Index Selection for Compiled Database Applications in Embedded Control Programs

Lubomir Stanchev and Grant Weddell

School of Computer Science, University of Waterloo
e-mail: {stanchev, gweddell}@uwaterloo.ca

Abstract

A *compiled database application* is a collection of modules in a software system that interact with a common database through a set of predefined transaction types. We call a compiled database application an *embedded control program* (ECP) if it is reasonable to consider the execution time of each transaction type to be either critical or non-critical. Usually, the common database for an ECP is referred to as the *control data*. In this paper, we consider the index selection problem for the control data of an ECP. We believe this is a novel problem because of the presence of real-time requirements. Unlike the objective of earlier work in index selection that aims to reduce the response time of queries, ours is to reduce storage requirements in a way that ensures efficient execution time for the critical query and update workload. We propose a solution that abstracts and manipulates the result requirements of the query component of the critical workload. The experiments we have conducted show that this approach can produce small physical structures that support fast execution of a workload with many updates.

1 Introduction

A basic problem in database systems is selecting the best possible set of indices 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 [3, 6] and Microsoft SQL Server [1, 2], the problem is formulated as an optimization problem in which the execution time of the queries is minimized subject to a fixed storage overhead for indexing. The problem, formulated as such, is similar to the knapsack problem[1] and greedy algorithms are commonly used to solve it.

However, we believe there is an application area for database technology that requires an alternative approach to the problem. Consider for example the Linux kernel. It can be viewed as a *compiled database application* consisting of a collection of modules that interact with a common database of information about processes, open files, etc. This interaction can be characterized in terms of a predefined collection of transaction types. We call a compiled database application an *embedded control program* (ECP) if it is reasonable to consider the execution time of each transaction type to be either critical or non-critical, and we refer to the common database of an ECP as the *control data*. The interaction between an ECP and its control data is usually implemented using a low level language such as `C`. An alternative that

[1] It is not identical to the knapsack problem because the decision to create one index can influence the benefit of creating other indices.

would benefit the software designer is to substitute this interaction with static OQL calls to a database that stores the control data. It is reasonable to assume that soft real-time requirements will be associated with the static OQL code that is part of a critical transaction type. As a result, the static OQL calls can be characterized into three types:

- critical queries,
- non-critical queries, and
- updates.

For example, a query that lists the files that have been opened by a specific process could be considered non-critical, while a query that finds unused process IDs is more appropriately classified as critical.

Note that we assume all updates are critical. For example, consider the Linux `fork` command, which creates a new process. The updates performed by the operation should be critical. This is because the operation that lists the unused process IDs will be blocked while the `fork` command is executing.

Our approach extends the research published in [7] in which an algorithm for finding the fewest number of indices that can be used to support fast execution of the critical queries in a workload is described. Specifically, we describe an algorithm for finding a physical design of small size that not only supports fast execution of the critical queries, but also allows for the changes from all updates to be propagated quickly to the proposed physical structures. These structures can include B^+ trees, hash indices, doubly linked lists and various combinations.

An overview of the proposed algorithm is shown in Figure 1. Note that throughout the paper we use an object-relational data model. The reason for this decision is that this model has rich semantics, which allows for applying extensive semantic optimizations in the proposed algorithm. The input of the algorithm consists of a set of critical queries together with the classes and class attributes to which updates can be applied. Step 1 of the algorithm rewrites the critical queries by simplifying queries containing path functions in their `where` conditions and by simplifying the `select` parts of queries. In Step 2, the access requirements of each simplified query is abstracted as a either a binding pattern or as a *parameterized access requirement type* (PART). In Step 3, the identified PARTs are merged using a greedy algorithm and by exploiting schema information such as integrity constraints. In the last step, the physical design that corresponds to the identified binding patterns and merged PARTs is created.

INPUT:
1. Critical queries: $\{Q_1, ...Q_r\}$,
2. Class update constraint: $\{C_1, ..., C_s\}$,
3. Attribute update constraint: a subset of $\{C_i : A_j\}$, and
4. Schema information.

INDEX SELECTION:
1. Simplify each Q_i as follows:
(a) Decompose path expressions; and
(b) Partition each query into a set of *object search queries* and a set of *attribute access queries*. Object search queries return a list of the `ID`s' of the objects that match the search criteria. The attribute access queries return attribute values for an object with a specified object `ID`.
2. Construct an initial *parameterized access requirement type* (PART) for each object search query (PARTs eventually become index structures).
3. Merge PARTs in order to reduce storage requirements. (Information from the database schema can help in this process.)
4. Build the output physical design as follows:
(a) Select database record layouts according to the attribute access queries.
(b) Modify the database schema and define indices according to the PART requirements.

Figure 1: An overview of the algorithm

Note the differences between our approach and that taken by most commercial DBMS implementations. In systems like IBM DB2 UDB and Microsoft SQL Server the best possible set of indices is found for a given schema, workload and database population statistics, within a specified amount of storage space. In contrast, our algorithm does not have storage requirements, and instead tries to find a solution with the minimum store overhead. Our solution is also constrained by the requirement that it should be possible for all updates to be efficiently propagated to the proposed physical structures. Also, our algorithm does not take into account the exact frequencies of the queries and updates in the workload. Instead, it partitions queries as critical and non-critical. Another difference is that the proposed algorithm explores uniting index structures containing common data, which, to the best of our knowledge, is not done in the cited commercial DBMS products.

In what follows, in Section 2 we introduce the database and query models we will be using. In Section 3 we formally define the physical design problem we will be solving and in Section 4 we present our solution. In Section 5 we do an overview of the experiments we have conducted and lastly, in Section 6, we summarize the presented work and outline directions for future research.

2 The Database and Query Models

We use an object-relational model to describe the data. An example of a company schema is shown in Figure 2.

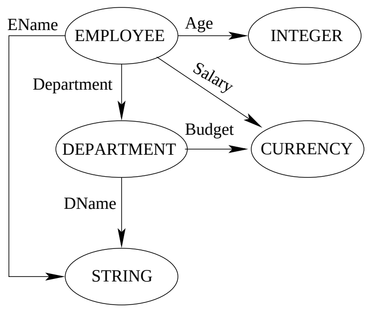

Figure 2: Example company schema

Classes, inheritance and class attributes are the building blocks of such a schema, where attributes can reference primitive types or other classes. In order to keep the model simple, we don't allow for multiple inheritance and for attributes with multiple or complex domains. In other words, if C is a class, t is an object belonging to C and A is an attribute of C, then $t.A$ denotes a single object or a primitive constant. As well, if C_1, C_2 and C_3 are classes, then $C_1 \Rightarrow C_2$ and $C_1 \Rightarrow C_3$ implies that either $C_2 \Rightarrow C_3$ or $C_3 \Rightarrow C_2$ must be true, where $C_1 \Rightarrow C_2$ is used to denote that C_1 inherits C_2. Primitive types include integers, strings, currency, etc. We will also include integrity constraints as part of our schema model. In this paper, we only consider integrity constraints of the form $C(\bar{X} \to \bar{Y})$, which denotes that for the class C, the sequence of attributes $\bar{X}$ functionally determines the sequence of attributes $\bar{Y}$.

