Index Selection for Embedded Control Applications using Description Logics

Lubomir Stanchev and Grant Weddell
School of Computer Science, University of Waterloo
E-mail: {lp2stanc, gweddell}@uwaterloo.ca

Abstract

We consider the problem of automated index selection for *embedded control programs* (ECPs). Such systems have the property that the transaction types, which can consist of queries and updates, are predefined and can be classified as either *critical* or *non-critical*. In this paper, we focus on the critical part of the transaction workload for ECPs. More precisely, our problem input consists of a set of critical transaction types and a database schema. The goal is to find a minimum number of *extended indices* that enable every critical operation to be performed efficiently. The proposed solution is novel in that it entails the use of a description logic (DL) reasoner to find cases in which extended indices can be combined or simplified.

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` [9] and `Microsoft SQL Server` [2] the problem is formulated as an optimization problem in which the execution time of the input queries and updates is minimized subject to a fixed storage overhead. However, as suggested in [7], an alternative formulation of the problem is needed for *embedded control programs* (ECPs). First, since constraints on execution time can exist, the performance of the critical transaction types needs to be guaranteed. And second, since most ECP applications are main-memory, it is important to economize storage and create as few indices as possible. We therefore formulate the problem of *index selection for ECPs* (ISECP) as the problem of finding the smallest number of indices that can be maintained efficiently during updates and that can be used to efficiently answer each of the given critical queries. We formally define the ISECP problem and the terms *efficiently maintainable index* and *efficiently answering a query* in Section 2. It turns out that the ISECP problem is NP-Hard and that this complexity is independent of the complexity of the DL reasoner. We prove this fact and exhibit an exponential algorithm for solving the problem in Section 3.

In order to understand the challenges associated with the ISECP problem, consider the following example. Suppose we are given the company database schema depicted in Figure 1 and the set of OQL queries shown in Table 1. Suppose as well that we know that no person that is younger than 32 years of age can receive a salary in excess of

90K. Then any index from the family of so called extended indices depicted in Figure 2 can be used to efficiently answer all three queries. The indices depicted in Figure 2 all index the objects in the class PERSON. These indices are first sorted relative to the attribute *name* in some order, next relative to the attribute A in ascending order, and next, the sorting is dependent on the value of A. If $A = a_1$, then the remaining sorting is relative to the *salary* attribute in ascending order. If $A = a_2$, then the remaining sorting is relative to the *dep* attribute in some order followed by an ordering on the *rank* attribute in descending order. If $A = a_3$, then the remaining sorting is arbitrary. For now, let $Ob(C)$ denote the set of objects that belong to a given class C. Then, for an object $p \in Ob(\texttt{PERSON})$, attribute A is defined as follows: if $p.Age < 30$ and $p \in Ob(\texttt{EMPLOYEE})$ then $A = a_1$, if $p.Salary > 100K$ and $p \in Ob(\texttt{MANAGER})$ then $A = a_2$, otherwise $A = a_3$. We have used a_1, a_2 and a_3 to denote three distinct integers. The value for A is well defined because the first two cases are disjoint.

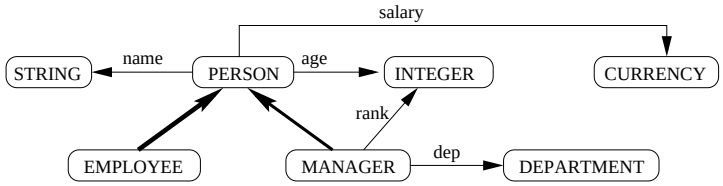
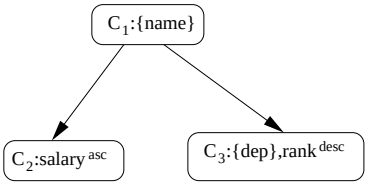


Figure 1: Example Company Schema.



$C_1 \equiv \texttt{PERSON}$
$C_2 \equiv \texttt{EMPLOYEE} \sqcap \forall age.(\lambda x)(x < 30)$
$C_3 \equiv \texttt{MANAGER} \sqcap \forall salary.(\lambda x)(x > 100K)$

Figure 2: A Family of Extended Indices that Supports the Example Queries.

