating $\mathcal{P}$ fills the ordering labels of $\mathcal{P}$ in a breath first traversal of the tree and for every ePART in the set $\{\mathcal{P}_i\}_{i=1}^k$ that has its first bit marked a node is created and for the subset of ePARTs in the set $\{\mathcal{P}_i\}_{i=1}^k$ that have their second bit marked a set of nodes is created. Moreover, the algorithm succeeds exactly when the ordering labels of the enumerated ePARTs extend to the inserted node. The ordering labels generated by the ePARTs that have their bit marked can be added last because they are defined on the table of the node of the root of the tree. However, since X'' cannot be simplified further, the ordering labels in X'' will be the same as the ordering label of $\mathcal{P}$, i.e. we have shown that $X'' \in X(\mathcal{P})$.

2. Suppose that $\bigcap\limits_{i=1}^{k} X^c(\mathcal{P}_i) \neq \emptyset$. Therefore, there exists an extended index X s.t. $X \in \bigcap\limits_{i=1}^{k} X^c(\mathcal{P}_i)$. We will show that under at least one permutation suggested by the second Oracle `merge_ePARTS` produces an ePART $\mathcal{P}$ such that $X \in X^c(\mathcal{P})$. W.l.o.g. assume that X cannot be simplified further and consider the tables of the nodes of the description tree of X. If X does not reference a set of tables that could be generated by a possibility suggested by the second Oracle, then X can be rewritten into a similar extended index X' that references a set of tables suggested by the second Oracle, where this procedure will involve several node break-ups. Therefore, suppose that X references the tables suggested by one of the possibilities of the second Oracle. Under this possibility the `merge_ePARTs` method will generate an ePART $\mathcal{P}$ such that $X \in X(\mathcal{P})$, i.e. under at least one possibility suggested by the second Oracle `merge_ePARTS` will not generate `UNDEFINED`. ■

Approximate Algorithms for merging ePARTs

In the previous section we showed an exponential algorithm for merging ePARTs. A drawback of the presented algorithm is that it takes exponential time for any input. In this section we present an approximate algorithm for merging ePARTs. This algorithm does not always produce the smallest possible set of ePARTs, but has the advantage that it has worst case polynomial running time. An area for future research is extending the presented approximate polynomial time algorithm to an exact exponential time algorithm that has polynomial best case running time.

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

We will refer to nodes in the graph with no incoming directed edges to them as root nodes. For a node n, its children nodes are the nodes for which there is a directed edge from n to them. For the first root node, relative to some order, we will exam all the nodes' children and merge as much of them as possible with the node using the `table_ePART_merge` procedure. In particular, if the node contains children with undirected edges between them and ordering labels of the children that can be merged using the $\oplus$ operation, then the method splits the nodes using the `intersections` method when this will help for the nodes to be merged with the root node. The procedure is repeated recursively for the children of the children's nodes and so on until nodes with no outgoing arcs are reached. After the described procedure is applied for the first root note, the nodes that contributed to the created ePART are removed from the graph and the edges between the nodes of the graph are recalculated using the same procedure as the one described in the previous paragraph. Then the described method is applied on one of the root nodes in the new graph and so on. After all nodes have been removed from the graph, we examine all created ePARTs that were created from a single original ePART. We then try to add them to the other ePARTs using the `gamma_ePART_merge`.

The presented algorithm runs in polynomial time, but it does not guarantee that the it will produce the optimal solution. It is just one example of how an approximate algorithm for the ePART merging problem can be created. Other avenues for future research that we believe are worth following include designing approximate algorithms for the ePART merging problem using dynamic programming and iterative dynamic programming ([18]).

4.2.8 Step 8 - Semantic ePART Simplification

This step involves semantic ePART simplification.

Exploiting Integrity Constraints

One way an ePART can be semantically simplified is by using the ideas suggested in [8]. In particular, we use the defined functional dependencies in the database schema, if any, to simplify the ordering conditions in the current set of ePARTs.

Consider our example schema from Figure 1.5 and the ePART $\langle \texttt{EMP}, \langle age, salary \rangle\rangle$. If we know that the functional dependency $\texttt{EMP}(age \rightarrow salary)$ holds, then we can rewrite the ePART as $\langle \texttt{EMP}, \langle age \rangle\rangle$. The reason is that we know that *age* functionally determines *salary*, i.e. once we fix the value for the attribute *age*, there will be only one distinct salary with that age and therefore there will be no need to search for a particular *salary* within the tuples with the same age.

Our algorithm will first examine the E-components of the ordering labels of each of the ePARTs produced by Step 6 of the algorithm. In particular, every E-component that contains an attribute that can be functionally determined by other attributes of the ordering label will be removed from the E-component.

When simplifying an ePART $\mathcal{P}$ produced by Step 6 of the algorithm, we will next examine the leaf nodes of $\mathcal{P}$. In particular, for each leaf node n of $\mathcal{P}$ we will examine the attributes in $\mathcal{L}^{\uparrow}(n)$ in order of their appearance. In particular, if an attribute appears after some attributes that functionally determine it, then remove the attribute from the ePART in the context in which it appears in $\mathcal{L}^{\uparrow}(n)$.

Note that this procedure is deterministic and can be performed in linear time because we know that the defined integrity constraints are acyclic. However, the query plans for the μSQL queries that use the simplified ePART and reference the deleted attribute will need to be modified.

Theorem 11 *Suppose that the ePART $\mathcal{P}_1$ was rewritten into the ePART $\mathcal{P}_2$ by the presented algorithm. Then each query Q that was used in the creation of $\mathcal{P}_1$ could be efficiently answered using any of the extended indices in $X(\mathcal{P}_2)$.*

Proof: First, consider a sSQL Q of type 1 or 2 (see Figure 4.3) that was used to construct the ePART $\mathcal{P}_1$ and w.l.o.g. assume that the ePART contains a node n with table R' and an extended γ-condition s.t. $\gamma(R') = \mathtt{ID}R$ (note that R is the table referenced by the query). Suppose that K is the rooted path that ends at n. It will be the case that $\langle A_1, \ldots, A_a$ will be the first l attributes in $\mathcal{L}^{\uparrow}(n)$, where only the ordering inside E components could be changed.

Suppose that the attribute A_i is removed from K and $i \in [1 \ldots l]$. We will next examine two cases: when A_i is part of an F-component and when A_i is part of an E-component. Suppose that X is an extended index for $\mathcal{P}_2$. The query Q can then be efficiently answered using the extended index X by substitute the search for a record with a particular value for A_i after having found a record with particular values for $\{A_j\}_{j=1}^{i-1}$ with a check of whether the value of A_i of the current record is equal to P_i. If it is, then the search will be continue, if not - the search will be aborted. The reason we can do this substitution is that the defined functional dependencies guarantees that if $A_i = P_i$ for the current record, then $A_i = P_i$ for all records with the same value for the attributes $\{A_j\}_{j=1}^{i-1}$.

Next, consider the case when i is still in the range $[1 \ldots l]$, but A_i is part of an E-component. Suppose that X is an extended index for $\mathcal{P}_2$. The query Q can then be efficiently answered using the extended index X by first performing the search for the other attributes from the E-component from which A_i was removed and then substituting the search for a record with a particular value for A_i after having found a record with particular values for the other attributes in the E-component and the attributes before that with a check of whether the value of A_i of the current record is equal to P_i. If it is, then the search will be continue, if not - the search will be aborted.

Next, consider the case when i is in the range $[l{+}1 \ldots a]$. We will examine two subcases: when A_i is an ordering attribute in Q and when a range condition is specified on A_i in Q.

If A_i is an ordering attribute, and X is an extended index for $\mathcal{P}_2$, then Q can be answered using X by skipping the ordering relative to the attribute A_i. This will not change the query result because the known functional dependency guarantees that if $A_i = P_i$ for the current record, then $A_i = P_i$ for all records with the same value for the attributes $\{A_j\}_{j=1}^{i-1}$.

Lastly, suppose that $i \in [l+1 \ldots a]$ and A_i is the attribute in Q on which a range condition is applied, which will imply that $i = l+1$. Again, let X be an extended index for $\mathcal{P}_2$. Q can be answered using X by substituted the search for the records in which the attribute A_{l+1} is in some particular range with the check of whether the current record (i.e. the one for which $A_j = P_j$ for $j = 1$ to l) has a value of the attribute A_{l+1} in the desired range. If it does, then the search will continue; if it does not, then the search will fail.

Next, consider the case when Q is a sSQL query of type 3. Since the performed modification does not change the tables of the ePART, the result is trivial. ■

Simplifying γ-conditions

Consider the ePART $\mathcal{P}$. This step will remove redundant γ-conditions from the nodes of $\mathcal{P}$. In particular, if a node in $\mathcal{P}$ contains a γ-condition that can be removed without changing the extended γ-condition of the node, then remove that γ condition. Theorem 4 shows that this will not change the set of queries that the extended indices produced from the ePART would be able to efficiently answer.

4.2.9 Step 9 - Converting PARTs into Physical Designs

In this step we convert each of the ePARTs produced by the previous step of the algorithm into extended indices. In particular, for each ePART $\mathcal{P}$ we consider the set of extended indices $X(\mathcal{P})$ and from it we choose the smallest extended index. Note that Step 6 of the algorithm produces a set of ePARTs, where there maybe one or more alternative for each ePART. In order to decide which ePART to chose our algorithm will apply Step 8 to each ePART, then choose the smallest extended index for the rewritten ePARTs, and then create the extended index that is the smallest of all suggested indices. Note that although the suggested procedure takes exponential time, it produces the smallest possible solution.

Chapter 5

Finding an Encoding with Compact Structures

5.1 cPARTs

Recall that we defined the abstraction ePART for extended indices (see Section 2.2.1). Similarly, we define the concept of a *compact structure Parameterized Access Requirement Type* (cPART) for compact structures. We define the syntax of a cPART to be exactly the same as the syntax of a compact structure, where a cPART is valid when the compact structure with the same syntax is valid. Unlike ePARTs, we define the semantic of a cPART to be a single compact structure, i.e. each cPART represents exactly the compact structure that has the same syntax. We will use $J(\mathcal{P})$ to denote the compact structure for the cPART $\mathcal{P}$. If $\mathcal{P}$ is a cPART, we will use $Q(\mathcal{P})$ to denote the set of queries that the corresponding compact structure can be used to efficiently answer. Note that since a compact structure and a cPART represent the same physical design, from now on we will use the two terms interchangeably.

Definition 60 (cover of a cPART) *Let $\mathcal{P}$ be a cPART. We define* $\texttt{cover}(\mathcal{P})$ *as the set* $\{\mathcal{P}' \mid Q(\mathcal{P}) \subseteq Q(\mathcal{P}')\}$*.*

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

Table 5.1: The cPARTs for the three sSQL query types

5.2 The Solution Steps

Steps 1-4 are the same for ePART and cPARTs. However, steps 5-9 are different.

5.2.1 Step 5 - Converting sSQL Queries into cPARTs

In this step we convert each sSQL created by the previous step of the algorithm into a cPART. Table 5.1 shows how the simple queries of the three types (see Table 4.3) are converted into cPARTs.

Theorem 12 (correctness of cPART creation) *If the query Q produced the cPART $\mathcal{P}$, then* $\texttt{closure}(\mathcal{P})$ *represents exactly the set compact structures that can be used to efficiently answer the query Q.*

Proof: $\Leftarrow$ From Theorem 5.2.4 it follows that $Q \in Q(\mathcal{P})$ and from Definition 60 it follows that for every $\mathcal{P}' \in \texttt{closure}(\mathcal{P})$, $Q \in Q(\mathcal{P}')$.

$\Rightarrow$ We will show that if $Q \in Q(\mathcal{P}')$, then $\mathcal{P}' \in \texttt{closure}(\mathcal{P})$.

First, suppose that Q is an sSQL query of type 1 or 2 (see Figure 4.3). Since $Q \in Q(\mathcal{P}')$, from Theorem 5.2.4 it follows that $\mathcal{P}'$ must contain a complete path $K = \langle n_1, \ldots, n_{k+1}\rangle$ that has the properties described in Table 2.10. Therefore, every query in the set $Q(\mathcal{P})$ is in the set $Q_K(\mathcal{P}')$ and from Theorem 5.2.4 and definition 60 it follows that $\mathcal{P}' \in \texttt{closure}(\mathcal{P})$.