The query language that we will be using is parameterized OQL. For example, consider the following OQL query:

```
select E.ID
from EMPLOYEE E
where E.Department.Budget
between :L and :H
order by E.Age asc.
```

It returns the identifiers for all employees who work for a department having budget between the value of the parameters `L` and `H` ordered by age and starting with the youngest. Note that all objects have the system attribute `ID`, which uniquely identifies them.

3 The Problem

We are given as input a database schema Σ, a set of critical queries $\{Q_i\}_{i=1}^{r}$ over Σ, a subset $\{C_i\}_{i=1}^{s}$ of the classes in Σ, a subset of the set $\{C_i :: A_j\}_{i=1,}^{c}{}_{j=1}^{a}$, which denotes the attributes in Σ (we have used $C_i :: A_j$ to denote the attribute A_j of the class C_i) and statistics about the size and distribution of the data. The queries $\{Q_i\}_{i=1}^{r}$ are *simple queries* and are of one of the three types shown in Table 1.

We have used $Attr(C)$ to denote the attributes of the class C and $PFs(C)$ to denote the path functions that start from the class C, i.e. $Pf \in PFs(C)$ iff $C.Pf$ is "well defined". The classes $\{C_i\}_{i=1}^{s}$ are the set of classes for which objects can be created or destroyed and are used to define the *class update constraint.* The subset of the set $\{C_i :: A_j\}_{i=1,}^{c}{}_{j=1}^{a}$ lists the attributes, which value could be modified and forms the *attribute update constraint.*

(*type*)	(*query format*)
(1)[†]	`select` $x.B_1$, ..., $x.B_n$ `from` $C\ x$ `where` $x.B_{n+1} = :P_1$ `and` ... `and` $x.B_{n+m} = :P_m$ [`order by` $x.B_1\ dir_1$, ..., $x.B_p\ dir_p$]
(2)[†]	`select` $x.B_1$, ..., $x.B_n$ `from` $C\ x$ `where` $x.B_{n+1} = :P_1$ `and` ... `and` $x.B_{n+m} = :P_m$ `and` $x.B_1$ `between` $:P'_m$ `and` P''_m [`order by` $x.B_1\ dir_1$, ..., $x.B_p\ dir_p$]
(3)[‡]	`select` $x.B_1$, ..., $x.B_n$ `from` $C\ x$ `where` $x.Pf = :P$

[†] $\{B_1, ..., B_{n+m}\} \subseteq Attr(C)$, $0 \leq p \leq n$, $dir_i \in \{$`asc`, `desc`$\}$
[‡] $\{B_1, ..., B_n\} \subseteq Attr(C)$, $Pf \in PFs(C)$

Table 1: A simple query can be of one of these 3 types

The goal is to find a set of structures that can support fast answering of the critical queries, can be updated quickly and don't take much space. By *fast answering* of the critical queries we will mean response time that is $O(|Result(Q)| + \log |D| * |Q|)$ for each query Q, where $|Result(Q)|$ denotes the result of the query Q, $|Q|$ denotes the size of the definition of the query Q and $|D|$ denote the size of the database. By *fast update* of the created structures we will mean that an insert or a delete of an object or modification of the values of its attributes should take at most $O(\log |D| * |\Sigma|)$ time. We will only consider plain record structures (e.g. doubly linked lists), conventional and distributed index structures (e.g. B^+ trees, hash indices, distributed B^+ trees) and combinations of them when building the physical design.

4 The Proposed Solution

We propose a solution consisting of four steps (see Figure 1). In the first step, the input queries are simplified and broken into even simpler queries, where possible. In the next steps, a suitable physical design for the simplified queries is chosen.

4.1 Step 1. Query Simplification

In this step we rewrite the simple queries. Recall that a simple query can be of one of the three query types shown in Table 1. A rewritten simple query will be of one of the three query types shown in Table 2.

This step consists of two parts. The first part involves rewriting simple queries containing path functions into ones that don't. Suppose we are given a query of type 3 and $Pf = A_1.A_2.\ ...\ .A_k$. Then, if we know that none of the attributes forming the path function can be updated, we can substitute Pf with A in the query and create an attribute A in the schema having the property $(\forall c \in C)(c.A = c.Pf)$. Such a constraint can be maintained efficiently because no updates to the attributes along the path function Pf are allowed. Next, suppose an update to the attribute A_r is allowed, $2 \leq r \leq k$. Then, we can break a simple query of type 3 (see Table 1) into the queries:

select $x.B_1$, $x.B_2$, ..., $x.B_n$
from $C\ x$
where $x.A_1.A_2.\ ....A_{r-1} = :P$

and

select x.ID

(*type*)	(*query format*)
(1)[‡]	select x.ID from C x where $x.B_1 = {:}P_1$ and ... and $x.B_n = {:}P_n$ [order by $x.B_{n+1}$ dir_1, ..., $x.B_{n+p}$ dir_p]
(2)[‡]	select x.ID from C x where $x.B_1 = {:}P_1$ and ... and $x.B_n = {:}P_n$ and $x.B_{n+1}$ between $:P'_n$ and $:P''_n$ [order by $x.B_{n+1}$ dir_1, ..., $x.B_{n+p}$ dir_p]
(3)[‡]	select $x.B_1$ from C x, where x.ID$= {:}P$

[†] $\{B_1, ..., B_{n+p}\} \subseteq Attr(C)$, $dir_i \in \{$asc, desc$\}$
[‡] $B_1 \in Attr(C)$

Table 2: A rewritten simple query can be of one of these 3 types

```
from $C'_1$ $x$
where $x.A_r...A_k$ = :$P$,
```

where C'_1 is a newly introduced class for the schema that is a subclass of the class reached by following the path function $A_1.A_2....A_{r-1}$ starting from the class C (we will refer to this class as C_1) and the objects in C'_1 can be described by the following query:

```
select $x$.*
from $C_1$ $x$, $C$ $y$
where $y.A_1.A_2. ....A_{r-1} = x$.
```

The following example demonstrates an application of the first part of Step 1 to a query referencing to the example company schema from Figure 2.