1.1 Related Research

This work is based on two previous papers. In [10] a similar problem is described, but updates are not considered and only enhanced indices that are "paths" rather than "trees" are examined (i.e. the algorithm from [10] will create two, rather than one, extended indices when applied to our motivating example). More recently in [7] updates are considered, but unconventional indices in which the same element can appear more than once are produced. Although, such indices can save space and speed up performance, to the best of our knowledge they are not currently supported by any commercial database vendor.

2 The ISECP Problem

An ISECP problem consists of a database schema formulated as a *terminology* in a DL dialect called $\mathcal{DLFD}$ [8] and a workload formulated as a set of *critical operations*. A $\mathcal{DLFD}$ terminology, written as $\mathcal{T}$, corresponds to a set of *inclusion dependencies* of the

(*number*)	(*query*)
1	`select` x `from PERSON as` x `where` $x.name = :P$
2	`select` x `from EMPLOYEE as` x `where` $x.name = :P_1$ `and` $x.salary \geq :P_2$ `and` $x.age < 30$ `order by` $x.salary$ `asc`
3	`select` x `from MANAGER as` x `where` $x.name = :P_1$ `and` $x.salary > 100K$ `and` $x.dep = :P_2$ `order by` $x.rank$ `desc`

Table 1: Example Critical Queries.

form $C \sqsubseteq D$, with left and right-hand concept descriptions defined by the grammar in Figure 3. The grammar assumes a set $\{PC_1, PC_2, \ldots\}$ of *primitive concepts* and a set $\{A_1, A_2, \ldots\} \cup \{\texttt{ID}\}$ of features[1]. As usual, a sequence of features is called a *feature path* and is denoted as *Pf*, possibly subscripted. The semantics is with respect to a structure $(\Delta, \leq, (\cdot)^I)$ in which Δ is a domain of objects, $\leq$ is an ordering relation over Δ and $(\cdot)^I$ is an interpretation that begins by mapping primitive concepts to a subset of Δ and features to total functions over Δ (this function must be the identity relation in the case of `ID`). The inclusion dependency $C \sqsubseteq D$ holds iff $C^I \subseteq D^I$. The logical implication problems asks $\mathcal{T} \models C \sqsubseteq D$; that is, if all all interpretations that satisfy each inclusion dependency in $\mathcal{T}$ must also satisfy $C \sqsubseteq D$.

We use "$\bot$" as a shorthand for $C \sqcap \neg C$, $C_1 \sqcup C_2$ as a shorthand for $\neg(\neg C_1 \sqcap \neg C_2)$ and $C_1 \equiv C_2$ as a shorthand for the pair of inclusion dependencies $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. We also require that the defined functional dependencies (see the last line in Figure 3) are acyclic (e.g. $C \models C\{A^=\} \rightarrow B^=$ and $C \models \{B^=\} \rightarrow A^=$ cannot both be a logical consequence of a given terminology.)

Some inclusion dependencies that relate to our example schema are as follows.

$\texttt{EMPLOYEE} \sqsubseteq \texttt{PERSON}$
$\texttt{MANAGER} \sqsubseteq \texttt{PERSON}$
$\texttt{PERSON} \sqcap \forall Age.(\lambda x)(x < 32) \sqcap \forall Salary.(\lambda x)(x > 90K) \sqsubseteq \bot$
$(\lambda x)(x > 90K) \equiv (\lambda x)(90K < x \leq 100K) \sqcup (\lambda x)(x > 100K)$
$(\lambda x)(x < 32) \equiv (\lambda x)(30 \leq x < 32) \sqcup (\lambda x)(x < 30)$

Note that we have used λ expressions to denote primitive concepts over concrete domains. Implementation details follow in Section 3.1.

A workload consists of a set of critical queries and updates. Each query has the form:[2]

[1] Hereon, we refer to attributes as features in order to be consistent with DL terminology.
[2] A formal semantics for such queries is beyond the scope of this paper

