Query Processing in Embedded Control Programs

David Toman and Grant Weddell

Department of Computer Science
University of Waterloo, Canada
{david,gweddell}@uwaterloo.ca

Abstract. An embedded control program can be viewed as a small main-memory database system tailored to suit the needs of a particular application. For performance reasons, the program will usually define concrete low level data structures to encode the database, which in turn must be understood by anyone who needs to develop or modify the program. This is in contrast with the data independence that can be achieved by using a database system. However, because of space and performance requirements, the use of current database technology is not likely to be feasible in this setting. To explore one obstacle to this, we have developed a query optimizer that compiles queries on a conceptual schema to native Java or C code that navigates low level data structures. Of crucial significance is that an arbitrary collection of such structures, *perhaps already devised for an earlier version of the control program*, can be given as a part of the input to the optimizer. We present an overview of the underlying algorithms that are used to accomplish this. The algorithms are based on a novel resource bounded plan generation mechanism in which integrity constraints abstracting the definition of stored views are applied to source queries to extend the search space of possible query plans. We also report on some preliminary experimental results that suggest generated code navigates concrete data structures with an efficiency comparable to code written directly by expert programmers.

1 Introduction

Our contributions in this paper are twofold. First, we outline a new application area for database technology: the class of *embedded control programs* (ECPs). An ECP is a software subsystem consisting of one or more modules that implement a main memory database system tailored to suit the needs of a particular embedded control application. The main memory data is called *control data*.

An existing code base for an ECP will usually define low level data structures, a collection of C records containing arrays and pointers for example, to encode the control data. These structures must be understood by anyone who needs to develop or modify the ECP, which is in contrast with the data independence that can be achieved by using a database system. However, due to space and performance requirements, the use of existing database technology is not likely to be feasible in this setting. For example, there might be no (or very little) external storage available to the ECP. The main contribution of this paper is a new technique that enables one to compile high level declarative queries formulated on a conceptual view of an ECP's control data to a program in second or third generation programming languages that uses low level data structures to effect query evaluation. Our technique focuses on the issue of finding *navigational* query plans, those that utilize pointer navigation in places where standard relational query processing engines would need to scan a table or an index (a much more costly operation when accessing main memory data). An added complication relates to ECPs that are *legacy* systems with an existing and often very large base of code. In such circumstances, a query optimizer must be capable of *fine grained information integration*; that is, it must be possible to supply an arbitrary concrete encoding of main memory data already devised for an existing version of an ECP as part of the input to the optimizer.

To demonstrate that our technique is feasible and is capable of fined grained information integration, we have implemented a query optimizer for the **DEMO** system[1] that compiles SQL-like queries on a conceptual schema to native Java or C code that navigates an existing collection of low level data structures. This is in contrast with current ongoing efforts to develop "ultra-light" or main memory database engines in which data must be stored in product-specific internal forms. Our second contribution is a presentation of the underlying algorithms used by this optimizer. The algorithms are based on a novel resource bounded plan generation mechanism in which integrity constraints are applied to source queries to a *given limit* in order to extend the search space for possible query plans. An important role played by the constraints is to relate the "stored schema" that abstracts concrete data structures to a much more transparent conceptual schema on which user-defined queries are specified. Thus, the constraints are an abstraction of stored views with the additional property of being bi-directional; that is, they can be used to reason about conceptual data given view data and vise versa—a property that is crucial to our algorithms. As might be expected, the aforementioned limit on applying integrity constraints must be set very high to ensure completeness results (in cases where such limits exist). This remains true for simple families of queries. However, our experience indicates that low thresholds for this limit work well in practice.

The remainder of the paper is organized as follows. Section 2 gives a more thorough introduction to the topic of ECPs, and presents a pair of case studies that further clarify issues related to the processing of their queries. Our first case study also reports on some preliminary experimental results that suggest the code generated by our optimizer is capable of navigating legacy data structures with an efficiency that compares to legacy code written directly by expert programmers. In Sections 3 and 4, we outline the foundations of our approach to query optimization. In particular, we present the basic concept of resource

[1] **DEMO** is an acronym for *Design Environment for Main Memory Object Oriented databases.* The **DEMO** project at the University of Waterloo is an ongoing collaborative effort with Nortel Networks Ltd. [23] to adapt database technology for use in "industrial strength" ECPs.

bounded plan generation described above. Section 5 elaborates on this general approach with more details relating to the processor architecture, to plan selection and to code generation. Section 6 summarizes the contributions and gives a brief overview of related work.