Example 1 *Consider the following query and suppose that updates to all mentioned classes and attributes in the query are allowed:*

```
select E.ID
from EMPLOYEE E
where E.Department.Budget = :P
```

We can rewrite this query into the following simple queries, which we will refer to as Q_d and Q_e respectively:

```
select D.ID
from SP_DEPARTMENT D
where D.Budget = :P
```

and

```
select E.ID
from EMPLOYEE E
where E.Department = :P.
```

The original query can then be executed using the query plan:

```
for $d$ in $Q_d(:P)$
  for $e$ in $Q_e(d)$
    return $e.ID$.
```

Note that the simple queries are based on the modified schema shown in Figure 3, containing the new class SP_DEPARTMENT. *In the figure we have used a thick arrow to denote inheritance. The objects in the class* SP_DEPARTMENT *correspond to the result of the following query:*

```
select D.DName, D.Budget
from DEPARTMENT D, EMPLOYEE E
where D=E.Department.
```

The second part of this step involves rewriting the queries so that they have a single select attribute. For example, a query of the form: "select $x.A_1$, $x.A_2$, ... $x.A_k$ from C x where *conditions* order by *attribs*" is rewritten into k+1 queries, where the first query is

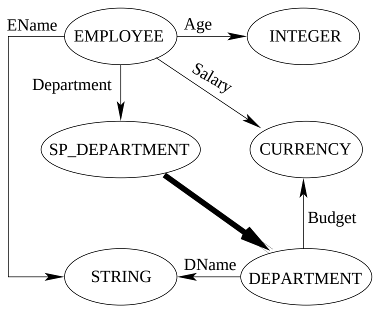

Figure 3: A modified schema for the simplified workload

an *object search query* of the form "`select` x.ID `from` C x `where` *conditions* `order by` *attribs*" and the i^{th} new query, where $i > 1$, is an *attribute access query* of the form "`select` A_{i-1} `from` C x `where` x.ID $= :P$".

4.2 Step 2. Abstracting Query Requirements

In this step we look at the rewritten simple queries and abstract the requirements of the attribute access queries as binding patterns and of the object search queries as PARTs. We start by formally introducing the concept of access requirements.

4.2.1 Defining Access Requirements

Before we formally define what we mean by an access requirement, we will first need to define its building blocks, i.e. a binding pattern and an order description.

Definition 1 (binding patterns) *Given a class C over a schema Σ, we define a binding pattern for C as a mapping of the elements of a subset of the attributes of C to the elements of the set* {`f b r`}. *A binding pattern describes the capability of retrieving the values for the attributes that map to* `f` *for objects for which each of the attributes that maps to* `b` *has some fixed value and the value for each of the attributes that maps to* `r` *is in some specific range.*

Example 2 *In the example company schema, the binding pattern* `EMPLOYEE(EName : b, Age : r, ID :` f) *describes the capability of retrieving the* `ID`*s of all employees with a given name and in a specified age group.*

Binding patterns are good for describing simple parameterized queries without an `order by` clause. Since we also care about ordering, we define order descriptions.

Definition 2 (order descriptions) *Given a class C over a schema Σ, we define an order description for C as a mapping from the elements of a subset of k attributes of C to a pair $(i, order_i)$, where $i \in [1..k]$, $order_i \in [0, 1]$ and i takes all values in its range. An order description describes the ordering of the objects belonging to the class C, where objects are ordered first according to the value of the attribute that maps to $(1, order_1)$ in ascending order if $order_1 = 0$ and in descending order if $order_1 = 1$, second accordingly to the attribute that maps to $(2, order_2)$ and s.o.*

Example 3 *In the example company schema the order description* $\texttt{Q}_1$(`EName :` $(1, 0)$, `Age :` $(2, 1)$) *describes that the result of executing* $\texttt{Q}_1$ *is ordered first according to* `EName` *in ascending order, and next, if more than one employees have the same name, then they are ordered according to their age, starting with the oldest.*

Note that, given a simple rewritten query Q (see Table 2), it is straight forward to find its binding pattern and order description. The way, in which this can be done, in shown in Table 3. We will refer to a binding pattern - order description pair as an *access requirement.* The reason for choosing this name is because such a pair describes the characteristics of the access plan that computes the result of a query.

Note however that a single physical structure, such as a B^+ tree index, can be used to answer queries with different access requirements. This is why we will examine Parameterized Access Requirement Types (PARTs), and Access Requirement Types (ARTs), where every (P)ART describes a set of access requirements. It also true that every PART describes a set of ARTs. The physical structures that can be used to efficiently answer all the queries described by a (P)ART are presented in Section 4.4.

$(type)$	$(query\ format)$	$(order\ description)$
1	$C(B_1 : \texttt{b}, ..., B_n : \texttt{b}, \texttt{ID} : \texttt{f})$	$C(B_{n+1} : (1, dir_1), ..., B_{n+p} : (p, dir_p))$
2	$C(B_1 : \texttt{b}, ..., B_{n-1} : \texttt{b}, B_n : \texttt{r}, \texttt{ID} : \texttt{f})$	$C(B_n : (1, dir_1), ..., B_{n+p} : (p+1, dir_{p+1})$
3	$C(\texttt{ID} : \texttt{b}, B_1 : \texttt{f})$	$C()$

Table 3: Access requirements for the query types from Table 2

4.2.2 Finding PARTs and Binding Patterns

We start this subsection by formally defining the syntax and semantics of an ART. The semantic of an ART will be defined in terms of the set of access requirements it corresponds to.

Definition 3 (ARTs) *Given a schema Σ, an ART expression can be defined using the following grammar:*

$ART := C() \mid C(S) \mid C(S, E) \mid C(E)$
$S := A : (order, dir) \mid S,\ S$
$E := A : dir_1 \mid E,\ E$
order := integer
dir := 0, 1 or hash
dir_1 *:= 0 or 1*
A *:= attribute*
C *:= class of* Σ

It is also true that given an ART expression $I = C(A_1 : (order_1,\ dir_1),\ ...,\ A_{k-1} : (order_{k-1},\ dir_{k-1}),\ A_k : dir_k,\ ...,\ A_{k+r} : dir_{k+r})$, $\{A\}_{i=1}^{k+r}$ are attributes of the class C and $\{order_i\}_{i=1}^{k-1}$ is a permutation of the numbers 1 through $k-1$. As well, we will require that if $r = -1$, i.e. there is no E part in the ART expression, then there exists $p \in [1..k]$ such that $dir_i = hash$ for $i \in [1...p-1]$ and $dir_i \neq hash$ for $i \in [p...k-1]$. If $r \geq 0$ we will require that $dir_i \neq hash$ for $i \in [1..k-1]$.