	Syntax:† $\mathcal{DLFD}$		Semantics: Definition of "$(\cdot)^I$"
C	::=	PC	$(PC)^I \subseteq \Delta$
	|	$C_1 \sqcap C_2$	$(C_1)^I \cap (C_2)^I$
	|	$\forall A.C$	$\{x : (A)^I(x) \in (C)^I\}$
	|	$\neg C$	$\Delta \setminus (C)^I$
D	::=	C	
	|	$D_1 \sqcap D_2$	$(D_1)^I \cap (D_2)^I$
	|	$\forall A.D$	$\{x : (A)^I(x) \in (D)^I\}$
	|	$C\{\, \mathsf{Pf}_1^{\sim_1}, ..., \mathsf{Pf}_k^{\sim_k} \,\} \rightarrow \mathsf{Pf}^{\sim}$	$\{x : \forall\, y \in (C)^I.$ $\bigwedge_{i=1}^{k} (\mathsf{Pf}_i)^I(x) \sim_i (\mathsf{Pf}_i)^I(y) \Rightarrow (\mathsf{Pf})^I(x) \sim (\mathsf{Pf})^I(y)\}$

† $k > 0$, $\sim \in \{<, \leq, =, \geq, >\}$

Figure 3: Syntax and Semantics of $\mathcal{DLFD}$.

```
select x
from C as x
where x.A_1 = :P_1 and ... and x.A_a = :P_a and [x.A_{a+1} between :P_{a+1} and :P_{a+2}]
order by [x.A_{a+1} d_{b+1}], x.B_1 d_1, ..., x.B_b d_b.
```

Note that we use A and B, possibly subscripted, to denote features. In the above query, $\{P_i\}_{i=1}^{a}$ are parameters, P_{a+1} and P_{a+2} are either both parameters, or $P_{a+1} = -\infty$ and P_{a+2} is a parameter, or $P_{a+2} = +\infty$ and P_{a+1} is a parameter. Also, $d_i \in \{\texttt{asc}, \texttt{desc}\}$ for $i \in [1..b{+}1]$ and $[\cdot]$ is used to denote an optional component, where we require that if one of the optional components is present, then so is the other. For example, the second query from Table 1 is based over the concept description $\texttt{EMPLOYEE} \sqcap \forall Age.(\lambda x)(x < 30)$ and $P_{a+2} = +\infty$.

A critical update can add or remove an object from a primitive concept or modify the value of an object's feature. We will require that every critical update is consistent, which in turn implies that the database scheme $\mathcal{T}$ is acyclic with respect to inclusion dependencies. For simplicity, we assume that critical updates of all three types are defined for each primitive concept in $\mathcal{T}$.

The solution of the ISECP problem is a set $E = \{e_i\}_{i=1}^{m}$ of *extended indices* that have the following properties: (1) There exists an extended index $e_i \in E$ that can be used to *efficiently answer* every critical query Q. We define this to hold when Q can be answered using a single search in e_i (this will take $O(log(|e_i|))$ time for a balanced tree index) followed by $|Q|$ pointer chases, where $|e_i|$ and $|Q|$ denote, respectively, the size of the extended index e_i and the size of the query result for Q; (2) Every extended index $e_i \in E$ is *efficiently maintainable*, i.e. every critical update can be synchronized with e_i in $O(log(|e_i|))$ time; and (3) There does not exist a set of less than m indices for which (1) and (2) hold.

Definition 1 (Extended Index) *An extended index e is a rooted ordered tree with nodes N_e and node labels of the form "$C : A_1^{d_1}, \ldots, A_a^{d_a}$", where $d_i \in \{\texttt{asc}, \texttt{desc}\}$ for $i \in [1..a]$. We use $Desc(e)$, $Feat(e)$ and $Order(e)$ to denote "C", $\{A_i\}_{i=1}^{a}$ and "$A_1^{d_1}, \ldots, A_a^{d_a}$" respectively, and write r_e to denote the root node in e. The extended index e is well formed relative to a terminology $\mathcal{T}$ iff each of the following conditions are satisfied: (1) For any $\{n_1, n_2\} \subseteq N_e$, if n_1 is an ancestor of n_2 then $\mathcal{T} \models Desc(n_2) \sqsubseteq$*

$Desc(n_1)$ and $Feat(n_1) \cap Feat(n_2) = \emptyset$, and (2) For any $\{n_1, n_2\} \subseteq N_e$, if n_1 and n_2 have a common parent then $\mathcal{T} \models Desc(n_1) \sqcap Desc(n_2) \sqsubseteq \bot$.