Second, suppose that Q is an sSQL query of type 3. From the fact that $Q \in Q(\mathcal{P}')$ it follows that $\mathcal{P}'$ must contain a leaf node n with table R and stored attributes that include the attributes $\{D_i\}_{i=1}^{d}$. Therefore, every query in the set $Q(\mathcal{P})$ is in the set $Q(\mathcal{P}')$ and from Definition 60 it follows that $\mathcal{P}' \in \texttt{closure}(\mathcal{P})$. $\blacksquare$

5.2.2 Step 6 - Making cPARTs Efficiently Maintainable

In this step we modify the cPARTs created in the previous step of the algorithm in such a way so that the defined primitive updates can be efficiently propagated to them. The reason this step is require is because not all type of primitive updates can be performed on a compact structure efficiently (see 6).

In particular, in order for a cPART to be efficiently maintainable, we require that:

1. If $\mathcal{P}$ is a cPART and a n is a leaf node in $\mathcal{P}$ with table R and an insertion can affect R, then n should not contain an ordering condition in its leaf nodes.

2. If $\mathcal{P}$ is a cPART and a n is a leaf node in $\mathcal{P}$ with table R and a deletion can affect R, then the type of n should be `dll`.

3. If $\mathcal{P}$ is a cPART and a n is a leaf node in $\mathcal{P}$ with table R and a modification 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 sequence of insertions and deletions.

Consider a cPART $\mathcal{P}$ and a node n of the cPART with table R. If a primitive insertion is defined on R, we will move the ordering condition from n and append it to the ordering condition or the parent node in $\mathcal{P}$. If a parent node does not exist, we will create a parent node of type extended index with table R and again move the ordering condition from n to the parent node. On the other hand, if a primitive insertion is defined on R, all we need to do is change the type of n to `dll` if it is `ll`. If a modification is defined on R and the modification does not preserve the order defined by the ordering condition of n, then we will rewrite the cPART to allow primitive insertions and deletions to the table R to be performed.

Definition 61 (simple cPARTs) *Simple cPARTs are those cPARTs that are have the* *syntax of cPARTs created from sSQL queries as shown in Table 5.1 and then made efficiently maintainable using the procedure suggested in this section.*

Note that simple cPARTs reference the same table in all the nodes, contain at most 3 nodes that form a path and are in the sequence: hash node, index node, and leaf node, and the index node has no γ-condition.

5.2.3 Step 7 - cPART Merging

The cPART merging problem is to merge a set of cPARTs in such a way so that the result is as small as possible. Consider a set of cPARTs that have the property that their root node contains a non-empty ordering label or ordering condition. These cPARTs could be merged into a single cPART. This can be done by first breaking every hash node $\langle R, \{A_1, \ldots, A_a\}$ into a sequence of a hash node, where the $i^{\mathtt{th}}$ hash node has the label $\langle R, \{A_i\}\rangle$. Then, for example, we can build an index on the distinct values of the first attributes in the ordering label or ordering condition in the root node of the cPARTs. Although the presented merge is valid, it is unlikely to save space. The reason is the values for different attributes are usually disjoint (e.g. it is the unlikely that an employee has a name that coincides with the salary of another employee). This is why, given a set of cPARTs that have a non-empty ordering label or an ordering condition in their root nodes, we will consider merging them exactly when all their root nodes contain an ordering label or an ordering condition that start with the same attribute or their root node is a hash node that contains this attribute.

Assumptions

Our algorithm for merging cPARTs uses the following assumptions to compare cPART sizes.

1. The hash structure $\langle R, \{A_1, \ldots, A_a\}\rangle$ takes less space than the extended index $\langle R, \langle A_1, \ldots, A_a\rangle\rangle$ for any value of R and $\{A_i\}_{i=1}^{a}$.
2. The compact structure $\langle R, \langle A_1\rangle, [\langle R, \langle A_2\rangle, [\langle R, \bar{C}, \langle\rangle, \mathit{type}\rangle]\rangle]\rangle$ takes less space than the compact structure $\langle R, \langle A_1, A_2\rangle, [\langle R, \bar{C}, \langle\rangle, \mathit{type}\rangle]\rangle$ for any value of R, A_1, A_2, $\bar{C}$, and *type*.

The result of the first assumption is that we will choose a hash structure over an extended index when possible. Since a hash structure is usually faster than an extended index, we believe this is the right thing to do. If the assumption one was not present, we would always choose an extended index over a hash structure because an index has more capabilities than the corresponding hash structure.

The second assumption states that the same information should not be duplicated when this can be avoided. In particular, the first compact structure stored the distinct values of A only once in an extended index, while the second compact structure stores the same distinct value of A multiple times in a compact structure.

Complexity of cPART Merging

Theorem 13 (cPART merging is NP-Hard) *In the presence of the stated assumptions cPART merging is NP-Hard.*

Proof: We will prove the theorem by a reduction from the set-packing problem ([13]). Suppose that we are given an instance of the set packing problems. Let the given sets be $\{\mathcal{M}_i\}_{i=1}^m$. Let $\mathcal{M}$ be the set $\bigcup_{i=1}^{m} \mathcal{M}_i$ and f be a one-to-one function that maps the elements of $\mathcal{M}$ into the tuples of the table R with attributes $A, B_1, \ldots, B_m$ and $\bar{C}$. Let R_i be the tuples that f maps $\mathcal{M}_i$ to, where $i = 1$ to m. We next create the cPARTs $\mathcal{P} = \langle R, \bar{C}, \langle A\ \texttt{asc}\rangle, \texttt{dll}\rangle$ and $\{\mathcal{P}_i = \langle R_i, \bar{C}, \langle A\ \texttt{asc}, B_i\ \texttt{asc}\rangle, \texttt{ll}\rangle\}_{i=1}^m$. W.l.o.g. suppose that $\mathcal{P}_i$ is of size one unit for $i \in [1 \ldots m]$. Also assume that A is a key for the table R and that the size to store the value for the attribute A is much bigger than the size of a pointer.

Let $\{\mathcal{P}'_i\}_{i=1}^k$ be the solution to the ePART merging problem for the ePARTs $\mathcal{P}$ and $\{\mathcal{P}_i\}_{i=1}^m$. It is the case that in the optimal solution $\{\mathcal{P}_i\}_{i=1}^m$ cannot be merged only between themselves because such a merge will not save space. In particular, if $\mathcal{P}_i$ and $\mathcal{P}_j$ are merged ($i, j \in [1 \ldots m]$, $i \neq j$), then a index on the distinct values for the attribute A in $R_i \cup R_j$ needs to be build because the ordering of the two cPARTs are incompatible. Although this index will take space, it not save space because next/previous pointers for the records in R_i and R_j still need to be stored. Therefore, $\{\mathcal{P}'_i\}_{i=1}^k$ consists of the ePART $\mathcal{P}'$ and a subset of the ePARTs $\{\mathcal{P}_i\}_{i=1}^m$.

The ePARTs in the set $\{\mathcal{P}_i\}_{i=1}^m$ that are used in creating the ePART $\mathcal{P}'$ must be based on disjoint tables. Indeed, suppose that $\mathcal{P}_i$, $\mathcal{P}_j$, and $\mathcal{P}$ are merged into the ePART $\mathcal{P}'$ and $R_i \cap R_j \neq \emptyset$. Let $t \in R_i \cap R_j$. Then $\mathcal{P}'$ must contain a leaf node that is based on a table that contains $\mathcal{P}'$. However, since two incompatible orders (i.e. $\langle A$ `asc`$, B_i$ `asc`$\rangle$ and $\langle A$ `asc`$, B_j$ `asc`$\rangle$ are defined on this table, it must be the case that $\mathcal{P}'$ contains two distinct leaf nodes that are based on tables that contain the tuple t. Those nodes must be connected with some non-leaf nodes that are extended indices or hash structures. At least one of these nodes must contain the attribute A. However, an extended index or a hash structure on distinct A values will take much more space then it could save from eliminating pointers in the linked lists - contradiction.

Therefore if at least $N+1$ cPARTs contributed to the creation of the cPART $\mathcal{P}'$, then $\mathcal{M}$ contains N disjoint sets and the answer to the particular instance of set packing problem is yes. If less than $N+1$ cPARTs contributed to the creation of the cPART $\mathcal{P}'$, then the answer to the particular instance of the set packing problem is no. ■

Assuming that the length of the longest input sSQL query is constant, we will show that the cPART merging problem is in DEXPTIME. We will do so by constructing an exponential time algorithm that solves the problem.

Our algorithm first partitions the input cPARTs relative to the table they reference, where all the cPARTs that reference the same base table or that reference a MV that references only that base table are clustered together. As explained earlier in this chapter, cPARTs from different clusters will be based on different attributes and merging such cPARTs is not expected to save space.

Next, we will merge the cPARTs in each cluster using the following pseudo-code.

```
cPARTs merge_cPARTs(cPART P'_1, ..., cPART P'_k) {
   cPARTs result = ∅;
   int result_size = ∞;
   for {P̄_i}_{i=1}^m ∈ find_clustering(P'_1, ..., P'_k) {
     for (i = 1; i ≤ m; i++) {
       if ((|P̄_i| > 1) && ( ⋀_{P ∈ P̄_i}^k is_single_node(P_i)))
         skip to next iteration of outer for loop;
```

```
        if (merge_into_one($\bar{\mathcal{P}}_i$) == NULL)
          skip to next iteration of outer for loop;
        cPART $\mathcal{P}_i$ = merge_into_one($\bar{\mathcal{P}}_i$);
      }
      if (calculate_size($\{\mathcal{P}_i\}_{i=1}^{m}$) < $result\_size$) {
        $result = \{\mathcal{P}_i\}_{i=1}^{m}$;
        $result\_size$ = calculate_size($\{\mathcal{P}_i\}_{i=1}^{m}$);
      }
    }
}
```

```
merge_into_one(cPART $\mathcal{P}_1, \ldots,$ cPART $\mathcal{P}_k$) {
    cPARTs $result$;
    for ($i = 1; i \leq k; i++$) node $n_i$ = root($\mathcal{P}_i$);
    if (find_smallest_prefix($\{n_i\}_{i=1}^{k}$) == NULL)
      if $k > 1$ return NULL else return $\mathcal{P}_1$;
    $A$ = find_smallest_prefix($\{n_i\}_{i=1}^{k}$);
    $R = \underset{i=1}{\overset{k}{\texttt{union}}}\ R_i$;
    set of efficient predicates $\bar{\gamma} = \emptyset$;
    cPARTs $\bar{\mathcal{P}} = \emptyset$;
    Boolean $is\_hash$ = TRUE;
    for($i = 1; i \leq k; i++$) {
      if (type($n_i$) $\neq$ hash) $is\_hash$ = FALSE;
      if table(root($n_i$)) $\subset R_i$ {
        $\gamma$ : $\gamma(t)$ iff $t \in$ table(root($n_i$));
        $\bar{\gamma} = \bar{\gamma} \cup \{\gamma\}$;
        $is\_hash$ = FALSE;
      }
      $\bar{\mathcal{P}} = \bar{\mathcal{P}} \cup$ remove($\mathcal{P}_i, A$);
    }
    if ($is\_hash$ == TRUE)
```

20 $\quad\quad result = result \cup \langle R, \{A\}, [$`merge_cPARTs`$(\bar{\mathcal{P}})]\rangle$;
21 $\quad$ `else`
22 $\quad\quad result = result \cup \langle \bar{\gamma}, R, \langle A \rangle, [$`merge_cPARTs`$(\bar{\mathcal{P}})]\rangle$;
23 $\quad$ `return` $result$;
24 }

Line 3 of the `merge_cPARTs` pseudo-code uses the `find_clustering` method to enumerate all possible clustering of the ePARTs $\{\mathcal{P}'_i\}_{i=1}^k$. As shown in Section 4.2.7, there are $\mathcal{B}_k$ ways to cluster the ePARTs. For each clustering, Lines 4-10 find if the cPARTs in each cluster can be merged together, and if it can the result of the merge is recorded. Note that the pseudo-code does not merge cPARTs based on single nodes because, as the described earlier, building an extended index or a hash index to merge such cPARTs will not result in economy of space. On the other hand, as described in Definition 39, linked lists are stored on top of each other. Lines 11-14 are executed exactly when the cPARTs form each cluster were merged into a single node. The pseudo-code in this line calculates the size of the created cPARTs and if their size is smaller than the physical design that is found so far, then the found cPARTs are recorded as the smallest so far.