We define $B_i = A_j$ and $D_i = dir_j$ if "$A_j : (i, dir_j)$" is in the S part of I and we also define $B_i = A_i$ and $D_i = dir_i$ for $i \in [k..k+r]$. Then I will be equivalent to the ART expression $C(B_1 : (1,\ D_1),\ ...,\ B_{k-1} : (k-1,\ D_{k-1}),\ B_k : dir_k,\ ...,\ B_{k+r} : dir_{k+r})$. The set of access requirements that correspond to an ART expression are shown in Table 4.

We are now ready to define PART expressions, where each object search query will be abstracted as such. Note the reason we defined ARTs was to make the definition of what constitutes a valid PART more lucid.

Definition 4 (PARTs) *Given a schema Σ, we define a PART expression using the following grammar:*

$PART := C(P)$
$P := S \mid C(S) \mid C(S,\ (?\ b\ C(P)\ ...\ C(P)))$
$\mid () \mid S,\ E$
$S := A : (order, dir) \mid S,\ S$
$E := A : dir_1 \mid E,\ E$
b *:= positive integer variable*
order := positive integer
| positive integer variable
A *:= attribute*
C *:= a class of* Σ
dir := 0, 1 or hash
dir_1 *:= 0 or 1*

One of the differences in the syntax of a PART and an ART is the introduction of the "?" construct. Its semantics is $(?\ b\ C_1(P_1)\ C_2(P_2)\ ...\ C_k(P_k)) = C_b(P_b)$, where the value of b can be an integer between 1 and k. As well, we will require that no variable can appear more than once in a PART expression.

Definition 5 (valid valuation for PARTs) *Let I be a PART expression and v a valuation for I that maps the variables in I to constants. Then $v(I)$ will have the form $C_1(S_1, C_2(..., C_k(S_k, E_k))..)$. We will say that v is a valid valuation for I iff C_i is a superclass of C_j for $1 \leq i < j \leq k$ and $J = C_k(S_1, S_2,\ ...,\ S_k,\ E_k)$ is a valid ART expression. If v is a valid valuation, the expression J is the ART expression that corresponds to the PART expression I under the valuation v, i.e. $v(I) = J$.*

Definition 6 (valid PART) *The PART expression I is valid iff for any valuation of the "b" variables in I (see Definition 4) there exists a valid valuation for the remaining variables.*

Example 4 *Let us examine the PART expression* `DEPARTMENT(DName` : $(a, 0)$, (? x `SP_DEPARTMENT(Budget` :

($type$)	($ART\ parameters$)	($access\ requirement$)	($comment$)
1	$k = 1,\ r = -1$	$C()$	
2	$k > 1,\ r = -1,\ p \leq k-1$ $D_i = hash,\ i \in [1, p]$ $D_i \neq hash,\ i > p$	$C(B_1 : \texttt{b}, ..., B_q : \texttt{b})$	$1 \leq q \leq k-1$
		$C(B_1 : \texttt{b}, ..., B_q : \texttt{b})$ $C(B_{q+1} : D_{q+1},\ ...,\ B_{q'} : D_{q'})$	$p \leq q < q' \leq k-1$
		$C(B_1 : \texttt{b},\ ...,\ B_{q-1} : \texttt{b},\ B_q : \texttt{r})$ $[C(B_q : D_q)]$	$p < q \leq k-1$
3	$k = 1,\ r \geq 0$	$C(B_1 : \texttt{b},\ ...,\ B_{q-1} : \texttt{b}, B_q : \texttt{b})$ $C(B_{q+1} : D_{q+1},\ ...,\ B_{q'} : D_{q'})$	$1 \leq q \leq q' \leq r+1$
		$C(B_1 : \texttt{b},\ ...,\ B_{q-1} : \texttt{b}, B_q : \texttt{r})$ $C(B_q : D_q,\ ...,\ B_{q'} : D_{q'})$	$1 \leq q \leq r+1$, $q-1 \leq q' \leq r+1$
4	$k > 1,\ r \geq 0$	$C(B_1 : \texttt{b},\ ...,\ B_{q-1} : \texttt{b}, B_q : \texttt{b})$ $C(B_{q+1} : D_{q+1},\ ...,\ B_{q'} : D_{q'})$	$k-1 \leq q \leq q' \leq k+r$
		$C(B_1 : \texttt{b},\ ...,\ B_{q-1} : \texttt{b}, B_q : \texttt{r})$ $C(B_q : D_q,\ ...,\ B_{q'} : D_{q'})$	$k \leq q \leq r+k$, $q-1 \leq q' \leq r+k$

Table 4: Access requirements described by the ART expression $C(B_1 : (1,\ D_1),\ ...,\ B_{k-1} : (k-1,\ D_{k-1}),\ B_k : D_k,\ ...,\ B_{k+r} : D_{k+r})$

$(b, 0)$) DEPARTMENT())) *defined over the example company schema. Then $a = 1$, $b = 2$ is the only possible valid valuation for those variables. If $x = 1$ then the PART expression will be evaluated to* DEPARTMENT(DName : $(1, 0)$, SP_DEPARTMENT(Budget : $(2, 0)$)), *which corresponds to the ART expression* SP_DEPARTMENT(DName : $(1, 0)$, Budget$(2, 0)$), *otherwise it will be evaluated to the ART expression* DEPARTMNET(DName : $(1, 0)$).

The purpose of this step is to describe the access requirements for each type of simple query. We will describe the requirements for the object search queries using PART expressions and for the attribute access queries using binding patterns. Note that attribute access queries don't need to be characterized by an order description because they always return a single value. The following theorem follows from the definition of a PART and a binding pattern.

Theorem 1 *Each of the rewritten simple query types from Table 2 corresponds to one of the query types shown in the left column of Table 5. Moreover, the requirements of each of these query types can be described using the PART expression or binding pattern shown in the right column of Table 5.*

4.3 Step 3. Merging PARTs

In this subsection we describe how PART expressions can be merged. The reason for this merging is to save space.