2 Embedded Control Programs: Case Studies

To be more concise: an ECP is a software subsystem consisting of one or more modules that are responsible for managing a *uniform workload* for a *predefined set of transaction types* on a main memory database of control data. The assumption that there is a uniform workload is a key part of the definition that distinguishes an ECP from an arbitrary software subsystem. In particular, it becomes reasonable to ignore issues that relate to bulk reformatting of the control data. The following are examples of software systems that illustrate this. Each has a component ECP that consists of a single "dot h" file, the low level *schema* of the underlying control data, with a number of "dot c" clients that "implement" the predefined set of transaction types.

- An operating system kernel. The control data corresponds to information relating to open files, executing processes, and so on. The components of the ECP include the kernel functions and procedures together with utilities for managing the control data (a "`ps`" command, for example). In this case, the need to provide services for management and analysis purposes make it especially desirable to provide a conceptual interface to the kernel data. Other capabilities of a database system relating to concurrency control would also be useful.
- The control program for a communications system such as a telephone switch. In this case, the need for very high transaction throughput on so-called "office data" is clearly acute and therefore precludes using a database system that introduces *any* interpretive overhead on query evaluation. Virtually all the capabilities of a database system would be beneficial: physical data independence, non-procedural query languages, concurrency control and backup and recovery.

Both examples suggest that tools for processing queries in an ECP should perform as many as possible of the run-time tasks of existing database engines at the time the ECP is compiled or generated, particularly in the case of query optimization and plan generation. In the following subsections, we present a corresponding pair of case studies in order to clarify the issues related in particular to query processing in an ECP.

The first considers kernel data in the Linux operating system (the Linux kernel is the ECP for this case). We show how information on processes and open files may be viewed as a relational schema, and how queries on this schema are compiled by our optimizer to code that navigates the internal C records. We also report on some experimental results for this case that relate to a hypothetical re-implementation of the Linux `ps(1)` command as an SQL query on the relational

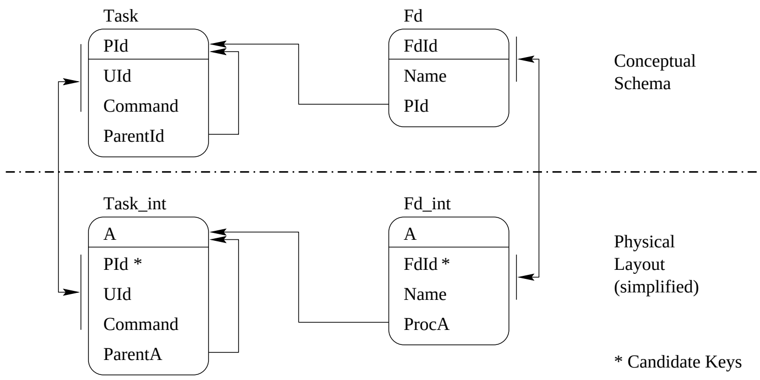

Fig. 1. Linux processes and file descriptors.

schema. The results indicate that the code generated by our query optimizer for this particular query navigates the Linux kernel data structures with an efficiency superior to the existing implementation. The second case study considers a more general class of ECPs: embedded control programs for *private branch exchanges* (PBXs). The case is a more dramatic means of clarifying issues related to query processing for ECPs, particularly the need to support the specification of legacy data structures as a parameter to query processing.