Operationally, an extended index e has an entry for each object $o \in (Desc(r_e))^I$. The index is sorted first on $Order(r_e)$. Suppose $\langle n_1, \ldots, n_k \rangle$ are the children of r_e in order. Then, the index is next sorted on the feature A in ascending direction, where $o.A = i$ for some $1 \leq i \leq k$ if $o \in (Desc(n_i))^I$ and $o.A = 0$ otherwise. The index is next sorted depending on the value of the feature A. The objects for which $o.A = i$ are sorted relative to the labels of the node n_i and its descendants, where the order is defined in the same manner as for r_e. The objects for which $o.A = 0$ are sorted in any order. We will require that an extended index has an efficient "get next" capability, i.e. once an element is found, then the next element according to the defined index order can be found in amortized constant time.

Note that an extended index that is well-formed relative to a given terminology defines a partial order on the objects included in the index. This is a simple consequence of conditions (1) and (2), and our operational definition of ordering within the index.

We define T to be a label function that maps an (ordered) node-labeled rooted tree to a string. We will use $\mathcal{L}(n)$ to denote the label of the tree with root n. For a leaf node we define $\mathcal{L}(n)$ to be the label of the node n. For a non-leaf node we define $\mathcal{L}(n) = Desc(n) : Order(n)[(\mathcal{L}(n_1)), \ldots, (\mathcal{L}(n_k))]$, where $\langle n_1, \ldots, n_k \rangle$ are the (ordered) children of n. For a tree t we define $T(t) = \mathcal{L}(t_r)$, where t_r is the root node of the tree t. Throughout the paper we will sometimes use $T(t)$ to describe the tree t.

Given an extended index e, we will use $Q(e)$ to denote the set of queries that we expect to be efficiently answerable using e. First, suppose that e contains a single node with label $C : A_1^{d_1}, \ldots, A_a^{d_a}$. Then $Q(e)$ denotes the set of queries in which the first k features are fixed ($0 \leq k \leq a$), possibly a range is specified on the $k+1^{st}$ feature, and the order $A_{k+1}^{d_{k+1}}, \ldots, A_m^{d_m}$ is specified on the query result, where $k \leq m \leq a$. Next, suppose that e is a tree with nodes N_e. For a node $n \in N_e$, let $\langle C_1 : L_1, \ldots, C_k : L_k \rangle$ be the labels along the path from the root of e to the node n. Then, we define $Q(n) = Q(e')$, where e' is the index that has a single node labeled $C_k : L_1, \ldots, L_k$. We define $Q(e) = \bigcup_{n \in N_e} Q(n)$.

3 The Proposed Solution

Our solution to the problem consists of four steps. In the first step, we transform each input query into a *parameterized access requirement type* (PART), where a PART describes a set of extended indices. (Figure 2 depicts the PART $C_1 : \{name\}[(C_2 : salary^{\texttt{asc}})(C_3 : \{dep\}, rank^{\texttt{desc}})]$.) In the second step, we use the database schema to simplify a set of PARTs. In Step 3 we merge PARTs in order to save space and in the final step we build an extended index for each remaining PART.

Definition 2 (PART/PART permutation) *A PART $\mathcal{P}$ is an unordered rooted tree with node labels of the form "$C : \{H_1\}, V_1, \ldots, \{H_n\}, V_n$", where H_i is of the form "$A_1, \ldots, A_a$", V_i is of the form "$B_1^{d_1}, \ldots, B_b^{d_b}$" and $d_i \in \{\texttt{asc}, \texttt{desc}\}$. A complete permutation π for a PART $\mathcal{P}$ converts it into an extended index. It does so by converting each H component of a node's label into a V component and by fixing*

the tree order. The first task is accomplished by first permutating the features inside the component, then adding the superscript `asc` *or* `desc` *to each feature and finally removing the curly braces. A PART $\mathcal{P}$ is well formed relative a terminology $\mathcal{T}$ iff for every complete permutation π for $\mathcal{P}$, $\pi(\mathcal{P})$ is a well formed extended index relative to $\mathcal{T}$. A PART $\mathcal{P}$ describes the set of extended indices that can be produced under the different complete permutations for $\mathcal{P}$. We will use $E(\mathcal{P})$ to describe this set. We also define $Q(\mathcal{P}) = \bigcap_{e \in E(\mathcal{P})} Q(e)$. This is the set of queries that can be efficiently answered by any of the extended indices of $\mathcal{P}$. A regular permutation π for a PART $\mathcal{P}$ converts into the PART $\mathcal{P}'$ for which $E(\mathcal{P}') \subseteq E(\mathcal{P})$. We will say that the permutation π for the PART P is more restrictive then the permutation π' for the same PART iff $E(\pi(\mathcal{P})) \subset E(\pi'(\mathcal{P}))$. We will call a PART $\mathcal{P}$ a simple PART if it consists of a single node and a nested PART otherwise.*