We start by explaining how PARTs can be merged. Suppose we have k PARTS: I_1, I_2, ..., I_k. Suppose as well that they don't share variables in common (if they do, we can use variable renaming to rewrite them so they don't). We will say that they can be united into a single PART iff it is true that $I_i = C_i(S_i,\ R_i)$, where S_i and R_i are strings, S_i contains no part of an E expression (see Definition 4), C_i are class names, S_i are not empty and there exists a valid variable valuation that makes the elements in the set $\{S_i\}_{i=1}^{k}$ identical, up to equating "hash" and "dir" attributes (see Definition 4). The result of merging I_1, I_2, ..., I_k is $C(S,\ (?\ x\ C_1(R_1)\ C_2(R_2)\ ...\ C_k(R_k)))$, where $C = \bigcup_{i=1}^{k} C_i$ and S is an expression with the least number of variables that is equivalent to every element of the set $\{S_i\}_{i=1}^{k}$ under some variable binding and x is a newly introduced variable with range $[1, k]$. Uniting a hash subexpression $A : (x,\ hash)$ and a dir subexpression $A : (x,\ dir)$ yields the subexpression $A : (x,\ dir)$, where A is an attribute and dir is equal to 0 or 1. Note that subexpressions of the form $A : (x,$

(*simple query*)	(*PART/binding pattern*)
`select` x`.ID from` C x	$C()$
`select` x`.ID from` C x `where` $x.B_1 =$:P_1 `and` ... `and` $x.B_n =$:P_n	$C(B_1 : (x_1, hash),\ ...,\ B_n : (x_n, hash))$
`select` x`.ID from` C x `where` $x.B_1 =$:P_1 `and` ... `and` $x.B_n =$:P_n `order by` $B_{n+1}\ dir_1,\ ...,\ B_{n+p}\ dir_p$	$C(B_1 : (x_1, hash),\ ...,\ B_n : (x_n, hash),$ $B_{n+1} : (n+1, dir_1),\ ...,\ B_{n+p} : (n+p, dir_p))$
`select` x`.ID from` C x `where` $x.B_1 =$:P_1 `and` ... `and` $x.B_{n-1} =$:P_{n-1} `and` $x.B_n$ `between` :P'_n `and` :P''_n `order by` $B_n\ dir_1$	$C(B_1 : (x_1, hash),\ ...,$ $B_{n-1} : (x_{n-1}, hash),\ B_n : (n, dir_1))$
`select` x`.ID from` C x `where` $x.B_1 =$:P_1 `and` ... `and` $x.B_{n-1} =$:P_{n-1} `and` $x.B_n$ `between` :P'_n `and` :P''_n `order by` $B_n\ dir_1,\ ...,\ B_{n+p}\ dir_{p+1}$	$C(B_1 : (x_1, hash),\ ...,\ B_{n-1} : (x_{n-1}, hash),$ $B_n : dir_1,\ ...,\ B_{n+p} : dir_{p+1})$
`select` x`.ID from` C x `where` $x.B_1$ `between` :P'_1 `and` :P''_1 `order by` $B_1\ dir_1,\ ...,\ B_{n+p}\ dir_{n+p}$	$C(B_1 : dir_1,\ ...,\ B_{n+p} : dir_{n+p})$
`select` $x.B_1$ `from` C x `where` x`.ID =` :P	$C($`ID : b`$,\ B_1 :$ `f`$)$

Table 5: Representing the rewritten queries from Table 2.

1) and A : (x, 2) can not be united. We will refer to $\{S_i\}_{i=1}^{k}$ as the *starting sequences* and to $\{R_i\}_{i=1}^{k}$ as the *remaining sequences.*

Note that if R_j is empty for some $j \in [1, k]$ then $C_j()$ can be substituted with $C'_j()$, where $C'_j = C_j - \bigcup_{i=1 \text{ to } k}^{i \neq j} C_i$ and $C_1 - C_2$ is used to denote the set of all objects that are in C_1 but not in C_2. We will refer to this rule as the *negation rule.* The reason this rule can be applied is because $C(S,\ (?\ x\ C_1(R_1)\ ...\ C'_j()\ ...\ C_k(R_k)))$ can be used to answer a query with access requirements corresponding to $C_1(S)$, ..., $C'_j(S)$, ..., $C_k(S)$ and therefore it can be used to answer a query with access requirements corresponding to $((\bigcup_{i=1 \text{ to } k}^{i \neq j} C_i) \cup C'_j)(S) = ((\bigcup_{i=1 \text{ to } k}^{i \neq j} C_i) \cup (C_j - \bigcup_{i=1 \text{ to } k}^{i \neq j} C_i))(S) = C_j(S)$.

Example 5 *Continuing with our example schema from Section 1, the PART expressions* `DEPARTMENT(DName` : $(1, a)$) *and* `SP_DEPARTMENT(Dname` : $(1, c)$, `Budget` : $(2, b)$) *can be united into the PART expression* `DEPARTMENT(Dname` : $(1, e)$, (? x `DEPARTMENT()` `SP_DEPARTMENT(Budget` : $(2, b)$))*. We can use the negation rule to rewrite the last PART expression as* `DEPARTMENT(Dname` : $(1, e)$, (? x `NSP_DEPARTMENT()` `SP_DEPARTMENT(Budget` : $(2, b)$))*, where* `NSP_DEPARTMENT` = `DEPARTMENT` - `SP_DEPARTMENT`.

There is a second rule, called the *functional dependency rule*, that can be applied to simplify a newly created PART expression. The rule states that if $C_i(\bar{A} \rightarrow B_{i,j})$ holds, where $\bar{A}$ are attribute of S and $B_{i,j}$ is an attribute in R_i and no part of its E part (see Definition 4), then the attribute $B_{i,j}$ can be removed from the R_i part of the newly created PART. The reason why this rule can be applied is because the functional dependency shows that there will be only one different value for $B_{i,j}$, when the values for the attributes in S are fixed, and therefore there is no need to index or hash this value. The following example demonstrates an application of this rule.

Example 6 *Suppose we are given the queries:*

```
select E.ID
from EMPLOYEE E
where E.Salary > :P1 and
E.EName = :P2
```

and

```
select E.ID
from EMPLOYEE E
where $E.EName =\ :P_1$.
```

As well, suppose that we know that the integrity constraint `EMPLOYEE(EName` $\rightarrow$ `Salary)` *holds. The PART expressions for the two queries will be* `EMPLOYEE(EName` : $(1,$ `hash`$)$, `Salary` : $(2,1)$) *and* `EMPLOYEE(EName` : $(1,$`hash`$)$). *When we merge them we will get the PART expression* `EMPLOYEE(EName` : $(1,$`hash`$)$, (? x `EMPLOYEE(Salary` : $(2,1)$), `EMPLOYEE`)). *Using the integrity constraint we can rewrite the PART expression as* `EMPLOYEE(EName` : $(1,$ `hash`$)$). *Then, the first query can still be answered, but using the query plan:*

```
for e in Q(:$P_2$)
  if $e.Salary >\ :P_1$ return e,
```

where Q is a query with access requirements corresponding to the simplified PART.

Now, we are ready to present our algorithm for uniting PART expressions. It goes as follows.

1. Cluster the PART expressions in different groups in such a way that PART expressions from different groups can not be merged because they don't share attributes in common. Apply the next steps to each group separately.