2.1 Case 1: The Linux Kernel

Our first case considers kernel data in the Linus operating system [3, 4]. The kernel data structures contain hundreds of Kb of run-time data stored in highly compact data structures linked by pointers. However, a conceptual view of this data based on a collection of relation schemes is relatively straightforward to devise, which then enables the use of standard database application development techniques to be applied to the production, among other things, of Linux "system management" applications such as `ps(1)`, `top(1)`, `vmstat(1)`, and so on. In consequence:

- A clean and simple conceptual schema can be separated from implementation details, which in turn can be hidden from application developers to facilitate *data independence*.
- Application data retrieval requests can be expressed as high level queries over the conceptual view.
- The query optimizer, with the help of a small wrapper that attaches the generated code to the Linux kernel, provides a means for efficient execution of high level declarative queries over the actual kernel data structures.
- There is no interpretative overhead for query plan execution; the operations in our plans correspond directly to C statements.

For the remainder of this paper, we use examples derived from this case study; and for the sake of brevity, we focus on two data structures maintained by the kernel: *processes* running in the system, and *files* accessed by these processes. This simplified setting is depicted in Figure 1 in which the top part of the figure corresponds to the relevant part of a *conceptual schema* consisting of two relations:

```
Task(PId  integer, /* proc. Id */   Fd(FdId integer,     /* File Id */
     PPId integer, /* parent Id */     Name char(255),  /* file name */
     UId  integer, /* owner uid */     PId  integer)    /* owner PId */
     Cmd  char(16)) /* command */
```

Also, Linux ensures that every file descriptor has exactly one "owner" process, and that every process has a parent (the first process is its own parent). It is now possible to specify queries on this schema.

Example 1 A very simple example is a query that, given a user id, `uid`, prints out all tasks in the system owned by this particular user; a `ps(1)` replacement[2]:

```
select PId, ParentId, Command
from   Task
where  UId = :uid
```

Although the query resembles a simple selection from a base table, the generated code must perform a self-join of the `Task_int` tables to obtain the process id of the parent task. However, the query optimizer will recognize the opportunity to implement the self-join by navigating the `ParentA` field. Running the code generated by our query optimizer for this query against the actual data structures yields the following result (over a slightly modified Linux Kernel 2.2.5):

```
Enter uid: 501
  PId ParentId Command
  731      730 db2sysc
  732      731 db2sysc
  737      732 db2sysc
  736      732 db2sysc
 2336     2335 bash
```

It turns out that the running time is much shorter than the running time of `ps(1)`, although we suspect that this mainly due to avoiding the overhead associated with numerous system calls.

The relational abstraction of process and file descriptors, however, allows one to pose queries that would require a large amount of effort to express in C, but that remain easy to formulate in SQL:

1. For all processes, list the command that invoked their parent;
2. Find the "first" process in the system;
3. List all processes with no children; with one child; . . . ,
4. List all files open by a particular process;
5. List all processes that share a particular file (based on file name); and
6. List pairs of processes that share an arbitrary file.

[2] For a particular choice of command line options.

In each case, our query optimizer will automatically generate C code that can be directly embedded in application programs without any additional runtime support. In addition, the query optimizer naturally favors navigational plans; the generated code often rivals code written by proficient C programmers, particularly in cases where complex navigational patterns need to be used.

The conceptual schema is connected to a corresponding physical schema using integrity constraints. The bottom part of Figure 1 illustrates this: the `Task_int` and `Fd_int` relations abstract the collections of data records used by the kernel, including explicit address fields (`A`, `ParentA`, and `ProcA`). The actual connection between the conceptual schema and the physical layout is more complex than suggested by Figure 1; we have simplified the way file names are obtained from a file descriptor (which in reality involves navigating several other data structures). Note that the *address* fields allow direct navigation between instances of the schemas in the lower part of the figure (cf. section 3.2).

Although our example in Figure 1 is limited, the approach appears general enough to capture all of the data structures used in the Linux kernel. We can provide relational abstractions for other kernel subsystems, e.g., for the memory management system, the IPC, the sockets and networking system, etc.