The `merge_into_one` pseudo-code tries to merge the input cPARTs into one. When this cannot be done, `NULL` is returned. In line 3 of the pseudo-code the `find_small_prefix` method tries to find a common prefix attribute of the ordering labels and ordering conditions of the root nodes of the input cPARTs. The method returns `NULL` when such a prefix does not exist. Note that when all the input cPARTs 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 an extended index or hash structure on it will save the most space. The union of the tables of the root nodes will be the table for the root of the resulting cPART, where we will denote this table as R. In lines 12-15 of the pseudo-code a γ condition is added to the root of the created cPART for each cPART that has a root node with table different then R.

Line 17 of the pseudo-code removes the common attribute from the cPARTs. In particular, if the common attribute is the last attribute in the ordering label of an index node or a hash node, then `remove`$(\mathcal{P}_i, A)$ returns $\mathcal{P}_i$ without the root node (note that we know that

$\mathcal{P}_i$ is a simple cPART, i.e. a path of nodes, and therefore this operations is well defined). In all other circumstances, the attribute A_i is removed from the ordering label or ordering condition of the cPART and the modified cPART is returned.

Lines 19-23 construct the result of the merge. In particular, a hash node or an extended index node with table R, γ condition the union of the required γ-conditions, and ordering label consisting of the common attribute, is created. The children of the node are determined by calling the `merge_cPARTs` recursively with parameters the input cPARTs without the common attribute.

Theorem 14 (properties of the cPART merging algorithm) *Under the following assumptions:*

- *lack of integrity constraints other that containment constraints,*
- *several cPARTs are mergable into a single cPART exactly when at least one of the cPARTs consists more than one nodes and:*
  - *the index root nodes of the cPARTs contain ordering labels that start with the same attribute,*
  - *the hash root nodes of the cPARTs contain ordering labels that contain this attribute,*
  - *the cPARTs that consist of a single node have the property that the ordering condition of this node starts with this attribute,*

the presented simple cPART merging algorithm is correct (i.e.if the cPARTs $\{\mathcal{P}_i\}_{i=1}^k$ are merged in the cPART $\mathcal{P}$, then $\mathcal{P} \in \bigcap_{i=1}^{k} \texttt{cover}(\mathcal{P}_i)$) and creates the cPARTs of the smallest size. Moreover, if N simple cPARTs are merged and the size of the longest cPART is m, then the algorithm takes $O((\mathcal{B}_N)^m)$ time.

Proof: We will first prove that if the cPARTs $\{\mathcal{P}_i\}_{i=1}^k$ are merged in the cPART $\mathcal{P}$, then $\mathcal{P} \in \bigcap_{i=1}^{k} \texttt{cover}(\mathcal{P}_i)$. Let i be an integer in the range $[1 \ldots k]$. We will prove that $\mathcal{P} \in \texttt{cover}(\mathcal{P}_i)$, i.e. that if $Q \in Q(\mathcal{P}_i)$, then $Q \in Q(\mathcal{P})$. Indeed, suppose that $Q \in Q(\mathcal{P}_i)$.

If Q is a sSQL query of type 1 or 2, then $\mathcal{P}_i$ will have at most 3 nodes that reference the partial match, range, and ordering attributes of Q in this order. Since $\mathcal{P}_\rangle$ is a simple cPART, it must also be the case that $\mathcal{P}_\rangle$ contains a path of at most 3 attributes and all nodes are based on the same table. However, $\mathcal{P}$ is constructed in such a way that in contains a complete path that contains the partial match, range, and ordering attributes of Q, and range and ordering attributes only appear in index nodes or leaf nodes. Moreover this path, which will refer as K, ends with a node that has the underlying table of Q, which will denote as R. As well, all the nodes along the path K are either based on the table R, or are index nodes that contain a γ condition such that $\gamma(t)$ is `TRUE` iff $t \in R$. From Definition 41 it follows that $Q \in Q_K(\mathcal{P})$ and from Theorem 5.2.4 it follows that $Q \in Q(\mathcal{P})$.

Our algorithm finds the smallest possible set of cPARTs because it explores all possible ways for merging the cPARTs. The running time of the algorithm is $O((\mathcal{B}_N)^m)$ because the algorithm does one partitioning of the space of N cPARTs and then it does $O(m)$ sub-partitionings. ■

Note that, in the presence of other integrity constraints, the create cPARTs are not necessarianism as simple as possible and therefore as small as possible. In particular, they can be simplified by the semantic cPART rewriting algorithm that is presented in Section 5.2.9.

Approximate Algorithm

We next present a polynomial time approximate algorithm for merging cPARTs. Our algorithm first partitions the input cPARTs relative to the table they reference, where all the cPARTs that reference the same base table or that reference a MV that references only that base table are clustered together. cPARTs from different clusters will be based on different attributes and merging such cPARTs is not expected to save space.

Next, a greedy approach is applied. Note that complexity of exact algorithm comes from the `find_clustering`$(\mathcal{P}_1, \ldots, \mathcal{P}_k)$ method. When applying the greedy algorithm, in stead of examining all possible clusterings for $\{\mathcal{P}_i\}_{i=1}^k$ we use heuristics to select a particular clustering. In particular, we put cPARTs that consists of a single node in separate clusters. Suppose that the input cPARTs that consists of more than one node are $\{\mathcal{P}_i\}_{i=1}^k$ and their paths are $\{K_i\}_{i=1}^k$. We next examine what could be the first attributes in $\{K_i\}_{i=1}^k$ under different permutations of the complete paths. Choose the possibility that saves most space, i.e. choose the attribute A if merging the cPARTs which paths start with the attribute A under some permutations will save more or equal space than choosing any other attribute and put those cPARTs in a separate cluster. Next choose the second best attribute and cluster the remaining cPARTs that start with that attribute under some permutations in a separate cluster. Repeat the procedure until all cPARTs are clustered.

One of the shortcomings of the presented greedy algorithms it that it can create leaf nodes with ordering conditions that are unnecessary. In particular, if the ordering condition of a leaf node can be removed without changing the set of queries that the cPART can be efficiently used to answer, then do so.

If there are N cPARTs to be merged, and m is the size of the longest cPART, then the presented greedy algorithm takes $O(m^2 \cdot N^2)$ time.

5.2.4 Step 8 - Semantic cPART simplification

In this step we will first simplify each index node of each cPART by performing the procedure described in Section 4.2.8. Next, we will examine all complete paths in each cPART. For each complete path K, examine the concatenation of the ordering labels and linked lists along the path K. If this sequence of attributes contains an attribute A that is functionally determined by the attributes before it, then delete the attribute from the node in K in which it was found to appear. If as a result of the deletion the ordering label of an index node or a hash node becomes empty, then delete the node all together and connect the parent of the node directly with the children of the node.

Theorem 15 *If the cPART $\mathcal{P}_1$ was rewritten into the cPART $\mathcal{P}_2$, then $Q(\mathcal{P}_1) \subseteq Q(\mathcal{P}_2)$.*

Proof: If $\mathcal{P}_2$ was created from $\mathcal{P}_1$ by simplifying and index node, then the theorem result follows from Theorem 11. Therefore, assume that this is not the case.

Let $Q \in Q(\mathcal{P}_1)$. We will show that $Q \in Q(\mathcal{P}_2)$.

First, consider the case when Q is a sSQL query of type 1 or 2 (see Figure 4.3) and let K be the complete path in $\mathcal{P}_1$ used to efficiently answer Q (see Theorem). If $\mathcal{P}_2$ was created from $\mathcal{P}_1$ by removing an attribute A_i from a node outside the path K, then Q will be efficiently answerable using $\mathcal{P}_2$ because the complete path K will not be affected.

Therefore, assume that the attribute A_i was removed from the node n that belongs to the path K. Let $\dots A_1, \dots, A_a$ be the nodes in the in the ordering labels and ordering condition of the nodes along the path K in this order.

We will next examine two cases: when A_i is part of a hash node and when A_i is part of an extended index. In the first case, the query Q can be efficiently answered using $\mathcal{P}_2$ by substitute the search in the deleted hash node for A_i with a check of whether the value of A_i for a record with the fixed values for $\{A_j\}_{j=1}^{i-1}$ has the required value for A_i. This check can be performed when a leaf node of the compact structure is reached and this does not sacrifice efficiency because the search structures for a compact structures are scanned depth-first. The reason we can do this substitution is that the defined functional dependencies guarantees that if $A_i = P_i$ for the current record, then $A_i = P_i$ for all records with the same value for the attributes $\{A_j\}_{j=1}^{i-1}$.

Next, consider the case when A_i is part of an extended index. We will examine two subcases: when A_i is an ordering attribute in Q and when a range condition is specified on A_i in Q. If A_i is an ordering attribute, then Q can be efficiently answered using $\mathcal{P}_2$ by skipping the ordering relative to the attribute A_i. This will not change the query result because the known functional dependency guarantees that if $A_i = P_i$ for the current record, then $A_i = P_i$ for all records with the same value for the attributes $\{A_j\}_{j=1}^{i-1}$.

Lastly, suppose that A_i is a range attribute in Q. Q can be answered using $\mathcal{P}_2$ by substituted the search for the tuples in which the attribute A_i is in some particular range with the check of whether the value of A_i for a record with the fixed values for $\{A_j\}_{j=1}^{i-1}$ is in the specified range. This check can be performed when a leaf node of the compact structure is reached and this does not sacrifice efficiency because the search structures for a compact structures are scanned depth-first.

Next, consider the case when Q is a sSQL query of type 3. Removing an attribute from the path K will not effect the capability of the compact structure to answer Q because the tables of the leaf nodes of the compact structure are not modified. ■

5.2.5 Step 9 - Converting cPARTs into Compact Structures

Since there is a one-to-one correspondence between cPARTs and compact structures, nothing needs to be done in this step.

Chapter 6

Analytical Results

Recall that in Chapter 2 we defined μSQL queries and MVs to have certain syntax. In Chapters 3 and 4 we showed that the chosen syntax is not too general, i.e. it μSQL queries to be executed efficiently and MVs to be efficiently maintained. In this chapter we show that the chosen syntax for critical queries and MVs is not too restrictive. In particular, under certain assumptions, we show that:

- the syntax for μSQL queries cannot be extended further without crossing the eSQL or cSQL boundary (see Figure 1.3) and
- the syntax for MVs can not be extended further without loosing their efficiently maintainable property or considering underlying queries for MVs that go beyond the examined in this work conjunctive queries with grouping and aggregation.

6.1 Efficiently Maintainable Materialized Views

The general syntax of an eSQL query is shown in Figure 2.3. The underlying query for a MV differs from an eSQL query in the following ways:

1. The underlying query of every MV can be rewritten to reference exclusively base tables and therefore w.l.o.g. we only consider such underlying queries.

2. We assume that the tuples in a MV are unordered and therefore the underlying query for a MV does not contain an `order by` clause.

3. Unlike eSQL queries, we do not consider parameterized MVs and therefore we do not allow expressions of the form "$B_1 = :P_1$" or "C_1 `between` $:P_1'$ `and` $:P_1''$" to appear in the `where` condition of the underlying query of a MV.

4. We allow grouping and aggregation in the underlying query of a MV. Note that grouping and aggregation in μSQL queries are specified precisely through the use of MVs.

Definition 62 (admissible view definition query) *We define the general syntax of the underlying query of a MV as follows.*

> `select` [`count`*(*)* `as` C,] $agg_{c_1}(A_1)$ `as` $C_1, \ldots, agg_{c_l}(A_l)$ `as` C_l, $A_{l+1}, \ldots, A_a$
> `from` T_1 `as` $e_1, \ldots, T_r$ `as` e_r
> `where` $\theta(e_1, \ldots, e_r)$ `and` $\gamma_1(e_1)$ `and` $\ldots$ `and` $\gamma_r(e_r)$
> `group by` $A_{l+1}, \ldots, A_a$,

where $agg \in \{\texttt{min}, \texttt{max}, \texttt{sum}, \texttt{avr}\}$, $\{|A_i|\}_{i=1}^{l}$ *are not-necessarily distinct standard attributes of* $\{T_i\}_{i=1}^{r}$, $\{|A_i|\}_{i=l+1}^{a}$ *are distinct non-*`ID` *attributes of* $\{T_i\}_{i=1}^{r}$, $\{A_i\}_{i=1}^{l} \cap \{A_i\}_{i=l+1}^{a} = \emptyset$, *and* θ *is a valid join graph for the base tables* $\{T_i\}_{i=1}^{r}$.

Recall Definition 14 of efficiently maintainable MVs. We will next examining the set of efficiently maintainable MVs relative to a physical design that consists of $|def(V)|_t$ regular indices for a MV V. Note that we assume that no integrity constraints can be deduced except for the foreign constraints that are needed to make the underlying join condition of the view a valid join graph.