3.1 Step 1: Converting Critical Queries into PARTs

Suppose we are given a query Q as part of the critical workload (see Section 2 for the syntax of Q). In this step of the algorithm we will convert Q into the PART $C : \{A_1, \ldots, A_a\}, B_1^{d_1}, \ldots, B_b^{d_b}$ if Q does not contain the optional components and into the PART $C : \{A_1, \ldots, A_a\}, A_{a+1}^{d_{b+1}}, B_1^{d_1}, \ldots, B_b^{d_b}$ otherwise. For example, the queries from Table 1 will be converted into the PARTs $C_1 : \{name\}$, $C_2 : \{name\}, salary^{\texttt{asc}}$ and $C_3 : \{name, dep\}, rank^{\texttt{desc}}$ respectively, where $\{C_i\}_{i=1}^3$ are defined in Figure 2.

Note that for each concept term over a concrete domain that is introduced (e.g. $(\lambda x)(x > 100K)$), we need to add constraints to the terminology that describe the relationship between the introduced concept term and the other concept terms over the same concrete domain. For example (see [5]), suppose we are given the domain of real numbers and a set of concept terms corresponding to predicates that are built using the comparisons $\{>, \geq, <, \leq, =\}$ and the constants $\{a_i\}_{i=1}^n$. We can then partition the domain into $2n + 1$ groups. There will be one group for each of the n constants and $n + 1$ groups for the $n + 1$ intervals defined by them. One then creates $2n + 1$ primitive concepts to represent the groups. As a result, each original concept term can be described using the additional $2n + 1$ primitive concepts (see Section 2 for an example). For a concept term over the domain of integers, a similar procedure can be applied (see [6]).

Second, note that the created PART for a query Q exactly describes the set of extended indices that can be used to answer Q efficiently. If Q does not contain the optional components, then any index on C that is ordered by the features $\{A_i\}_{i=1}^a$ in arbitrary order and direction followed by the ordering $B_1^{d_1} \ldots B_b^{d_b}$ can efficiently support Q. In order to do so, we just need to search for the values $\{P_i\}_{i=1}^a$ of $\{A_i\}_{i=1}^a$ in the extended index and then return the result in the order defined by the index. Similarly, if Q contains optional components, then we need to search for the values $\{P_i\}_{i=1}^a$ for $\{A_i\}_{i=1}^a$, followed by a search of the value P' for A_{a+1} and then return all objects from the index in order until the value P'' for A_{a+1} is passed. We have used P' and P'' to denote P_{a+1} and P_{a+2}, respectively, when $d_{b+1} =$ `asc` and P_{a+2} and P_{a+1}, respectively, when $d_{b+1} =$ `desc`.

3.2 Step 2: Simplifying PARTs

We can use schema information for simplifying PARTs along the lines suggested in [3]. For example, consider the third query from Table 1 and the inclusion dependency $\texttt{MANAGER} \sqsubseteq \texttt{MANAGER}\{name^{=}\} \rightarrow rank^{=}$. The PART $\texttt{MANAGER} : \{name, dep\}, rank^{desc}$ for this query can be rewritten as $\texttt{MANAGER} : \{name, dep\}$. The reason is that two managers which have the same name will have the same rank, and consequently there is no need to sort the managers with the same name according to their rank.

In general, consider a simple PART $\mathcal{P} = C : \{A_1, \ldots, A_a\}, B_1^{d_1}, \ldots, B_b^{d_b}$ produced by Step 1 of the algorithm. Let $\langle A'_1, \ldots, A'_a \rangle$ be a permutation of $\{A_1, \ldots, A_a\}$. Then, $\mathcal{T} \models C \sqsubseteq C\{A'^{=}_1, \ldots, A'^{=}_i\} \rightarrow A'^{=}_j$, for $1 \leq i < j \leq a$, implies that the feature corresponding to A'_j in $\{A_i\}_{i=1}^{a}$ can be removed from $\mathcal{P}$. Similarly, $\mathcal{T} \models C \sqsubseteq C\{A'^{=}_1, \ldots, A'^{=}_i\} \rightarrow B^{=}_j$, for $1 \leq i \leq a$ and $1 \leq j \leq b$, implies that B_j can be removed from $\mathcal{P}$. And finally, $\mathcal{T} \models C \sqsubseteq C\{A'^{=}_1, \ldots, A'^{=}_i, B^{=}_1, \ldots, B^{=}_j\} \rightarrow B^{=}_k$, for $0 \leq i \leq a$, $1 \leq j < k \leq b$, implies that B_k can be removed from $\mathcal{P}$. Again, the reason that this is possible is that a functional dependency guarantees that, once the features at its left-hand side are fixed, there can be at most one different value for the right-hand-side feature. Our assumption of acyclicity of the functional dependencies in the schema ensures that this process is deterministic.