2. Find the starting attribute with the greatest benefit and merge all PART expressions for which there exists a valuation that evaluates them to valid ART expressions that start with that attribute. Suppose the expressions $I_1 = C_1(...)$ ... $I_k = C_k(...)$ are the ones for which there exists a valuation that evaluates them to ART expressions that all start with the attribute A. Then we calculate the benefit of A as $(\sum_{i=1}^{k} |C_i(A)|) * |A|$, where $|C_i(A)|$ denotes an approximation of the number of different values for the attribute A for objects of type $|C_i|$ in the database and $|A|$ denotes the size of the encoding for the attribute A.

3. Simplify the created in the previous step PART expression. First, apply the negation and then apply the functional dependency rule.

4. Unite the PART expressions that under some valuation start with the attribute with the second greatest benefit and then, in the same way, repeat the procedure for the remaining attributes.

5. At this step look at all PART expressions that have been created in the previous steps and apply this algorithm to their remaining sequences. For example, given a PART expression $C(A : (1, \_)$, ? x $C_1(P_1)$ ... $C_k(P_k))$, where "_" stands for don't care, apply the algorithm to the set of PART expressions $\{C_i(P_i)\}_{i=1}^{k}$ and use the result to rewrite the original PART expression. In our example, if the algorithm rewrites $\{C_i(P_i)\}_{i=1}^{k}$ into $\{C_i^{'}(P_i)\}_{i=1}^{m}$, then the original PART expression will be rewritten as: $C(A : (1, \_)$, ? x $C_1^{'}(P_1)$ ... $C_m^{'}(P_m))$.

It is easy to see that the complexity of the above algorithm is $O(p * n * log(n) * r)$, where n is the number of input PART expressions, p is the number of attributes in the schema and r is the length of the longest PART expression.

4.4 Step 4. Building the Physical Design

So far we have described how to create a set of PART expressions corresponding to the object search queries and how to create binding patterns corresponding to the attribute access queries. In this section we first define what does it mean for a physical design to "efficiently support" a workload, given a database schema and a description of possible updated. Next, we describe how to create a physical design that efficiently supports the attribute access queries and then we continue by extending the physical design to also efficiently support the produced in the previous sub-sections PARTs.

Definition 7 (supporting a workload) *Suppose we are given a database over a database schema Σ, a set of classes $\{C_i\}_{i=1}^{a}$,* *objects to which can be added or removed through updates, a set of attributes $\{C_j :: A_i\}_{i=1}^{a}, {}_{j=1}^{c}$ that can be modified and a critical workload W. We will say that a physical design efficiently supports W iff every query Q in W can be answered in time $O(|Result(Q)| + \log|D| * |Q|)$, where D denotes the size of the database, and, at the same time, any allowed update to the database can be propagated to the physical design in $O(\log|D| * |\Sigma|)$ time.*

4.4.1 Supporting Attribute Access Queries

Suppose in Step 2 of the algorithm we have identified the set of binding patterns $\{O_i\}_{i=1}^{o}$, which correspond to the attribute access queries of type 3. Draw a directed graph where each binding pattern is represented by a node and there is a directed edge between $C_1(\texttt{ID} : \texttt{b}, A_1 : \texttt{f})$ and $C_2(\texttt{ID} : \texttt{b}, A_2 : \texttt{f})$ iff C_2 is a superclass of C_1. Initially, each node is labeled with the corresponding class name and free attribute of the binding pattern it represents. We next unite nodes that have edges between them in both directions into a single node. The new node will have the class name of the nodes that are merged (those classes will be identical) and attribute list corresponding to the union of the attributes of the two nodes. Since we don't allow for multiple inheritance (see Section 2), the so created graph will be a forest. Examine each tree starting with the leaves and for each leaf node create a doubly linked list of the objects from the class in the node label, and store for each object only the attributes specified in the node. Going up the three, create doubly linked lists of records for all objects from the corresponding class and with the corresponding attributes, but create records only for objects which haven't already been stored. It is easy to see that the so created design efficiently supports the partial workload consisting of the attribute access queries, regardless of the allowed updates, as long as there exists an efficient way to identify the record corresponding to the object that is being updated or deleted.

4.4.2 Supporting PART Expressions

We start by defining a language for describing conventional physical structures (CPSs). Those will be the atomic building blocks with which we will extend the design from the previous subsection to support the merged PARTs.

Definition 8 (CPSs) *We define a CPS using a language over the following grammar:*

CPS ::= $C^{tree}(T)$ | $C^{hash}(H)$ | $C^{dtree}(B, T)$
| $C^{dhash}(B, H)$
T ::= $A : dir$ | T, T
H ::= A | H, H
A ::= *primitive attribute*
B ::= *non-primitive attribute*
dir::= *0 or 1*

Recall that primitive attributes are attributes with primitive domain such as integers, strings, etc. Non-primitive attribute are attributes with a domain containing non-primitive objects.

A CPS expression of the form $C^{hash}(A_1, ..., A_n)$ corresponds to a hash index on $(A_1, ..., A_n)$. Such a structure is designed to answer queries with a binding pattern $C(A_1 : \texttt{b}, ..., A_n : \texttt{b}, \texttt{ID} : \texttt{f})$ and no order description.

On the other hand, a CPS expression of the form $C^{tree}(A_1 : dir_1, ..., A_n : dir_n)$ describes a B^+ index tree, ordered by A_1 in direction dir_1, through A_n in direction dir_n. It is well known that such an index can be used to answer a query with binding pattern $C(A_1 : \texttt{b}, ..., A_{r-1} : \texttt{b})$, where $1 \leq r - 1 \leq n$, and no order description or order description $C(A_r : dir_r, ..., A_k : dir_k)$, where $r \leq k \leq n$. This index can also be used to answer range queries with binding pattern $C(A_1 : \texttt{b}, ..., A_{r-1} : \texttt{b}, A_r : \texttt{r})$, where $r < n$, and no order description or order description $C(A_r : dir_r, ..., A_k : dir_k)$, where $r \leq k \leq n$.