6.1.1 Materialized Views with no Grouping and Aggregations

We will first examine the case when the underlying query for V has no aggregation or grouping in it, i.e. V has an underlying query of the following form:

select $A_1, \ldots, A_a$
from T_1 as $e_1, \ldots, T_r$ as e_r
where $\theta(e_1, \ldots, e_r)$ and $\gamma_1(e_1)$ and $\ldots$ and $\gamma_r(e_r)$

where $\{|A_i|\}_{i=1}^{a}$ are distinct attributes of the base tables $\{T_i\}_{i=1}^{r}$, and θ is a valid join graph for those tables.

Consider the MV V with the following underlying query over our example company schema from Figure 1.5.

select $e_1.name$, $e_2.title$
from EMP as e_1, DEP as e_2
where $e_1.dep = e_2.$ID

Suppose that there is a single department and several employees work for that department. If the title of the department changes, then all the tuples in the MV V will need to change. However, there is a tuple in V for every employee and therefore V is not an efficiently maintainable MV.

We will first show that the attributes $\{A_i\}_{i=1}^{a}$ must belong to a single base.

Theorem 16 *If V is an efficiently maintainable MV with underlying query with no grouping and aggregation, then the attributes $\{A_i\}_{i=1}^{a}$ must belong to a single base table.*

Proof: Indeed, suppose that the attributes $A_1, \ldots, A_a$ belong to the base tables T_1 and T_2 (see the general syntax for a MV). We will examine several cases.

First, suppose that there is a directed path from T_1 to T_2 in the graph defined by θ. Let T_1' and T_2' be the set of tuples from T_1 and T_2, respectively, that contribute to Q. It is possible that T_2' contains a single tuple and that T_1' contains half of the tuples in T_1. If the value of the tuple in T_2' is changed in such a way so that the same tuples are joined with it in Q, then at least $|T_1'|$ tuples will have to be modified in V, which implies that V will not be efficiently maintainable. On the other hand, if the value of the tuple in T_2' is changed in such a way so that new tuples are joined with it in Q, it may be the case that T_2' joins with $|T_1|/2$ new tuples and therefore at least $|T_1|/2$ tuples will have to be modified in V, which implies that in this scenario V will not be efficiently maintainable again.

Second, suppose that there is a base table T_3 such that there is a direct path from T_3 to T_1 and from T_3 to T_2 in the graph defined by θ. Let T_1', T_2' and T_3' be the tuples from T_1, T_2 and T_3, respectively, that contribute to Q. It is possible that T_1' contain a single tuple, while T_3' contains $|T_3|/2$ tuples and each tuple in T_3 joins with a different tuple in T_2. Then, an update to the tuple in T_1' that does not change the tuples that are joined with it in Q will require at least $|T_2'|$ modifications to V, which implies that V will not be efficiently maintainable. On the other hand, if the value of the tuple in T_1' is changed in such a way so that new tuples are joined with it in Q, it may be the case that T_1' joins with $|T_3|/2$ new tuples from T_3 and therefore at least $|T_3|/2$ tuples will have to be modified in V, which implies that in this scenario V will not be efficiently maintainable again.

Third, suppose that there is a base table T_3 such that there is a directed path from T_1 to T_3 and from T_2 to T_3 in the graph defined by θ. Let again T_1', T_2' and T_3' be the set of tuples selected from T_1, T_2 and T_3, respectively, in Q. It is possible that T_1' and T_3' contain a single tuple, while T_2' contains $|T_2|/2$ tuples. Then an update to the tuple in T_1' that does not change the tuples that are joined with it in Q will require at least $|T_2'|$ modifications to V, which implies that V will not be efficiently maintainable. On the other hand, if the value of the tuple in T_1' is changed in such a way so that new tuples are joined with it in Q, it may be the case that T_1' joins with a new tuple in T_3 that joins with $|T_2|/2$ new tuples from T_2 and therefore at least $|T_2|/2$ tuples will have to be modified in V, which implies that in this scenario V will not be efficiently maintainable again.

The forth case is when there is no directed path between T_1 and T_2 in the graph defined by θ and there does not exist a base table T_3 such that there are directed paths from T_1 and T_2 to T_3 or from T_3 to T_1 and T_2 in the graph defined by θ. Let again T_1' and T_2' be the set of tuples selected from T_1, and T_2, respectively, in Q. In this case it is possible that T_1' contains a single tuple, while T_2' contains $|T_2|/2$ tuples. Then, an update to the tuple in T_1' that doesn't change the tuples that are joined in Q will require at least $|T_2'|$ modifications to V and therefore V is not efficiently maintainable. On the other hand, if the value of the tuple in T_1' is changed in such a way so that new tuples are joined with it in Q, it may be the case that T_1' joins with $|T_2|/2$ new tuples from T_2 and therefore at least $|T_2|/2$ tuples will have to be modified in V, which implies that in this scenario V will not be efficiently maintainable again. ■

Consider a MV V with the following underlying query over our example schema from Figure 1.5.

> **select** $d.title$
> `from EMP as` e, `DEP as` d
> `where` $e.dep = d.$`ID`

Note that V is not efficiently maintainable. In particular, consider the case where there is a single department and all the employees work for that department. Initially, the MV will contain |`EMP`| tuples with the same value for the attribute *title*. Changing the title of the single department will require all the tuples in the MVs to be modified, and therefore the MV is not efficiently maintainable.

Theorem 17 *If V is an efficiently maintainable MV with underlying query Q with no grouping and aggregation, then the underlying query of V can be rewritten to reference a single base table.*

Proof: Suppose the V references more than one table and V is efficiently maintainable. W.l.o.g. suppose that only an attribute from underlying base table T_1 of V is projected out in V (see theorem 16).

First, suppose that there exist in the underlying join graph θ of V a base table T_2 that is different from T_1 and for which $\mathtt{descendants}^{\theta}(T_2) = \emptyset$. If the efficient predicate of T_2 is a tautology, i.e. equivalent to `TRUE`, then the table T_2 can be removed from the underlying query of V without changing its semantics (see Section 4.2.3). Therefore, assume that the efficient predicate γ_2 for T_2 is not equivalent to `TRUE`. Suppose that initially T_2 contains a single tuple and all the tuples in all the tables referenced in Q contribute to V, i.e. all tuples pass all efficient predicates and join with at least one tuple from the other tables. Modifying the single tuple in T_2 to a value that does not pass the efficient predicate defined on the base table T_2 will make V empty. Modifying it again to a value that passes the efficient predicate defined on the base table T_2 could introduce an arbitrary number of new tuples. Therefore, V is not efficiently maintainable in this case.

Since a base table T different from T_1 for which $\mathtt{descendants}^{\theta}(T) = \emptyset$ does not exist, θ must be in the shape of an inverse tree with root T_1. Suppose that T_1 contains a single tuple.

Then V could contain a number of tuples that are the same because only attributes from T_1 could be projected out in V. Modifying the single tuple in T_1 will result in modifying all the tuples in V and therefore V is not efficiently maintainable. ■

In this subsection we have shown that if the underlying query for a MV contains no grouping and aggregation, then it must have the syntax for efficiently maintainable MVs of type 1 (see Table 2.5)

6.1.2 Materialized Views with Grouping and Aggregation

In this section we consider MVs with admissible view definition queries (see Definition) that contain a `group by` clause.

Consider an example MV with the following underlying query.

```
select e.name, d.title, count(*) as C
from EMP as e, DEP as d
where e.dep = d.ID
group by e.name, d.title
```

It may be the case that there is a single department all the employees belong to that department. It may also be the case that all the employees have different names. Than a single change to the title of the department will involve modifying all the tuples in the MV and therefore it is not efficiently maintainable.

We will first show that the attributes over which the grouping is defined must belong to a single table.

Theorem 18 *If V is an efficiently maintainable MV with admissible view definition underlying query that contains a* `group by` *clause, then the* `group by` *clause can be defined using attributes from at most one table.*

Proof: This proof is similar to the proof of Theorem 16. Suppose that Q is an underlying query of V that contains grouping on the attributes from the two distinct tables T_1 and T_2. Since we do not allow 1:1 integrity constraints to be specified between tables, it may be the case that the same compound tuple from T_1 can appear multiple times with different compound tuples from T_2 or the same compound tuple from T_2 can appear multiple times with different compound tuple from T_1. W.l.o.g. suppose that the same compound tuple of T_1 joins with multiple compound tuples of T_2 that all appear in V. If the compound tuple in T_1 is constructed from a single tuple, then a change to an attribute of this tuple that appears in the `group by` clause of Q will result in changing multiple tuples in V and therefore V will not be efficiently maintainable. ■

Next, consider a MV with the following underlying query.

```
select e.name, agg(d.budget)
from EMP as e, DEP as d
where e.dep = d.ID
group by e.name,
```

where $agg \in \{\texttt{min}, \texttt{max}, \texttt{sum}, \texttt{avr}\}$. This query is equivalent to the following query.

```
select e.name, d.budget
from EMP as e, DEP as d
where e.dep = d.ID
group by e.name, d.budget
```

From Theorem 18 we now that a MV that is defined using the rewritten query cannot be efficiently maintainable and therefore the MV is not efficiently maintainable.

This example generalizes in the following theorem.

Theorem 19 *If V is an efficiently maintainable MV with admissible view definition underlying query Q that contains a* `group by` *clause, then the join conditions for Q must be in the shape of an inverse tree and only attributes that belong to the root table of the tree can appear in the* `group by` *clause of Q.*

Proof: Suppose that Q is an underlying query of V and let the `group by` attributes of Q belong to the base table T_1.

Assume that there exist in the join graph θ of Q a base table T_2 that is different from T_1 and for which $\texttt{descendants}^{\theta}(T_2) = \emptyset$. If an aggregate is defined on some of the attributes of T_2, then Q can be rewritten into an equivalent query in which attributes from T_1 and T_2 appear in the `group by` clause and therefore V will not be efficiently maintainable.

Therefore, assume that no aggregates are defined on the attributes of T_2. If the efficient predicate of T_2 is a tautology, i.e. equivalent to TRUE, then the table T_2 can be removed from Q without changing its semantics (see Section 4.2.3). Therefore, assume that the efficient predicate γ_2 for T_2 is not equivalent to TRUE. Suppose that initially T_2 contains a single tuple and all the tuples in all the tables referenced in Q contribute to V, i.e. all tuples pass all efficient predicates and join with at least one tuple from the other tables. Modifying the single tuple in T_2 to a value that does not pass the efficient predicate defined on the base table T_2 will make V empty. Modifying it again to a value that passes the efficient predicate defined on the base table T_2 could introduce an arbitrary number of new tuples. Therefore, V is not efficiently maintainable in this case.

This implies that there does not exist a table T_2 different from T_1 that has the property that $\texttt{descendants}^\theta(T_2) = \emptyset$ and therefore θ is in the shape of an inverse tree with root T_1. ■

Next, consider a MV V with the following underlying query Q over our example database schema from Figure 1.5.

```
select max(e1.age) as max_age, e2.title
from EMP as e1, DEP as e2
where e1.dep = e2.ID
group by e2.title
```

In the presence of arbitrary updates to the tables EMP and DEP we can efficiently maintain V only if we store information about the different ages in the departments with the same title ([26]). Suppose we somehow encode this information using regular indices. W.l.o.g, assume that there are two departments with the same title and half of the employees work for one of the departments and the other half for the other. W.l.o.g. assume also that each employee has a distinct age. Changing the title of one of the departments would involve moving half of the ages from the structure for the old department's title to the structure for the new department's title, which would take more than logarithmic time. This implies that V is not efficiently maintainable.

The presented example generalized to the following theorem.

Theorem 20 *If V is a MV with admissible underlying query with* min *or* max *aggregation, then there exists an underlying query for V with table join condition θ in which the* min *and* `max` *aggregations are performed only on the attributes that belong to the table represented by the root of the inverse three specified by θ.*

Proof: Suppose that Q is an admissible underlying query of V. Let θ be the join condition for Q and T_1 be the root of the inverse tree represented by θ. Suppose that Q contains the aggregation $max(A)$ and $A \in T_2$ where $T_2 \not\equiv T_1$. Since θ is in the shape of an inverse tree, there must exist directed path from T_2 to T_1 in θ. Let $\bar{B}$ be the `group by` attributes of T_1 in Q. In the presence of arbitrary updates to the tables T_1 and T_2, we can efficiently maintain V only if we store information about the distinct values for the attributes $\bar{B}$ and the corresponding values for the attribute A.