3.3 Step 3: Merging PARTs

The idea behind PART merging is to combine two or more PARTs into a single PART that has the same query answering capabilities. PART merging is beneficial because it saves space and, also, allows updates to be performed faster.

Definition 3 (PART merging) *We will use "$\oplus$" to denote the PART merging operator.* $\mathcal{P} = \mathcal{P}_1 \oplus \ldots \oplus \mathcal{P}_m$ *holds when* $Q(\mathcal{P}) = \bigcup_{i=1}^{m} Q(\mathcal{P}_i)$.

There are two cases we will consider when merging simple PARTs $\mathcal{P}_1 = C_1 : L_1$ and $\mathcal{P}_2 = C_2 : L_2$. The first case occurs when $\mathcal{T} \models C_2 \equiv C_1$. Failing this, the second case occurs when $\mathcal{T} \models C_2 \sqsubseteq C_1$. (On notation, since a permutation does not affect the concept description of a simple PART $\mathcal{P} = C : L$, we will sometimes write $\pi(L)$ to refer to the expression $\pi(\mathcal{P})$ without the concept description part.) In the first case, $\mathcal{P}_1$ and $\mathcal{P}_2$ can be merged if there exist permutations π_1 and π_2 such that $\pi_1(L_1)$ is a prefix of $\pi_2(L_2)$. The result of $\mathcal{P}_1 \oplus \mathcal{P}_2$ will be $C_2 : \pi_2(L_2)$, where π_2 is the least restrictive permutation such that there exists a permutation π_1 for which the merging condition holds. In the second case, $\mathcal{P}_1$ and $\mathcal{P}_2$ can be merged when there exist permutations π_1 and π_2 such that $\pi_1(L_1)$ is a prefix of $\pi_2(L'_2)$, where we have used L'_2 to denote the longest prefix of L_2 that consists entirely of curly braces components. Let π_1 and π_2 be the least restrictive permutations for which the merging condition holds. We define $L_3 = \pi_1(L_1)$, and L''_2 such that $L_2 = L'_2, L''_2$. We also define L_4 such that $\pi_2(L'_2) = L_3, L_4$. Then the result of $\mathcal{P}_1 \oplus \mathcal{P}_2$ will be $C_1 : L_3[(C_2 : L_4, L'_2)]$.

We next examine how a nested PART $\mathcal{P}_1$ can be merged with a simple PART $\mathcal{P}_2 = C : L$. The merging condition is that $\mathcal{P}_1$ should contain a path K that starts from the root of the tree and has labels $\langle C_1 : L_1, \ldots, C_k : L_k \rangle$ such that (1) the simple PART $\mathcal{P}'_1 = C_k : L_1, \ldots, L_k$ and the PART $\mathcal{P}_2$ are mergable and (2) if $\mathcal{T} \not\models C_k \equiv C$,

then $\mathcal{T} \models Desc(n) \sqcap C \sqsubseteq \bot$ should hold for each child n of the end node of the path K. It is easy to see that if $\mathcal{P}_1$ is a well formed PART, then there can be at most one such path in $\mathcal{P}_1$. Suppose the merging condition holds. Let $\mathcal{P}' = \mathcal{P}'_1 \oplus \mathcal{P}_2$. If $\mathcal{P}'$ is a simple PART, then $\mathcal{P}_1 \oplus \mathcal{P}_2$ is constructed from the PART $\mathcal{P}_1$ by relabeling the nodes along the path K using the label from the root node of $\mathcal{P}'$. Alternatively, if $\mathcal{P}'$ is a complex PART, then $\mathcal{P}_1 \oplus \mathcal{P}_2$ is constructed as in the previous case, with the addition that the child node in $\mathcal{P}'$ is appended as a child to the end node of the path K.