2.2 Case 2: A PBX

Now consider a more *data intensive* situation: a *private branch exchange* (PBX) for switching voice data at a private site. The software architecture for a typical PBX is illustrated in Figure 2. There are two subsystems that access a common

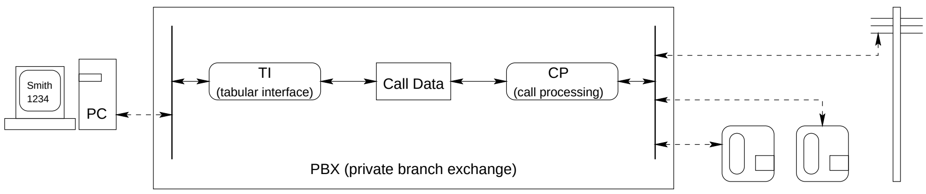

Fig. 2. Common Architecture for a PBX.

database of "call-data" storing diverse information about dial numbers, routing information, physical resources, test status information and subscribers. The first subsystem, *call processing* (CP), handles all clerical tasks relating to call setup, routing, hardware/software self diagnosis, and so on. CP handles transactions arriving from phone lines and handsets. The second subsystem, *tabular interface* (TI), provides external nested tabular views of various call-data and office parameters for network management purposes; subscriber data may be queried and new subscribers added by invoking transactions supported by this interface, for example.

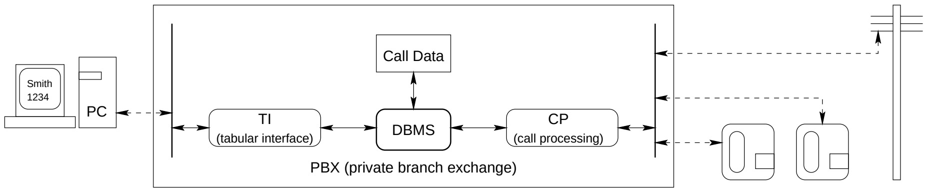

Fig. 3. A Preferred Architecture.

There are several problems with this architecture that arise from the typical situation in which the CP and TI subsystems are developed by programmers in a general purpose low level programming language such as C.

- Call data must be viewed by both systems in terms of its encoding as a set of C data (`struct`) types.
- The TI subsystem must be hand coded to provide the needed external views.
- Both subsystems must directly address any issues relating to concurrent access to call-data, and with its backup and recovery.

Now consider an alternative architecture illustrated in Figure 3. In this setting, a DBMS is used to manage the call-data. Although the introduction of a DBMS solves the above problems, the architecture of existing database technology results in an unacceptable reduction in transaction throughput for the PBX.

We would like to employ database technology in precisely the manner suggested by this alternative architecture, but with transaction throughput and store costs no less than what can be achieved with the basic architecture (in which expert programmers code in C). As we have noted earlier, this requires a great deal of what a database engine normally performs at runtime to be performed instead at compile-time. For example, access plans for statically specified queries must be "unfolded" directly into the TI and CP subsystems, and a simplified DBMS engine that excludes a query optimizer should be generated if there is no need to support dynamic ad-hoc queries. For the extreme case (static parameterized queries only, serialized transaction schedules suffice and no need for backup and recovery), there should be no DBMS footprint at all.

A further complicating factor, due to the infeasibility of complete rewrites of such systems, is that it must be possible to introduce this alternative architecture in an incremental fashion. Figure 4 illustrates an example of what we mean by this; only the TI subsystem for a hypothetical PBX software base has been reimplemented at the higher more abstract level supported by a DBMS. Thus, it becomes imperative to be able to relate a conceptual description of control data to an encoding corresponding to an existing collection of C types. The need to support very high transaction throughput also implies that a query optimizer is able to generate plans that navigate the C types with an expertise matching that of expert C programmers (existing systems are the performance benchmark).

There are other aspects of query processing in an ECP that derive from the embedded compile time nature of the application. The generated access plans

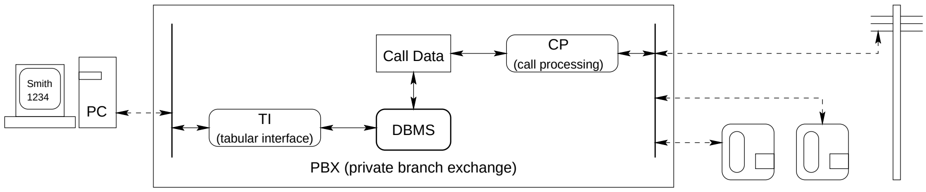

Fig. 4. An Incremental Architecture

should avoid the need to store arbitrarily sized intermediate results; that is, the temporary store required by a query plan should have a predictable constant size independent of the volume of control data. Conversely, although very high performance is needed for generated code, the amount of time spent on finding a query plan may be much longer than what would normally be acceptable in more traditional applications (e.g. in "interpreted" architectures in which queries are optimized at run-time).