Suppose we somehow encode this information using regular indices. W.l.o.g., suppose that there are two tuples in T_1 with different values for the attributes $\bar{B}$ and each joins with a distinct half of the tuples in T_2. W.l.o.g. assume also that each tuple in T_2 has a different value for A. Changing the value for the attributes $\bar{B}$ of one of the tuples in T_1 to be equal to the value for the attributes $\bar{B}$ of the other tuple in T_2 would involve moving half of the values for the attribute A from the structure for the old value of $\bar{B}$ to the structure for the new value of $\bar{B}$, which would take more than logarithmic time. ■.

We have shown that under certain assumptions MVs must have the syntax shown in Table 2.5. This is indeed the only syntax we allow for the underlying queries of MVs because we require that all MVs we consider are efficiently maintainable.

6.2 Critical Queries

In this section we will show that if we impose the following restrictions on the syntax of an eSQL Q query, then Q becomes a μSQL query.

- There exists an efficient plan for answering Q using $|def(Q)|_t$ regular indices.
- There exists an efficient plan for updating each of the indices after every possible primitive update to the database.
- No integrity constraints can be inferred except for the foreign constraints that are needed to make Q a valid eSQL query.

We will first show that the parameterized restrictions must belong to a single table. Consider the following query Q over our example company schema.

```
select *
from EMP as e1, DEP as e2
where e1.dep = e2.ID and e1.age = :P1 and e2.title = :P2
```

One strategy is to create a combined index on the attributes *age* and $title'$ for the tuples in the table EMP, where $title'$ is a derived attribute where for $t \in$ EMP, $t.title' = (t \Rightarrow dep).title$. Such a physical design can obviously be used to answer Q efficiently. The problem is that an update to the table DEP cannot be propagated to the created index efficiently. In particular, suppose that there are two departments with the same title and each joins with a distinct half of the employees. Then, an update that makes the titles of the two departments distinct will require rearranging half on the records in the index, which obviously cannot be done in logarithmic time.

A second strategy is to avoid creating derived attributes. We can answer Q, for example, by first probing an index on the attribute *title* of the table DEP with the value P_2 and then finding all the tuples in the table EMP that join with the found tuples. However, such a plan is not efficient. For example, it may be the case that the index probe selects all the tuples in the table DEP, i.e. all the tuples in the table DEP have the same title, but all those tuples also join with tuples in the table EMP for which the value for the attribute *age* is not equal to P_1. In other words, the proposed query plan may require scanning the whole table DEP when the query result is empty and therefore this plan is not efficient.

A third strategy is to consider answering Q by first probing an index on the attribute *age* of the table EMP with the value P_1 and then finding the corresponding tuple in the table DEP. Such a query plan can not be considered efficient for the same reason, i.e. it may be the case that the index probe returns all the tuples in the table EMP, but none of them joins with a tuple in the table DEP that has value P_2 for the attribute *title*. The problem with the last two strategies is that they scans tuples that are not guaranteed to contribute to the query result.

Since we have restricted the number of index scans to two, we have exhausted the possible query plans that can potentially answer Q in logarithmic time and therefore Q is not a μSQL query.

The presented example generalizes to the following theorem.

Theorem 21 *If Q is an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of a primitive update, then Q contains a parameterized restrictions imposed on at most one table.*

Proof: Indeed, suppose that parameterized restrictions are defined on two different tables of Q. As shown in the motivating example, two separate indices for the two tables will not help answer Q efficiently. The reason is that probing one of the indices does not guarantee that the found values will contribute to the query result. The only exception is when we know that probing one of the indices will always return a single tuple. However, there is no way to know this in the absence of uniqueness constraints.

The only other alternative is to create a join index on the two tables. However, as shown in the motivating example, such an index can not be efficiently maintained. The reason is that the MV on which the index is built is not efficiently maintainable. ■

Next, consider the following query over our company schema from Figure 1.5.

select $*$
from EMP as e_1, DEP as e_2
where $e_1.dep = e_2$.ID and $e_1.age = :P_1$ and $e_2.title =$ "sales"

One possibility is to first probe the index of the employees ordered by *age* for the age P_1. This scan can return several employees and for each found employee we need to checks whether it works in a department with title sales. However, this strategy is not efficient because it may be the case that all employees have the same age P_1, but none of them works for a department with *title* = sales, i.e. all employees are scanned without returning a single tuple in the query result.

A different strategy is to create a MV V for the departments with *title* = "sales". As shown in Section 3.2, such a MV can be efficiently maintained. Then we can scan V and for each entry in V scan an index on the table EMP that returns the employees in the given department. The problem is that such a plan is not efficient because it may be the case that all the departments have *title* = sales, but none of them have an employee with the specified *age*.

A third strategy is to modify the first strategy by building an index only on those employees who work for a department with *title* = "`sales`". However such an index cannot be efficiently maintained. For example, suppose that there are two departments and both have *title* = "`sales`". Suppose as well that half of the employees work for one of the departments and the other half of the employees work for the other department. Changing the *title* of one of the departments will require removing half of the tuples from the index, which cannot be performed efficiently.

The presented example generalizes to the following theorem.

Theorem 22 *Let Q be an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update. Suppose that there is a parameterized restriction in Q imposed on the table R_1. Then, if there is a non-parameterized restriction in Q on the table R_2, it must be the case that there is a directed path from R_2 to R_1 in the join tree defined by Q.*

Proof: Let θ be the join tree defined by Q and γ by the non-parameterized restriction for R_2. Indeed, suppose that there does not exists a directed path from R_2 to R_1 in θ. We have the following possibilities for building indices to answer Q.

1. A joint index on R_2 and R_1.
2. A joint index on $\gamma(R_2)$ and R_1.
3. First scan an index on R_2.
4. First scan an index on $\gamma(R_2)$.
5. First scan an index on the tuples in $\gamma(R_2)$ that join with R_1.
6. First scan an index on R_1.
7. First scan an index on the tuples in R_1 that join with R_2.
8. First scan the tuples in R_1 that join with $\gamma(R_2)$.

We will show that all of the enumerated strategies are unsuccessful.

The first two strategies are unsuccessful because they produce an index that can not be efficiently maintained. The reason is that the index is based on a MV that cannot be efficiently maintained.

Consider next Strategies 3-5. Not one of them can be used to answer the query Q efficiently. The reason is that they all involve first scanning an index for R_2 and it maybe the case that the resulting tuples from the scan do not join with tuples in R_1 that pass the parameterized restriction. On the other hand, an index on the tuples in R_2 that join with tuples in R_1 that pass the parameterized restriction will be equivalent to Strategy 2.

Consider Strategies 6 and 7. The query Q cannot be efficiently evaluated with either strategy because there is no guarantee that the scanned tuples of the index for R_1 join with tuples in R_2 that pass the condition γ.

Lastly, consider Strategy 8. Note that if there was a directed path in θ from R_2 to R_1, then this strategy would be successful (See Section 4.2.4). However, we will show that from our assumption that there does not exist a directed path from R_2 to R_1 in θ it follows that the described index is not efficiently maintainable. In particular, suppose that there are two tuples in R_2 and half of the tuples in R_1 join with one of the tuples and the other half - with the other (The fact that there is not a directed path from R_2 to R_1 in θ makes this case possible). Suppose as well that both tuples in R_2 pass the condition γ. Now, if one of the tuples is modified to a value that does not pass the condition γ, then half of the tuples from the described index may have to be removed and therefore the index is not efficiently maintainable. ■

We will next show that if Q is a critical eSQL query with the described properties, then the join condition θ of Q cannot contain a pattern in the shape of a "W" (see Figure 6.1).

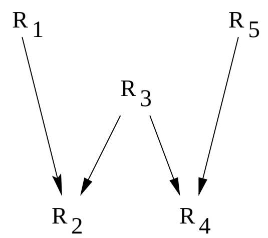

Figure 6.1: The "W" Pattern

Theorem 23 *Let Q be an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update. Then Q does not contain 5 tables that form a "W" pattern. In other words, if θ is the join tree of Q, then Q does not contain tables $\{R_i\}_{i=1}^{5}$ such that there is a directed path from R_1 to R_2, from R_3 to R_2, from R_3 to R_4, and from R_5 to R_4 in θ.*

Proof: Indeed, suppose that Q is a critical query with join condition θ, Q is based on the tables R_1, R_2, R_3, R_4 and R_5, and there is a directed path in θ from R_1 to R_2, from R_3 to R_2, from R_3 to R_4, and from R_5 to R_4 (see Figure 6.1).

First consider a query plan that starts with an index scan on some of the tuples of R_2 (a query plan that starts with an index scan on some of the tuples of R_4 will by analogous). Consider what an index X_2 on R_2 could looks like. For example, all the tuples in R_2 that join with tuples in R_1 and R_3 relative to θ could be indexed. We could also build an index X_1 on R_1 and X_3 on R_3 that return the tuples in the two tables that join with a particular tuple in R_2. Then an access could scan all the tuples in X_2 first. Then, for each resulting tuple the indices X_1 and X_3 could be scanned. For each resulting tuple of the index scan in X_3 the corresponding tuple in X_4 could also be retrieved by computing the value of the corresponding derived attribute. The problem with this access plan is that it can select only tuples in R_4 that do not join with tuples in R_5. This means that is possible to scan the whole index X_2 and produce and empty query result, i.e. the proposed query plan is not efficient.

Another alternative is to build an index on the tuples of R_2 that join with tuples in R_3, R_4, and R_5. However, such an index is not efficiently maintainable. For example, suppose that there exists a tuple t_4 in R_4 that joins with all the tuples in R_5. Suppose as well that this tuple initially does not contribute to the query result because it does not join with any tuples in R_3, i.e., the query result is initially empty. Next, a tuple $t_3 \in R_3$ may be modified to join with t_4, which may result in adding $|R_5|$ tuples to the query result and therefore the index is not efficiently maintainable.

Second, consider a query plan that starts with an index scan on some of the tuples of R_3. If the index contains all of the tuples in R_3, then the query plan will not be efficient because it might be the case that not one of the tuples in R_3 contributes to the query result. On the other hand, an index that stores the tuples in R_3 that join with the tuples in R_4 and R_5 is not efficiently maintainable. For example, suppose that there exists a tuple t_4 in R_4 that joins with all the tuples in R_5. Suppose as well that this tuple initially does not contribute to the query result because it does not join with any tuples in R_3, i.e., the query result is initially empty. Next, a tuple $t_3 \in R_3$ may be modified to join with t_4, which could result in adding $|R_5|$ tuples to the query result and therefore the index will not be efficiently maintainable.

Lastly, consider a query plan that starts with an index scan on some of the tuples in R_1 (a query plan that starts with an index scan on some of the tuples of R_5 will by analogous). As shown earlier, if the index scans the tuples in R_1 that join with R_2 and R_3, then the index will not be efficiently maintainable. On the other hand, an index that stores only the tuples in R_1, which all join with the tuples in R_2, could not be used to answer the query efficiently because the selected tuples do not have to join with tuples in R_3 and therefore may not contribute to the query result. ■

We will next show that if Q is an eSQL query that can be efficiently answered using an efficiently maintainable physical design, than Q does not contain four tables that form a *lightening pattern*. The lightening pattern is depicted in Figure 6.2. It represents a table join condition that contains four tables $\{R_i\}_{i=1}^4$ such that there exists a direct path from R_1 to R_2, from R_3 to R_2, and from R_3 to R_4. We also require that a non-parameterized or a parameterized predicate is specified on the table R_4 (If the predicate is absent, then Chapter 4 shows how a query with this pattern can be efficiently answered using a physical design that can be efficiently maintained).

Theorem 24 *Let Q be an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update. Then Q does not contain 4 tables that form a lightening pattern.*

Let θ be the join graph for Q. Suppose that Q does contain a lightening pattern formed by the tables $\{R_i\}_{i=1}^4$ and a predicate is specified on the table R_4.

As shown in the proof of Theorem 23, an approach in which an index on some of the tuples of R_1, R_3, or R_4 is first scanned is not successful. Therefore, we assume that an index on some of the tuples of R_2 is first scanned. If this index contains all the tuples in R_2 that join with tuples in R_3, then this index will not be useful for efficiently answering

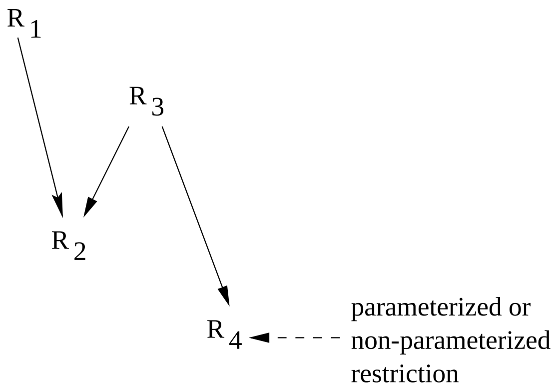

Figure 6.2: The "Lightening" Pattern

Q. The reason is that it may be the case that not a single tuple from the index joins with a tuple in R_4 that passes the predicate defined on the table.