Our algorithm for merging m PARTs follows. It runs in exponential time and uses dynamic programming. In the first step, we consider all possible pairs of initial PARTs and put each pair in a separate configuration. If the two PARTs that are in a configuration can be merged, then we do so. We will call a configuration that is built from k initial PARTs a k-configurations. In the second step we create 3-configurations by examining all possible pairs of a 2-configurations and an initial PART that wasn't used in the creation of the 2-configuration. Again, when the initial PART can be merged with any of the PARTs in the configuration, we do so. After the $(m-1)^{st}$ step is applied, a set of m-configurations will be produced. The algorithm next finds a m-configuration with the smallest number of elements and returns the PARTs in it.

Theorem 1 *Our PART merging algorithm finds the fewest number of PARTs that can be created from the initial PARTs by applying any PART merging strategy.*

Proof (Sketch) In order to prove the theorem we will prove the following claims: (1) The presented PART merging rules are correct relative to Definition 3; (2) If the preconditions for merging a simple PART and an arbitrary PART are not satisfied, then the PARTs cannot be merged; (3) If two complex PARTs can be merged, then the result of merging the two PARTs will be identical to merging the first PART with the simple PARTs used in the construction of the second PART and (4) The described PART merging algorithm finds the smallest set of PARTs that can be produced from the input set by applying the described PART merging rules. Claims (1) and (2) can be easily verified. Claim (3) is a direct consequence of Definition 3. We next prove claim (4). Let $\bar{\mathcal{P}} = \{\mathcal{P}_i\}_{i=1}^{m}$ be the input PARTs of the algorithm and $\bar{\mathcal{P}}' = \{\mathcal{P}'_i\}_{i=1}^{k}$ be the set of PARTs in a m-configuration with the smallest number of PARTs. First, note that the algorithm constructs the set $\bar{\mathcal{P}}'$ by applying the PART merging rules to the set $\bar{\mathcal{P}}$. Next, note that for every set of PARTs $\bar{\mathcal{P}}_1 \in \bar{\mathcal{P}}$ the algorithm considers any s-configuration that can be constructed by applying the merge rules to the elements of $\bar{\mathcal{P}}_1$, where s is cardinality of $\bar{\mathcal{P}}'$. This implies that the algorithm is exhaustive, i.e. it indeed finds a minimal set of PARTs that can be constructed from the input set by applying the PART merging rules. ■

We will next show how the PART merging algorithm works on our running example. Suppose that we are given as input the PARTs corresponding to the queries in Table 1 (See Section 3.1). The algorithm will build the following 2-configurations: $\{C_1 : \{name\}[(C_2 : salary^{\texttt{asc}})]\}$, $\{C_1 : \{name\}[(C_3 : \{dep\}, rank^{\texttt{asc}})]\}$ and $\{C_2 : \{name\}, salary^{\texttt{asc}}, C_3 : \{name, dep\}, rank^{\texttt{desc}}\}$. From those 2-configurations the algorithm will build several 3 configurations, where one of them will consist of the single PART depicted in Figure 2.

It is natural to ask if there exists a polynomial time algorithm for solving the PART merging problem. The answer is no if $P \neq NP$.

Theorem 2 *Suppose we are given a set of PARTs over a database schema $\mathcal{T}$ and an integer K. Suppose as well that we can check whether two PARTs are mergable and merge them if they are in polynomial time. Then the decision problem of whether the PARTs can be merged into $\leq K$ PARTs is NP-Complete.*

Proof (Sketch) First, note that the problem is in NP. The reason is that a certificate that partially fixes the order of the features inside curly braces of the given PARTs and groups the PARTs that are to be merged can be verified in polynomial time. Next, we will prove that the problem is NP-Hard by a reduction from minimum-maximal matching for a bipartite undirected graph, where the later problem is known to be NP-Complete ([4]). Suppose we are given the undirected bipartite graph $G = (V, E)$. Then, since the graph is bipartite, its vertices can be split into the sets $V' = \{V'_i\}_{i=1}^{a}$ and $V'' = \{V''_j\}_{j=1}^{b}$ in such a way so that there doesn't exist an edge between two vertices of the same set. Next, we will label the vertices in V' as $V'_i = C : A_i$ and the vertices in V'' as $V''_j = C : B_j, \bar{A}_j$, where $A \in \bar{A}_j$ iff there exists a vertex $V_k \in V'$ that has label $C : A$ and for which $(V_k, V_j) \in E$ holds. Next, we ask ourselves if the $a + b$ simple PARTs that are formed from the node labels of G can be merged into $\leq K$ PARTs. If the answer is yes, then there exists a maximum matching of G with $\leq K$ edges. This matching corresponds to the pairs of PARTs that are merged in the found solution. This is a matching because we have constructed the labels in such a way so that each PART can be merged with exactly one of its adjacent PARTs in the graph G. ■