3 Resource Bounded Plan Generation: Foundations

In this section, we focus on the basic ideas that underly our approach to finding valid (and preferably optimal) query plans for a particular storage layout. There are many alternative ways of describing how control data is encoded in an embedded controller's memory. Surprisingly, many of these definitions can be abstracted, without any loss of ability to find feasible query execution plans, by a combination of two basic ideas: the familiar notion of tuple and equality generating *generalized integrity constraints*, and notational conventions for capturing the *accessibility of relations*. We develop the foundations of our approach in this setting, beginning with the problem of finding query plans for conjunctive queries. We develop a query plan generator that performs a *resource-bounded expansion* of a query according to a supplied schema. We then outline extensions that are needed for discovering plans for general first-order queries. The following is used to demonstrate our approach for the remainder of the paper.

Example 2 (Running Example) Consider implementing a Linux application that lists files open by various processes in the system. The underlying conceptual schema is as follows:

```
Task(PId, UId, Command, ParentId)
Fd(FdId, Name, PId)
```

The schema captures the fact that, conceptually, the Linux kernel manages a set of *tasks* each associated with a (unique) process id (`PId`), a user id (`UId`), a parent task id (`ParentId`), and a command name (`Command`). Similarly, *descriptors of open files* are associated with an id, file name and the owning process id (cf. Figure 1). The following two relations

```
Task_int(A, PId, UId, Command, ParentA)
Fd_int(A, FdId, Name, ProcA)
```

capture the *physical layout* of the main-memory data: the values of the `A` fields represent *addresses* of records in main memory (cf. Section 3.2). Note that only the first two relation schemes are *visible* to the application developer; the later two are only visible to the query compiler.

We concentrate on the conjunctive fragment of first-order queries first. This fragment covers a wide class of *practical* queries, in particular those queries expressible using SQL's *select block*. It is well known [1] that conjunctive queries can be represented by a set of atomic formulas (representing their conjunction) and a set of *answer* variables.

Example 3 (Running Example: The Query) Our query in SQL, expressing the request for "what processes use what files?", is

```
select t.Command, f.Name
from   Task t, Fd f
where  t.PId = f.PId
```

This query can be represented by the pair $\langle (c, n), \{\mathtt{Task}(p, u, c, p'), \mathtt{Fd}(f, n, p)\}\rangle$ that in turn corresponds to a comprehension $\{((c, n) : \exists p, p', u, f.\mathtt{Task}(p, u, c, p') \wedge \mathtt{Fd}(f, n, p)\}$. There are two issues to clarify with this query.

- The relations `Task` and `Fd` are not explicitly stored in memory; their contents must be reconstructed from their *internal* representation in the kernel memory (which, in general, may be much more complex than in this example).
- The selection condition `t.PId = f.PId` has to be replaced by navigation of the `ProcA` field as the `PId` field is not stored in the `Fd_int` structures.

3.1 Conceptual Schema Declarations

To achieve our goals, the query compiler is provided with additional *schema information* that describes the relationship between the conceptual schema and the physically stored relations (cf. Definition 4 and Example 5), and the information about physically stored relations and the associated access methods (cf. Section 3.1). Even in cases where the underlying schema provides a direct access to every relation, this additional information will enable the query compiler to find alternative access plans (e.g., using navigation through other relations) which are often more efficient.

Integrity Constraints. The relationships between the conceptual schema and the actual encoding of the embedded application's data is described using generalized integrity constraints [1].