Next, consider an index on R_2 that stores the tuples in R_2 that join with tuples in R_3 and with the tuples in R_4 that pass the predicate (this scenario is only possible when the predicate defined on R_4 is not parameterized). Such an index is not efficiently maintainable. In particular, suppose that all the tuples in R_3 join with a single tuple t_4 in R_4. If t_4 does not pass the predicate defined on R_4, then the query result will be empty. However, if t_4 is modified and the new value passes the predicate γ, then it can be the case that $|R_2|$ new tuples added to the index as a result of a single primite update, and therefore the index is not efficiently maintainable. ■

We will next show that if Q is an eSQL query that can be efficiently answered using an efficiently maintainable physical design, than Q does not contain three tables that form the Λ *pattern* depicted in Figure 6.3. It represents a table join condition that contains the tables $\{R_i\}_{i=1}^{3}$ such that there exists a direct path from R_2 to R_1 and from R_2 to R_3. We also require that a non-parameterized predicates are specified on tables R_1 and R_3 (If a non-parameterized predicate is specified on only one of the tables, then Section 4.2.4 shows how a query with this pattern can be efficiently answered using a physical design that can be efficiently maintained).

Theorem 25 *Let Q be an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update.*

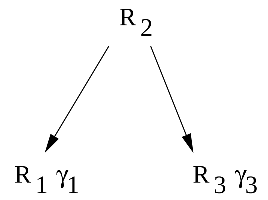

Figure 6.3: The Λ Pattern

Then Q does not contain 3 tables that form a Λ pattern.

Let θ be the join graph for Q. Suppose that Q does contain a Λ pattern formed by the tables $\{R_i\}_{i=1}^{3}$, the non-parameterized predicate of R_1 is γ_1, and the non-parameterized predicate of R_3 is γ_3.

As shown in the proof of Theorem 23, an approach in which a joint index on more than one tables is created is not feasible because such an index cannot be efficiently maintained.

We will first examine the strategy in which an index on some of the tuples in R_2 is scanned first. If this index is on all of the tuples in R_2, then such an index will not help answer Q efficiently. The reason is that it may be the case that not a single tuple in the index joins with tuples in R_1 and R_3 that pass the predicates γ_1 and γ_3, respectively. Therefore, assume that the index is on only those tuples in R_2 that join with tuples in R_3 that pass the predicate γ_3. However, such an index cannot be efficiently maintained. In particular, suppose that half of the tuples in R_2 join with one tuple in R_3 and the other half of the tuples in R_2 join with a different tuple in R_3. Suppose that initially both tuples in R_3 pass the condition γ_3. Now, if a primitive update modifies one of the tuples in R_3 so that the tuple no longer passes the γ_3 condition, then half of the tuples from the index may have to be removed, which cannot be done efficiently.

We will next examine the strategy in which an index on some of the tuples in R_1 is scanned first (The strategy where an index of some of the tuples in R_3 is scanned first is analogous). If this index is on all of the tuples in R_1 or all in the tuples in R_1 that pass the predicate γ_1, then such an index will not help answer Q efficiently. The reason is that it may be the case that not a single tuple in the index joins with a tuple in R_3 that passes the predicate γ_3. Therefore, assume that the index is on only those tuples in R_1 that pass the predicate γ_1 and that join with a tuple in R_2 that joins with tuples in R_3 that pass the predicate γ_3. However, such an index is not efficiently maintainable. In particular, suppose that there is a single tuple in R_1 that passes the γ_1 condition, it joins with all the tuples in R_2, and half of the tuples in R_2 join with one of the tuples in R_3 and the other half joins with the other tuple in R_3. Suppose that initially both tuples in R_3 pass the predicate γ_3. Now, if a primitive update modifies one of the tuples in R_3 to a new value that does not pass the γ_3 predicate, then half of the tuples in the index may need to be deleted and therefore the index is not efficiently maintainable. ■

Consider an eSQL query Q that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update. Let θ be the join condition of the query.

First, assume that the query contains a parameterized condition. Then, by theorem 21 it follows that this condition can be specified on at most one table. Let this table be R. Let G^t be the inverse tree in θ with root R. Theorem 22 implies that non-parameterized predicates can only be specified on the tables along the path G^t. Theorem 24 implies that if n is a node that is not in G^t, then there is a single incoming edge to n and there is a directed path from a node in G^t to n. This implies that θ indeed has exactly the required shape for μSQL queries.

Next, assume that the query contains no parameterized restrictions, but at least one non-parameterized restriction. Assume that there does not exist an inverse tree in θ that contains all of the underlying tables of Q on which non-parameterized restrictions are specified. This implies that there exist two distinct nodes n_1 and n_2 in θ that refer to tables with parameterized restrictions, but there does not exist a directed path between the nodes and there does not exist a node n_3 in θ s.t. there is a directed path from n_1 to n_3 and from n_2 to n_3. However, this implies that the nodes n_1 and n_2 are connected either using a Λ pattern, or using a lightening pattern, or that θ contains a "W" pattern. Theorems 25, 24 and 23 show that this can not be the case, which creates the contradiction. Therefore, in this case θ should also have the shape of a join condition for μSQL queries.

Lastly, assume that Q contains no parameterized or non-parameterized restrictions. The fact that θ contains no W patterns implies that θ has the shape of a join condition for μSQL queries.

We have proven the following theorem.

Theorem 26 *Let Q be an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update and let θ be the join condition of Q. Then θ has the required shape for μSQL queries.*

We will next examine what the parameterized condition for a critical query could look like. Recall that the parameterized condition for an eSQL query is of the following form.

$B_1 = {:}P_1$ `and` $\ldots$ `and` $B_b = {:}P_b$ `and`
C_1 `between` ${:}P_1'$ `and` ${:}P_1''$ `and` $\ldots$ `and` C_c `between` P_c' `and` P_c''

We will next show that the if Q is an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices, then Q cannot contain multiple range parameterized predicates.

In particular, suppose that Q has the following syntax.

`select` $*$
`from` R `as` e
`where` $e.A$ between ${:}P_1$ and ${:}P_2$ and $e.B$ between ${:}P_3$ and ${:}P_4$

Such a query cannot be answered using $|def(Q)|_t$ indices. In particular, a joint index on the attributes A and B could not be used to answer the query efficiently. It is also the case that an index on either of the attributes cannot be used to answer Q efficiently in the absence of uniqueness constraints. The reason is that such an index can select tuples that do not contribute to the query result.

In order to finish the proof that the examined critical eSQL queries must be μSQL it remains to show that the order condition in the query can be rewritten to be valid for some allowed inverse three in the join condition of θ (see Definition 12). We will do so by examining several table join patterns involving ordering conditions.

Theorem 27 *Let Q be an eSQL query that can be efficiently answered using $|def(Q)|_t$ number of indices which can be efficiently maintained after every kind of primitive update. Let θ be the join condition of Q. Then there exists an inverse tree G^{-t} that is a valid inverse three in θ for μSQL queries and the tuple ordering defined in Q can be rewritten to be valid relative to G^{-t}.*

Proof: Suppose that the ordering condition in Q is "$O = \langle A_1\ dir_1, \ldots, A_a\ dir_a \rangle$" and let θ be the join table condition of Q. We will prove that the four conditions from Definition 12 must hold for the ordering. In order to understand why the four conditions hold, recall that we allow query plans for Q that use $|def(Q)|_t$ number of indices which can be efficiently maintained. However, as shown in the proof of Theorem 21, a joint index on more than one of the underlying tables of Q cannot be efficiently maintained. Therefore, the physical design must consist of a set of indices, where there is at most one index for each of the underlying tables of Q.

First, assume that Condition 2 from Definition 12 does not hold, i.e. there exist $1 \leq i < j < k \leq a$ such that $A_i, A_k \in R_1$, $A_j \in R_2$, where R_1 and R_2 are two distinct underlying tables of Q. The only way such an ordering can be implemented is if some index scans on R_1 and R_2 are performed. The reason is that sorting N elements cannot be implemented faster than $O(N \cdot log(N))$ time without using additional data structures, i.e. the tuples need to come in the correct order from the index scan. If the desired ordering can be achieved by an index scan on only R_1 or only R_2, then the order condition can be rewritten to reference only one of the tables. If the desired ordering can be achieved by an index scan on only R_1 followed by an index scan on R_2, or an index scan on R_2 followed by an index scan on R_1, then the ordering condition can be rewritten so that it satisfies Condition 2 of Definition 12. The only other possibility is that answering the query requires an index scan on R_1 followed by an index scan on R_2 followed by an index scan on R_1. If there is a directed path between R_1 and R_2 in θ, then the ordering on R_2 that follows an ordering on R_1 can be removed from the query without changing its semantics. Therefore, assume that there does not exist a directed path between R_1 and R_2 in θ. If A_i is the `ID` attribute, then the second ordering on R_1 can be performed before the ordering on R_2 - contradiction. Therefore, assume that the A_i is not the `ID` attribute. This means that it is possible that R_1 contains several tuples with the same value for A_i and each joins with a number of tuples in R_2. Therefore, the second index scan on R_1 could not impose an ordering on the tuples of R_1 - contradiction. This implies that O can be always rewritten so that Condition 2 from Definition 12 holds.

Next, consider Condition 1 from Definition 12. We will examine three cases: when Q contains a parameterized restriction, when Q does not contain a parameterized restriction, but it contains non-parameterized restrictions, and when the only restriction on the tables of Q is the join condition.

First, suppose that Q contains a parameterized restriction. Let R_1 be the table on which the restriction is specified. Theorem 26 guarantees that Q contains an inverse tree with root R_1 and all the non-parameterized restrictions are on the tables in this tree. Suppose that the tuple ordering contains an attribute A that belongs to a table R_3 outside the inverse tree and the ordering on A cannot be removed without changing the query semantics. As shown in Theorem 22, any successful plan for answering Q must start with an index scan on R_1. Let R_2 be the unique node in θ that is part of the inverse tree and for which there is a directed path from R_2 to R_3 in θ that does not pass through any nodes of the inverse tree. The tuples from R_3 that contribute to the query result cannot be efficiently found without first finding the tuples from R_2 that contribute to the query result, i.e. the query plan must involve index scans on R_1, R_2, and R_3 in this order. However, the index scan on R_3 cannot efficiently change the already established order. The reason is that each tuple in R_3 joins with at most one tuple in R_2 and therefore any order defined on R_3 will not change the final ordering of the query result and therefore it can be removed from O without changing the query semantics - contradiction.

Next, suppose that Q does not contain parameterized restrictions, but it contains at least one non-parameterized restriction. We will prove that if a non-parameterized restriction is defined on table R_1, then there does not exist tables R_2 and R_3 such that there is a directed path from R_2 to R_1 and from R_2 to R_3 in θ and an ordering is defined on R_3 that is important (i.e. if the attributes of R_3 are removed from the ordering, then the semantics of the ordering will change). Suppose that this is not the case. From the proof of Theorem 25 it follows that any successful plan for answering Q must contain index scans on the tables R_1, R_2, and R_3 in that order. However, the index scan on R_3 cannot efficiently change the already established order. The reason is that each tuple in R_3 joins with at most one tuple in R_2 and therefore any order defined on R_3 will not change the final ordering of the query result and therefore it can be removed from O without changing the query semantics - contradiction.

Finally, suppose that Q does not contain any restrictions. We will show that Q does not contain four tables $\{R_i\}_{i=1}^{r}$ that have the property that there is a directed path from

R_1 to R_2, from R_3 to R_2, and from R_3 to R_4, where some of the attributes of R_4 participate in O and O cannot be rewritten in a way so that they do not. Suppose that this is not the case. From the proof of Theorem 23 it follows that any successful plan for answering Q must contain index scans on the tables R_2, R_3, and R_4 in that order. However, the index scan on R_4 cannot efficiently change the already established order. The reason is that each tuple in R_3 joins with at most one tuple in R_4 and therefore any order defined on R_4 will not change the final ordering of the query result and therefore it can be removed from O without changing the query semantics - contradiction.

Next, consider Condition 3 from Definition 12. Suppose that the condition does not hold, i.e. there is a directed path from R_1 to R_2 in θ, but the attribute A from R_1 comes before the attribute B from R_2 in O and O cannot be rewritten in a way so that this stops to be the case. However, for each tuple in R_1 there is at most one tuple in R_2, which means that the ordering on the attribute B can be removed from O without changing the order's meaning - contradiction.