Thirdly, a CPS expression of the form $C^{dhash}(B, A_1, ..., A_n)$ describes a distributed hash index. A distributed hash index with these parameters contains a hash index for each different object in the set $\{t.B | t \in C\}$. Those hash indices are on the attributes $\{A_i\}_{i=1}^{n}$ and each hash index indexes all objects having the corresponding fixed value for the attribute B. Not surprisingly, a distributed hash index with

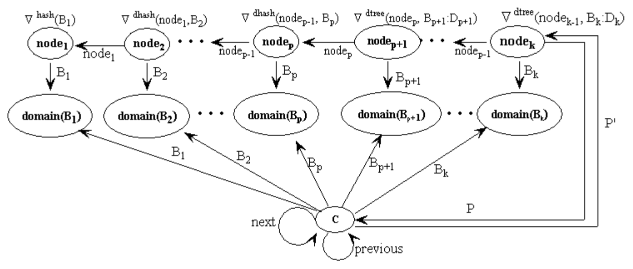

Figure 4: A schema for a physical design that supports an ART of type 2 (see Table 4)

the above parameters can be used to answer a query with binding pattern $C(B : \mathtt{b}, A_1 : \mathtt{b}, ..., A_n : \mathtt{b}, \mathtt{ID} : \mathtt{f})$*, assuming there is a way to identify the correct hash index, given a value for the* B *attribute.*

Finally, a CPS expression of the form $C^{dtree}(B, A_1 : dir_1, ..., A_n : dir_n)$ *describes a distributed* B^+ *tree index. Such an index is built from a set of* B^+ *trees, each ordered according to the attributes* A_1 *in direction* dir_1 *through* A_n *in direction* dir_n *and each containing all objects in* C *having a value for the* B *attribute equal to a value in the set* $\{t.B|r \in C\}$*. Such a distributed* B^+ *tree index can answer all queries with binding pattern* $C(B : \mathtt{b}, A_1 : \mathtt{b}, ..., A_{r-1} : \mathtt{b})$*, where* $1 < r - 1 \leq n$*, and no order description or order description* $C(A_r : dir_r, ..., A_k : dir_k)$*, where* $r \leq k \leq n$ *and all range queries with binding pattern* $C(B : \mathtt{b}, A_1 : \mathtt{b}, ..., A_{r-1} : \mathtt{b}, A_r : \mathtt{r})$*, where* $1 < r < n$*, and no order description or order description* $C(A_r : dir_r, ..., A_k : dir_k)$*, where* $r \leq k \leq n$*, assuming there is a way to identify the correct* B^+ *tree, given a value for the* B *attribute.*

Next, before showing how to create physical designs that efficiently support queries described by different PARTs, we show physical designs that support queries describing the four possible type of ARTs (see Table 4). After that, we will generalize the proposed physical structures to support PARTs.

The queries described by an ART of type 1 can be efficiently supported by adding a linked list of the IDs of objects of type C. However, such a physical design is already created in Section 4.4.1.

The queries described by an ART of type 2 can be efficiently supported by the physical design graphically depicted in Figure 4. In the figure we have used ∇ to denote a CPS and after the triangle we have specified the parameters of the CPS. The class $node_i$, $i \in [1..k]$, contains an object for all distinct values in the set $\{x.B_1, x.B_2, ..., x.B_i|x \in C\}$. The schema shown in Figure 4 is implemented by building the described in the figure CPSs and combining them. For example, each hash value of the hash index of $node_1$ will be a pointer to one of the hash indices in the distributed hash index on $node_2$ and s.o. Each hash value of the hash index of $node_k$ will be the start of a doubly linked list of the objects of type C. The later can be implemented by adding additional "next pointer" and "previous pointer" fields in the list of records of type C created in Section 4.4.1. In this way if an object is inserted, deleted or modified, the change can be propagated to the set of records created in Section 4.4.1 in constant time and can be propagated to each CPS that is affected in logarithmic time.

Similarly, the queries described by ARTs of type 3 and 4 can be efficiently supported by the physical designs shown in Figures 5 and 6. The fact that each of the described physical designs *efficiently* supports the queries for the corresponding ART expressions, regardless of the defined critical updates, can be easily verified. Indeed, any insertion or deletion from the described physical designs can be implemented by a constant number of insertions or deletions to CPSs and constant number of changes to the corresponding fields of the records created in Section 4.4.1. In tern, an insertion or a deletion to a CPS will have a logarithmic time bound. An update of an object's value can be done by performing a delete followed by an insert and therefore can again be done in the required time bound.

Next, we show how the set of queries describing a PART can be efficiently supported. Suppose we are given a PART expression I=$C(S$, $(?\ x\ C_1(P_1)\ ...\ C_k(P_k)))$ (see Definition 4). Then we can build the physical design corresponding to the ART expression S, where we substitute each pointer to a record of type C with a set of pointers to the physical structures T_i, where $i \in [1..k]$ and the physical structure

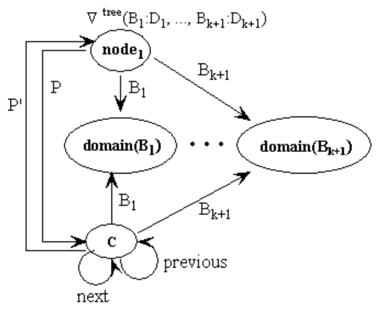

Figure 5: A schema for a physical design that supports an ART of type 3 (see Table 4)

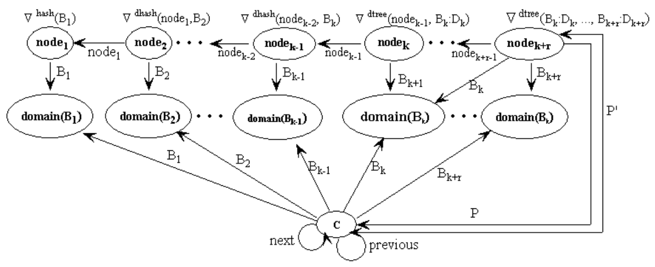

Figure 6: A schema for a physical design that supports an ART of type 4 (see Table 4)

T_i indexes all the objects in C_i that have the corresponding values for the attributes indexed in S according to the expression P_i. It is easy to see that each insert, delete or update to the so described structure consists of a constant number of inserts, deletes or updates to structures corresponding to ART expressions, which can be performed efficiently. Therefore, the so presented algorithm indeed builds a physical structure that efficiently supports the queries described by a PART expression.

This concludes our overview of the presented algorithm. The theorem stated bellow summarizes the results presented in this section.

Theorem 2 *The described in this section algorithm solves the problem stated in Section 3 by producing a solution of small size. The presented algorithm doesn't necessarily find the solution with the smallest possible size.*

In order to explain why producing an optimal solution to the problem is not needed, consider the following example.

Example 7 *Suppose we only have the PART expressions* $\texttt{C(A} : (x, 1)$, $\texttt{B} : (y, 1))$ *and* $\texttt{C(A} : (z, 1)$, $\texttt{B} : (w, 1))$. *When we unite them we can produce the PART expression* $\texttt{C(A} : (1, 1)$, $\texttt{B} : (2, 1))$ *or* $\texttt{C(A} : (2, 1)$, $\texttt{B} : (1, 1))$. *However, since we don't have exact information about how many different B values do we have on average for each different A value for the objects of type C, we can not discriminate between the two solutions. We can use statistical information to do so, however, such fine-grained data is rarely available.*

5 Experimental Results

We conducted a set of experiments based on the TPC-C benchmark [5]. This benchmark is based on an OLTP scenario. We have simplified the experimental setup by running the experiments on a single computer, rather than on a set of connected modules. In this way we have eliminated any variations in the results due to network congestion. The experiments were run on a PC with 1.5 GHz Intel Pentium IV CPU, 256 MB of main memory and 40GB hard disk with rotating speed of 7200 rpms and average seek time of 8.5 ms. The database on which the experiments were performed was IBM DB2 UDB v7, running on Microsoft Windows 2000.