Definition 4 (Integrity Constraints) *Let $P_1, \ldots, P_k$, and P be atomic formulas (relation references) with variables as arguments. We use the following integrity constraints:*

$$\begin{array}{ll} P_1 \wedge \ldots \wedge P_k \Rightarrow R & \text{(a tuple generating dependency)} \\ P_1 \wedge \ldots \wedge P_k \Rightarrow x = y & \text{(an equality generating dependency)} \end{array}$$

The variables in constraints are implicitly universally quantified, except for the variables of R that do not occur in any of the P_i's, which are quantified existentially. We also require that variables x and y appear in one of the P_i's in the case of equality-generating constraints.

This definition covers a very large class of integrity constraints, including the usual *functional dependencies* and *embedded inclusion dependencies*. While we are aware of poor computational properties of such a large class of integrity constraints (including many undecidability results [1]), we take a more pragmatic approach based on a resource-bounded exploration of the consequences of the integrity constraints. Again, while theoretically infeasible, this approach seems to work well in practice.

Example 5 (Running Example: Integrity Constraints) Here are some of the integrity constraints for our running example (cf. Figure 1):

```
Task(_,_,_,P) => Task(P,_,_,_)
Task(P,U,C,_) => Task_int(_,P,U,C,_)
Task_int(_,P,U,C,_) => Task(P,U,C,_)
Task_int(_,_,_,_,A) => Task_int(A,_,_,_,_)
Task(P,Q1,_,_), Task_int(_,P,_,_,A), Task_int(A,Q2,_,_,_) => Q1 = Q2
Task(P,Q,_,_), Task_int(_,P,_,_,A1), Task_int(A2,Q,_,_,_) => A1 = A2

Fd(I,N,_) => Fd_int(_,I,N,_)
Fd(_,_,P) => Task(P,_,_,_)
Fd_int(_,I,N,_) => Fd(I,N,_)
Fd_int(_,_,_,A) => Task_int(A,_,_,_,_)
Fd(F,_,P1), Fd_int(_,F,_,A), Task_int(A,P2,_,_,_,_) => P1 = P2
Fd(F,_,P), Fd_int(_,F,_,A1), Task_int(A2,P,_,_,_,_) => A1 = A2
```

We also assume that the first attribute of every relation is a key. Note that '_' denotes *anonymous variables* in the integrity constraints.

The constraints tell us, for example, that every file descriptor is associated with a single task, and that the task *pointer* in the internal representation of the file descriptor will indeed point to the correct internal task structure.

The Storage Model. To address the particular circumstance of main memory systems, one needs to specify how relations are physically stored in main memory. Recall that the goal is to be able to abstract virtually any data structure representing a collection of main memory *records* as a relation, and to support following "native C pointers" between such structures.

We therefore interpret every predicate in the plan as a collection of main memory record instances (e.g. instances of a C struct type) with the assumption that the first argument is the address of the record (and therefore a key) and that the remaining arguments are stored fields of the record; cf. Figure 1. While this requirement may seem very restrictive it is important to remember that other *conceptual* relations may be linked with these relations using integrity constraints. This can preserve a user's view of the conceptual schema of the embedded system in which physical addresses are not visible.

In particular, consider a predicate $P(x, y_1, \ldots, y_n)$ occurring in a query plan. The necessarily existing *binding pattern* for P (formally defined below) also implies the existence two "schema declarations:"

1. The functional dependency $(\{x\} \rightarrow \{y_1, \ldots, y_n\})$; and
2. The binding pattern $P^{bf\cdots f}$.

Therefore, in conjunction with appropriate inclusion dependencies, the addresses of records can be used to implement foreign-key joins. This fact may affect the way the conceptual schema and the integrity constraints for the main memory system are (or, more likely, should be) designed: we should make sure that we use the "navigational joins" for the most frequent foreign key joins performed in the system.

To define the *accessible* relations, one could give a set of relation names for which the embedded system has the ability to scan the instances of the corresponding relations. We generalize the notion of an accessible relation in the embedded system to take into account varying levels of *accessibility* of relations by using the familiar terminology for describing *binding patterns* in queries.