Next, consider Condition 4 from Definition 12. Suppose that the condition does not hold, i.e. R_1 contains the attribute A that is not the `ID` attribute, R_2 contains the attribute B, there is a directed path between R_2 and R_1 in θ, A and B both appear in the definition of O, and O cannot be rewritten to exclude one of the two attributes. If Q can be efficiently answered by an index scan on some of the attributes of R_2 followed by an index scan on some of the attributes of R_1, then O can be rewritten to exclude the attribute A. Therefore, it must be the case that the index scan on some of the attributes of R_1 appears before an index scan on some of the attributes of R_2. Since A is not the `ID` attribute, it can be the case that more than one tuples in R_1 have the same value for the attribute A. W.l.o.g. assume that A is the only attribute from R_1 that appears in O. It can be the case that all the tuples in R_1 have the same value for the attribute A and each tuple in R_1 joins with a single distinct tuple in R_2. Assuming that all the tuples in R_1 and R_2 contribute to the query result, a query plan that scans R_1 first and then scans R_2 could not be efficient. The reason is that the resulting tuples from R_2 will not be ordered by the required ordered and ordering them will require $O(N \cdot log(N))$ time if N is the number of tuples in R_1. ■

a

Chapter 7

Experimental Evaluation

We conducted a set of experiments based on the TPC-C benchmark ([34]), which is based on an OLTP like scenario. We chose this benchmark because it contains a mixture of simple queries and updates that mimic a typical ECP workload. We have simplified the experimental setup by running the experiments on a single PC. The parameters or the PC are:

- 1.5 GHz Intel Pentium IV CPU,
- 256 MB of main memory,
- hard disk with rotating speed of 7200 rpm and average seek time of 8.5 ms, and
- Windows 2000 operating system.

Although the TPC-C benchmark is designed for a client-server scenario, we decided to run the experiments on a single computer in order to eliminate variations in the results due to network congestion. It is our believe that if the experiments were ran in a client-server scenario with a dedicated connection, then the experimental results would be similar. The databases on which the experiments were ran were IBM DB2 UDB v8.1 and Microsoft SQL Sever 2000.

The TPC-C workload consists of five transaction types:

1. entering orders,

2. delivering orders,
3. recording payments,
4. checking the status of orders,
5. and monitoring the level of stock at the warehouses.

We first created two copies of the initial TPC-C database with 10 warehouses, where the TPC-C benchmarks describes that the benchmark should be ran with a number between 1 and 10 warehouses. One copy was stored using DB2 and the other one using SQL Server. Both copies included all primary key indices because both DBMSs create those by default.

The DB2 Index Adviser was then run with the queries and updates, together with their expected frequencies, included in the TPC-C workload and the resulting indices were added to the DB2 database. Similarly, the SQL Sever Index Tuning Wizard was executed on an actual trace of the TPC-C workload that was run against the created SQL Server database and the resulting indices were added to the SQL Server database. Next, the TPC-C workload ran for two hours against each of the databases.

Figure 7.1 shows the amount of main memory that was used by each of the experiments and the size of the recommended structures. Note that the size of the primary keys for the tables in the TPC-C benchmark is 200MB and the figure shows that total size of the created indices, i.e. DB2 suggested additional indices of size 661MB, while the SQL Server suggested additional indices of size only 6MB. Figure 7.2 shows the transactions/second for DB2 and Figure 7.3 shows the transactions/second for the SQL Server. During the experiments DB2 was able to perform 185 transactions/second, while the SQL Server - 349 transactions per second.

db2 our algorithm: 312.43 322.61 SQL our algorithm: 391.83 404.60

Next, we manually rewrote the TPC-C workload to reference only eSQL queries. This was an easy task because 92% of the TPC-C operations are directly expressible using the syntax for eSQL. We then manually broke the remaining 8% of the TPC-C queries into eSQL queries. This process involved introducing embedded C code. We then produced a set of compact structures and materialized views using the algorithm described in Section 5. Note that the approximate algorithm and the exact algorithm for cPART merging

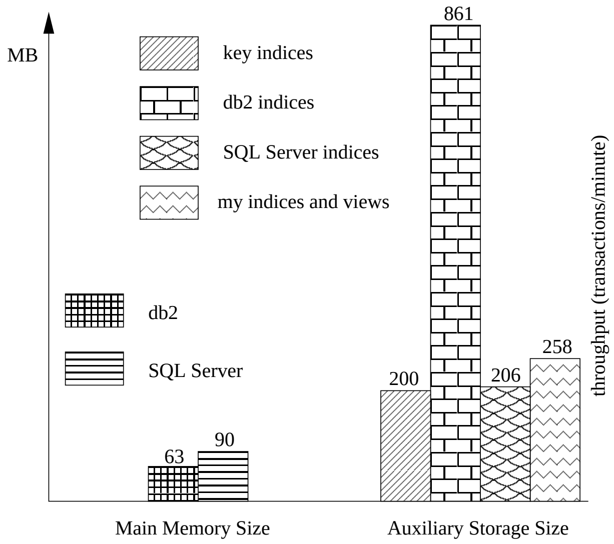

Figure 7.1: Main Memory and Auxiliary Storage Sizes

produced the same result for this workload. We then substituted each compact structure with a set of regular exogenous indices, where an index was created for each distinct path in the compact structure. Lastly, we ran the TPC-C workload using the DB2 and SQL Server using the described set of indices and MVs. Figure 7.2 shows the amount of main memory and auxiliary storage used by the algorithm. In particular, our solution used 58MB of auxiliary storage, were as DB2 used 661MB and the SQL Server used 6MB. The comparison of the results of the presented experiment with the results produced with the experiments with the physical designs suggested by DB2 and SQL Server are shown in Figures 7.2 and 7.3. Note that during the two hours of running this experiment averaged 312 transactions/minute when with the DB2 database and 392 transactions/minute when the SQL Server database. In other words, the physical design proposed by out algorithm outperformed the physical design produced by IBM DB2 by 69% and the physical design produced by the SQL Server by 12%. Moreover, unlike the physical designs suggested by the physical design advisors of the two commercial database, our solution guarantees logarithmic time execution of all the critical queries and updates in the workload.

Note that converting each compact structure into a set of indices underestimates the performance of our solution. The reason is that such an approach fails to recognize the

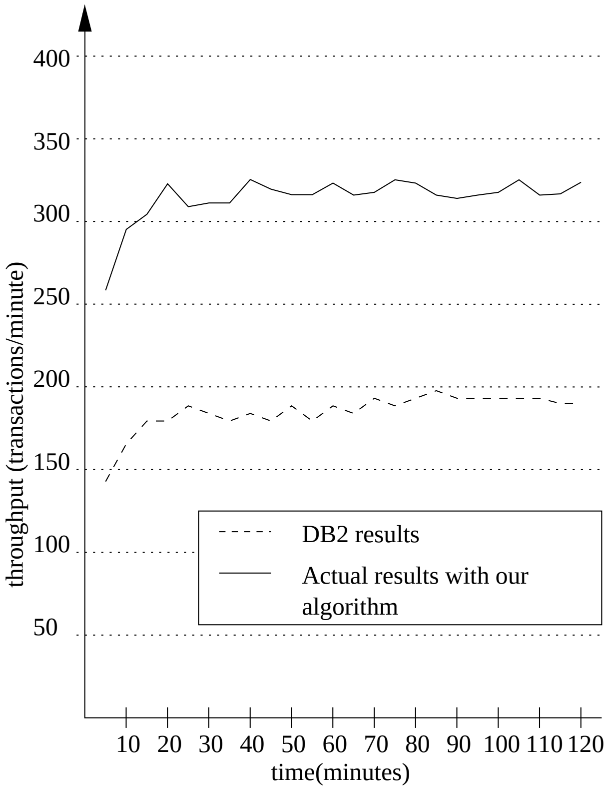

Figure 7.2: Performance Results for DB2 and Actual Results from our Algorithm

savings in update time as a result of storing the overlapping data between indices only once. For example, consider queries $Q3$ and $Q4$ from Table 1.5. An update to the table $\texttt{COMP}_\texttt{Y}\texttt{REMP\_RSTATS}$ will have to insert a single value in the suggested in Section 1.2 compact structure. On the other hand, if two indices are built for this compact structures, then two indices will have to be updated, which will take more time. In order to approximate by how much we have underestimated the performance of our algorithm, we have ran experiments to calculate how much time was spending in updated duplicated data that will not be duplicated if compact structures are created. The adjusted throughput in transactions/minute for DB2 and the SQL Server are shown in Figures 7.4 and 7.5.

Note that our estimates shows that we expect our solution to perform 323 transactions/minute using DB2 and 405 transactions/minute using the SQL Server. This means that we anticipate that our solution can outperform DB2 by 75% and the SQL Server by

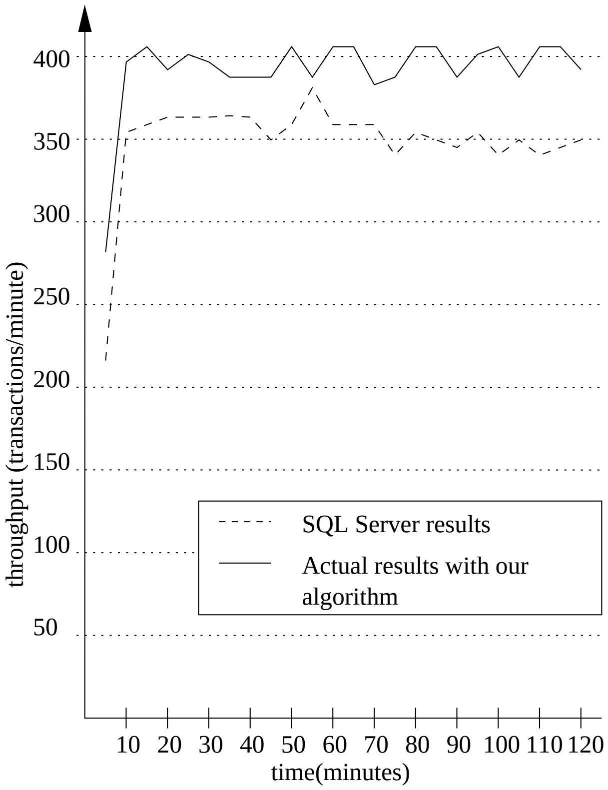

Figure 7.3: Performance Results for the SQL Server and Actual Results from our Algorithm

16%. More over, our solution uses the same amount of main memory as used by the two DBMSs, but we use 11.6 times less auxiliary storage than DB2 UDB and 9.7 times more auxiliary storage than the SQL Server, where the size of the key indices is not included in this computation.

Four observations about the presented experimental results are worth noting. First, our algorithm used less information, i.e. it did not used the query and update frequencies, and still produced a better solution than two of the leading database vendors. If our algorithm is modified to use query and update frequencies, then the performance gain could be even bigger. Our algorithm did not rely on frequency information because of our believe that database operation frequencies are not very stable for ECPs.

Second, this thesis introduces novel storage structures that are not implemented in any commercial database system. Therefore, performing a more precise experimental evalua-

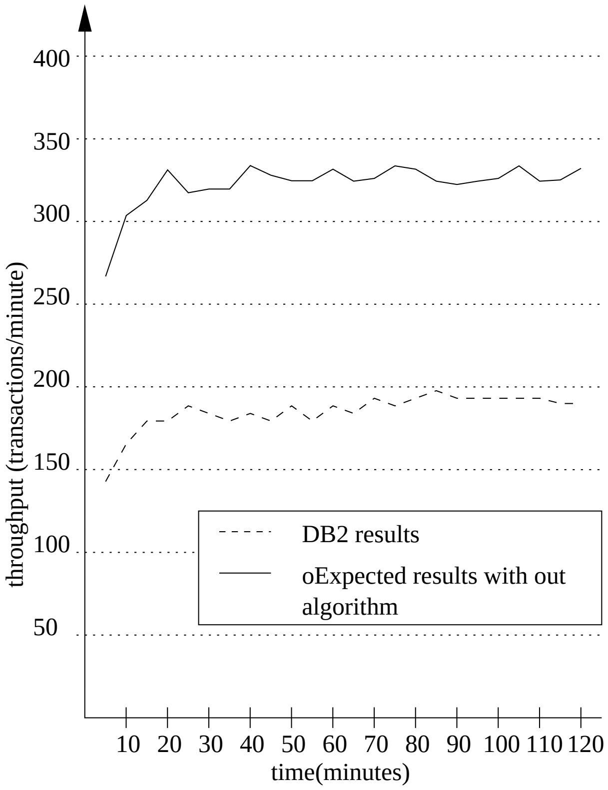

Figure 7.4: Performance Results for DB2 and Expected Results from our Algorithm

tion of our algorithm for compact structures (or for extended indices) is difficult without implementing such structures into existing systems.