We first extracted the workload in terms of the set of queries and updates, together with their expected frequencies, from the five transaction types included in the TPC-C benchmark. We presumed that all queries and updates are critical. Next, we supplied this workload to the IBM DB2 Index Advisor. The set of proposed indices consisted only of non-clustered indices; the reason is that IBM DB2 UDB automatically generates a clustered index for each table based on its primary key if such exists.

Next, we applied our algorithm to the extracted workload. Update and delete queries were rewritten as `select` queries in order to abstract their access requirements. Next, all queries that were not simple queries (see Table 1) were broken down into such. This involved creating access plans for them and abstracting parts of those plans as simple queries. When applying our algorithm, we only considered simple tree indices since they were the only index types suggested by the IBM DB2 Advisor software.

During the actual experiments, we ran three trials, one with only the default clustered indices based on the defined primary keys in the input tables, one using the recommended by IBM DB2 Index Advisor indices and one using the indices recommended by our algorithm. The initial state of the database was based on 10 warehouses and was of size 776 MB. Each trial was run for 2 hours. The results of the experiments are summarized in Figures 7 and 8.

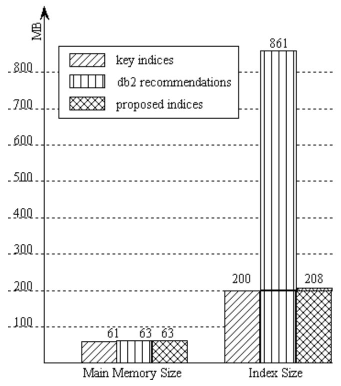

Figure 7: Main memory and index sizes

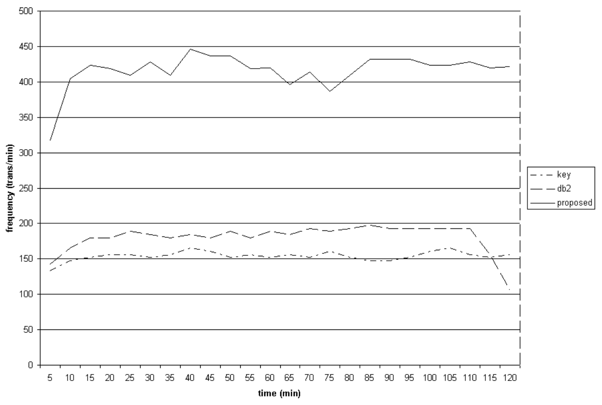

Figure 8: Throughput with different indices

The left part of Figure 7 depicts the amount of main memory used on average during the two hour run of the TPC-C workload in the three trials. The right part of the figure shows the size of the indices that were created for the three trials. The first trial was run with only the clustered indices that were created by default by IBM DB2 UDB. In the second trial, the indices recommended by IBM DB2 Index Advisor were also considered. The third trial was run with the indices produced by out algorithm. Their actual size is 11MB, but if we apply out technique for merging PARTs, their size can be reduced to 8MB. However, the trial was actually run with secondary indices of size 11 MB; the reason is that IBM DB2 UDB doesn't support the presented in the paper extended physical structures. Also, the third trial included the created by default primary keys of size 200MB since the IBM DB2 UDB software doesn't permit their removal.

Figure 8 shows the throughput in transactions per minutes during the trials. The throughput was measured every 5 minutes during the trials and is interpolated on the figure. The experimental results show that the proposed in this paper solution increases performance by a factor of 2.3, relative to the solution proposed by IBM DB2 Index Advisor. We believe the main reason for this improvement comes from the fact that IBM DB2 Index Advisor recommends the best indices for executing each individual query, without putting too much emphasis on the cost of index update.

6 Summary and Future Research

We have introduced an algorithm for index selection in the context of compiled database application that are ECPs. The algorithm produces an encoding with small size that ensures fast execution of the critical queries and updates in a given workload. The main differences between the presented problem and the classical problem of index selection include:

- The presented problem separates the input workload into critical and non-critical queries and does not consider the exact query frequencies;
- The presented problem finds physical design that supports fast execution of the critical queries and updates; and
- The presented problem tries to find the solution with the smallest size.

Those differences are a direct consequence of the challenges related to index selection for ECPs. We believe the proposed solution is applicable to commercial ECPs, including programs with stringent storage requirements, such as mobile applications. The reason for our confidence comes from the results of the experiments we have conducted. For example, based on the TPC-C benchmark, our solution requires 83 times less memory than the IBM DB2 Index Advisor solution and the quality of the solution is improved by a factor of 2.3.

Some of the techniques we have used to meet the challenges related to ECPs include the following.

- Our algorithm unites index structures containing common data. This not only reduces the size of the physical design, but also speeds up update to those structures.
- Our algorithm uses schema information to simplify the proposed solution.
- Our algorithm suggests index structures that have direct pointers to data records. In contrast, most commercial DBMS systems support indices containing ROW-IDs.

A list of areas of future research follows.

- Extending the algorithm to deal with arbitrary queries by pinning down a methodology for breaking up of an arbitrary query into simple ones.
- Extending our schema model to include multiple inheritance and richer integrity constraints including equality constraints and extended functional constraints (see [4]).
- Extending the solution space by including materialized view as part of the proposed physical structures.

About the Authors

Lubomir Stanchev is a Ph.D. student in the School of Computer Science at the University of Waterloo. His interests include physical database design, distributed databases and multiple query optimization.

Grant E. Weddell is an Associate Professor in the Department of Computer Science at the University of Waterloo. His research interests derive from the goal of enabling database technology for embedded control and network management applications, with a particular focus on fine-grained information integration and semantic query optimization.

References

[1] Surajit Chaudhuri and Vivek Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. *Proceedings of the 23rd VLDB Conference*, pages 146–155, 1997.

[2] Surajit Chaudhuri and Vivek R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. *SIGMOD*, pages 367–378, 1998.

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

[4] David Toman and Grant Weddell. On Attributes, Roles, and Dependencies in Description Logics and the Ackerman Case of Decision Problem. *Proc. Description Logics*, 2001.

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

[6] Gray Valentin, Michael Zulian, Daniel C. Zilio, Guy Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes. *Proceedings of the 16th International Conference on Data Engineering*, pages 101–110, February 2000.

[7] Grant Weddell. Selection of Indexes to Memory-Resident Entities for Semantic Data Models. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):274–284, June 1989.