Corollary 1 *Even the simplified version of the ISECP problem in which functional dependencies are not part of the schema model and DL reasoning is in P is NP-Hard.*

Proof(Sketch) There is a straight-forward polynomial time reduction from the PART merging problem to the described simplified ISECP problem. ■

3.4 Step 4: Constructing Extended Indices

The final step is to produce an extended index for each remaining PART. Given a PART $\mathcal{P}$, we first add a child node n' with label "$C' : \texttt{ID}^{\texttt{asc}}$" to each non-leaf node n for which $\mathcal{T} \not\models \bigsqcup_{i=1}^{k} Desc(n_i) \equiv Desc(n)$, where $\{n_i\}_{i=1}^{k}$ are the children of n. We also add the constraint $C' \equiv C \sqcap \neg C_1 \sqcap \ldots \sqcap \neg C_k$ to $\mathcal{T}$, where $C \equiv Desc(n)$ and $C_i \equiv Desc(n_i)$. This procedure introduces in $\mathcal{P}$ an order for the objects that are not explicitly defined in it. We next convert the PART into an extended index by choosing a complete permutation for each PART and creating the corresponding extended index. We also make each extended index e a total order. Let $A_1 \ldots A_a$ be the features referenced in the path from the root of e to the node n. Then, we will append a sort on $\texttt{ID}$ ascending to the label of node n if $\mathcal{T} \not\models Desc(n) \sqsubseteq Desc(n)\{A_1^{=}, \ldots, A_a^{=}\} \rightarrow \texttt{ID}^{=}$. The fact that the resulting extended indices can efficiently support the input set of critical queries follows from the correctness of the previous steps of the algorithm. As well, each update can now be efficiently propagated to each extended index. The reason is that each extended index consists of the elements in some concept description and that there is a total order on the elements. The presented reasoning informally shows that our algorithm for solving the ISECP problem is sound.

Going back to our running example, suppose that no functional dependencies can be deduced from $\mathcal{T}$. Then the extended index $C_1 : name^{\texttt{desc}}[(C_2 : salary^{\texttt{asc}}, \texttt{ID}^{\texttt{asc}}), (C_3 : dep^{\texttt{asc}}, rank^{\texttt{desc}}, \texttt{ID}^{\texttt{asc}}), (C_4 : \texttt{ID}^{\texttt{asc}})]$ is a possible output of our algorithm, where $C_4 \equiv C_1 \sqcup \neg C_2 \sqcup \neg C_3$.

4 Extensions and Conclusion

In this paper we have explored only a limited type of critical queries that are based on a single concept description. However, those queries are fundamental because more complex queries can be efficiently answered on top of them. For example, a join between the concept description C_1 and the result of the critical query Q_2 that is based over the concept description C_2 can be performed by an index scan of the extended index for Q_2 followed by a set of scans on the extended index for C_1 if there is an inclusion dependency between C_1 and C_2. Note as well that such a query can be answered efficiently, i.e. in time $O(|Q| + log|I|)$, where $|Q|$ and $|I|$ represent, respectively, the size of the query result of Q and the size of the indices that are used to answer Q. Finally, note the fact that the proposed algorithm runs in exponential time doesn't make it impractical because the algorithm is part of a preprocessing stage, i.e. it only needs to be run once.

5 Acknowledgments

The authors greatfully acknowledge the support of the National Science and Engineering Research Council of Canada, of the Communications and Information Technology Ontario and of Nortel Networks Ltd.

References

[1] 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.

[2] 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.

[3] Eugene Shekita David Simmen and Timothy Malkemus. Fundamental Technique for Order Optimization. *ACM SIGMOD*, pages 57–67, 1996.

[4] E.L.Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[5] 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.

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

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

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

[9] 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.

[10] G. 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.