Third, we did not present any experimental results using the extended index physical design model because almost all commercial database implementations use exogenous index structures. However, we believe that simulating the behavior of extended indices using regular exogenous indices is not a trivial task if at all possible.

Forth, ECPs are usually main-memory. We therefore believe that experimental results of the proposed structures with main-memory databases are needed to further validate the proposed in this work research. However, existing main-memory database systems are mostly proprietary and not very well established. We therefore believe that showing how our solution outperforms commercial systems like IBM DB2 and Microsoft SQL Server is more valuable than showing how the proposed system can outperform less known main-

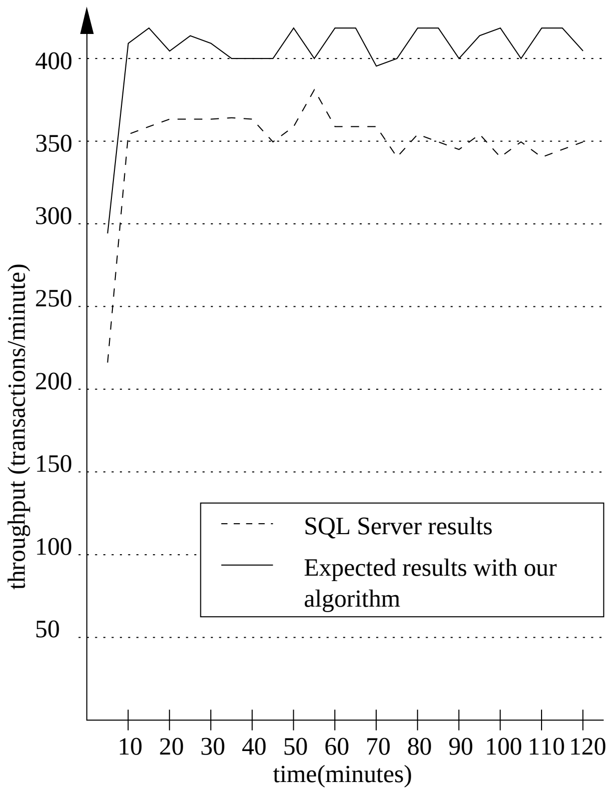

Figure 7.5: Performance Results for DB2 and Expected Results from our Algorithm

memory database systems.

Chapter 8

Conclusion and Future Research

8.1 Conclusion

We have formulated a problem that relates to automating physical database design for database systems deployed with embedded control programs, an important and emerging application area for relational database technology. The goal is to find as small as possible physical design that can efficiently support a predefined workload of critical queries and updates, where in this work we defined critical to mean logarithmic time. In Chapter 6 of the thesis we have shown that the admissible language for specifying critical operations is as general as possible in the presence of certain assumptions.

We solved the described problem by considering two different physical designs: an endogenous one consisting of extended indices and an exogenous one consisting of compact structures. When constructing these novel physical designs we tried to engineer them in a way that would allow for a maximal compaction. In particular, the proposed physical designs eliminate the need for storing the same information twice in some cases. This was done in a way that did not sacrifice performance for queries and even increased performance for updates. In general, the elimination of duplicate data does not always translate into performance gains and could even translate in performance losses - see [**?**].

In Chapters 4 and 5 of this thesis we showed that the problem of generating the smallest possible encoding that can be used to execute each of the defined critical database operations is NP-Hard for both physical designs models. Next, for each of the two physical design models, we presented exponential algorithms that solve the problem and polynomial time algorithms that approximate a solution. We also performed experiments with the compact structures encoding that show that our solution outperforms both DB2 UDB and the Microsoft SQL Server on the TPC-C benchmark. Moreover, unlike both products, our algorithm produces a physical design and query plans that *guarantee* efficient execution of all critical database operations in the workload.

8.2 Future Research

One of the shortcomings of the presented work is that the presented automating physical design system is static. In other words, it creates a physical design for a given set of transaction types and statistics. For an ECP the set of transaction types is relatively stable, i.e. it could only change when a new version of the software is installed. However, the database statistics that we used in choosing one physical design over another may not very stable, especially when it comes to statistics as the number of distinct values for an attribute or set of attributes - see **??**. We therefore believe that making the automating physical design system presented in this work dynamic is an important area for future research.

Another possible area for future research is considering the problem of creating SQL with QofS guarantees in a distributed server environment. A possible assumption for such a model is that the network between the different servers is dedicated. Novel problem such as data placement, devising parallel algorithms, and CPU utilization arise in this scenario.

Yet, a different possibility for future research is considering a problem in which time-bounds other than the logarithmic one are specified on database operations. For example, y-fast trees ([38]) and interpolation search trees ([16]) that could provide sub-logarithmic performance are worth exploring. A different set of physical designs models that is also worth considering include multi-valued indices and path function indices (see [6]). Another possibility is merging the extended index and compact structures physical design models and creating a hybrid physical design model. In particular, this model can identify leaf trees of compact structures for which the disjointness condition holds (i.e. sibling nodes are based on disjoint tables). Such trees can be substituted with extended indices. The result of this extension will be that extended indices that consist of more than one node could appear in the modified compact structure model.

Another possibility for future research is considering how the presented research could apply to applications in which the frequencies of the different transaction types are known. We believe that such extra knowledge could be used to significantly improve the quality of the presented solution. Note as well that our algorithm relies on a sound, but not complete integrity constraint reasoner. However, most knowledge based research (e.g. description logics research) has concentrated on sound and complete reasoners. Unfortunately, sound and complete algorithms for reasoning with integrity constraints become vastly undecidable as more types of integrity constraints are added to them. On the other hand, having an incomplete reasoner for a set of integrity constraint types could improve the quality of our solution more than having no reasoner at all.

Finally, recall that we restrict the query language for critical queries within a dialect of conjunctive SQL. Extending this language, e.g. allowing the addition of `union` operations and `where exists` clauses, is another area that we believe is worth exploring. Also, note that examined language for defining critical queries, i.e. μSQL, is the biggest possible set within eSQL relative to some assumptions. One of those assumptions is the lack of integrity constraints. For example, we believe that in the presence of functional dependencies the μSQL query set can be significantly extended without sacrificing its logarithmic time-bound performance and without going beyond the eSQL query set.

Appendix A

Summary of Notation

a natural number constant

b natural number constant

c natural number constant

d natural number constant

e alias for a table in a SQL query

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

i variable over the set of natural numbers

j variable over the set of natural numbers

k natural number constant

l natural number constant

m natural number constant

n node in a graph

n^r root node in a tree

p natural number that is used to denote the value of an `ID` attribute

q natural number

r natural number constant

s natural number constant

t tuple in a relation or the corresponding record

u natural number constant

v variable over the set of natural numbers

w Boolean variable

x variable over the set of natural numbers

y variable over the set of natural numbers

z variable over the set of natural numbers

A attribute of a table

B attribute of a table

C attribute of a table

D attribute of a table

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

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

G graph

G^t tree

H hash structure

I interpretation in a description logics sense

J compact structure

K path in a graph

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

M attribute of a table

N natural number

O order description

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

P query parameter

Q query

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

Q_P a query plan for the query Q

R relation

S attribute of a table

T table

U update

U_P an access plan for the update U

V materialized view

W linked list

X extended index

Y artificial index

$\mathcal{A}$ an element of a set

$\mathcal{B}$ set of binary boolean operations

$\mathcal{C}$ set of comparison operations

$\mathcal{D}$ physical design

$\mathcal{I}$ database instance

$\mathcal{P}$ PART

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

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

$\mathcal{O}$ set of binary operations

$\mathcal{S}$ predicate system

$\mathcal{V}$ set of variables

$\theta(\ldots)$ valid join condition

γ efficient predicate

κ a non-deterministic function that rewrites an ordering label

δ a primitive update

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

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 and `ID` if $n_1 \equiv n_2$

π projection in relational algebra

π^d duplicate preserving projection in relational algebra

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

σ selection in relational algebra

Σ database schema

Δ domain of objects

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

agg constant in the set $\{\texttt{min}, \texttt{max}, \texttt{sum}, \texttt{count}, \texttt{avr}\}$

$|\cdot|$ size of the enclosed component

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

$|def(\cdot)|_t$ number of underlying tables referenced in the definition of the enclosed component

ID a tuple identifier or a physical design identifier

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

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

$\texttt{Attr}^{\texttt{r}}(R)$ the reference attributes of the table R

$\texttt{Attr}^{\texttt{s}}(R)$ the standard attributes of the table R

$\texttt{Attr}(Q, R)$ the set of attributes in $\texttt{Attr}(R)$ that are referenced in the query Q

$\texttt{Attr}^{\texttt{r}}(Q, R)$ the set of attributes in $\texttt{Attr}^{r}(R)$ that are referenced in the query Q

$\texttt{Attr}^{\texttt{s}}(Q, R)$ the set of attributes in $\texttt{Attr}^{s}(R)$ that are referenced in the query Q

$\texttt{parent}^G(n)$ $\{n' :$ there is a directed edge from n' to n in the graph $G\}$

$\texttt{ancestor}^G(n)$ $\{n' :$ there is a directed path from n' to n in the graph $G\}$

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

$\texttt{child}^G(n)$ $\{n' :$ there is a directed edge from n to n' in the graph $G\}$

$\texttt{descendent}^G(n)$ $\{n' :$ there is a directed path from n to n' in the graph $G\}$

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

$[i \ldots j]$ the set of integers that are $\geq i$ and $\leq j$

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

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

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

$R_1 \subseteq_{\texttt{ID}} R_2$ iff $\pi_{\texttt{ID}} R_1 \subseteq \pi_{\texttt{ID}} R_2$

$R_1 \equiv_{\texttt{ID}} R_2$ iff $\pi_{\texttt{ID}} R_1 \equiv \pi_{\texttt{ID}} R_2$

Bibliography

[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] F. Baader and P. Hanschkle. A scheme for integrating concrete domains into concept languages. *Twelfth International Conference on Artificial Intelligence*, pages 425–457, August 1991.

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

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

[6] Elsa Bertino and Won Kim. Indexing Techniques for Queries on Nested Objects. *IEEE TKDE*, 1(2):196–214, 1989.

[7] Albert Croker and David Maier. A Dynamic Tree-Locking Protocol. *Proceedings of the Second International Conference on Data Engineering*, pages 49–56, 1986.

[8] Eugene Shekita David Simmen and Timothy Malkemus. Fundamental Technique for Order Optimization. *ACM SIGMOD*, pages 57–67, 1996.

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

[10] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *TODS*, 4(3):315–344, 1979.

[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] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company, 1972.

[14] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, pages 3–18, June 1995.

[15] Jonathan J. King. *Query Optimization by Semantic Reasoning.* UMI Research Press, 1981.

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

[17] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1998.

[18] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM TODS*, March 2000.

[19] P. Larson. Dynamic Hashing. *BIT*, 18(2):184–201, 1978.

[20] P. Larson. Linear Hashing with Partial Expansions. *VLDB*, 6:224–232, 1980.

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

[22] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. *VLDB*, pages 294–303, 1986.

[23] Huizhu Liu, David Toman, and Grant Weddell. Fine Grained Information Integration with Description Logics. *International Workshop on Description Logics*, 2002.

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

[25] Worm Mortensen. Fully-dynamic two Dimensional Orthogonal Range and Line Segment Intersection Reporting in Logarithmic Time. *Symposium on Discrete Algorithms*, pages 618–627, 2003.

[26] L. Mumick, D. Quass, and B. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. *SIGMOD*, 1997.

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

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

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

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

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

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

[33] David Toman and Grant Weddell. Attribute Inversion in Description Logics with Path Functional Dependencies. *Description Logic Workshop*, 2004.

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

[35] 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.

[36] Peter van Emde Boas. Design and Implementation of an Efficient Priority Queue. *Mathematics System Theory*, 10:99–127, 1977.

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

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

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

[40] www.wikipedia.org. *Wikipedia Encyclopedia*. GNU Free Documentation License, 2004.

[41] Jingren Zhou and Kenneth Ross. Buffering Access to Memory-Resident Index Structures. *VLDB*, pages 405–416, 2003.

[42] 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.

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