Definition 6 (Binding Patterns) *Let R be a relation of arity k and p a string in the alphabet $\{f, b\}$ of length k. Then R^p denotes a relation that limits access support to queries in which attributes corresponding to b positions in s are bound to a constant.*

Field Navigation. As pointed out above, one can expect that most accesses to stored tuples can be achieved by following pointers in the cases of key-foreign key constraints, rather than by executing iterators on the encodings of the base tables. This is one place where existing database technology often fails to match a skilled C programmer. However, the **DEMO** system is able to take advantage of the schema constraints and directly generate navigational code. There are two general cases that need to be considered.

- For the first occurrence of a variable in the plan that extracts a value from a corresponding field of a record, we simply generate an *assignment statement*.
- For any later occurrence of the same variable in the plan that corresponds to a value of another field extracted from another (possibly distinct) record, we generate an *if statement* that compares the value of the variable with the value in this field.

These two operations are justified by the $P^{bf\cdots f}$ binding pattern and the dependency $(\{x\} \rightarrow \{y_1, \ldots, y_n\})$. In addition, information about the cost of navigation vs. the cost of scanning a table is available to the query plan generator. The cost difference is usually so high that the plan generator picks navigation in almost all cases.

Iterators for Stored Relations. In addition, for any other binding pattern for P the embedded controller provides a *built-in* iterator that implements it.

Example 7 (Running Example: Binding Patterns) We modify our example to take into account the existence of predefined binding patterns in our example. That is, we are given a set describing the allowed access to relations

$$B = \{ \texttt{Task\_int}^{fffff}, \texttt{Fd\_int}^{fffb} \}.$$

The first pattern corresponds to the fact that Linux provides an iterator that can walk through the tree of all running processes, the second to the fact that each process descriptor provides a pointer to an array of files open by this process.

3.2 Query Compilation

Integrity constraints are crucial to a fundamental part of the DEMO system: the module responsible for query plan generation. The module uses a chase-like [1, 21] procedure that expands a query formula to an equivalent *conjunctive query* according to the schema integrity constraints. The goal is to identify sufficiently many *physically stored* relations in the expansion that enable a concrete means of determining the result of evaluating the original query. Even in cases for which the original query can be evaluated directly, this approach often leads to vastly superior query plans (recall that every attribute dereference is a significant overhead in main memory databases). We describe the general process in three steps, beginning with an introduction to the chase-like expansion procedure. We then show how the expanded query is used to find feasible plans, and also discuss families of constraints for which we can achieve completeness. (It is known that no complete procedure for finding execution plans for the general family of integrity constraints can exist.) At the end of this section, we extend our simple storage model to account for binding patterns of relations in the conceptual schema.

Definition 8 (Query Plan) *Let E be a query and B a set of binding patterns. We say that E satisfies B if E can be ordered in such a way that for each $R \in E$ there is a binding pattern $R^p \in B$ such that all attributes of R marked b appear in a preceding atom (in the order) or are parameter of the query.*

The plan generation algorithm takes as input a set of binding patterns. Note that there may be relations that do not appear in this set (i.e., don't have any allowed binding pattern, which is different from the $b \cdots b$ pattern).

Query Expansion. First we show that a conjunctive query can be *expanded* using the schema information to an equivalent conjunctive query that satisfies given binding patterns (i.e., represents a *query plan*). We use the schema constraints to define a set of rewriting rules on sets of atomic formulas as follows:

Definition 9 (Expansion Rules) *Let I be a set of integrity constraints and Q a set of atomic formulas. We define* $\mathrm{Inf}(I)$ *to be the set of rules*

$$Q \mapsto Q \cup \{R\theta\} \quad \text{for } (P_1 \wedge \ldots \wedge P_k \Rightarrow R) \in I, \text{ and } P_i\theta \in Q$$
$$Q \mapsto Q[x\theta \mapsto y\theta] \text{ for } (P_1 \wedge \ldots \wedge P_k \Rightarrow x = y) \in I, \text{ and } P_i\theta \in Q$$

induced by I; θ stands for renaming of variables *in the integrity constraints to match those in the set Q; for embedded tuple generating constraints we require that variables, that are being substituted for the variable names in the constraint that only appear in the consequent, do not appear in Q.*

We can expand every query Q using the above rules and form a set, $\mathrm{Exp}(Q, I)$, the closure of Q under $\mathrm{Inf}(I)$.

Unfortunately, in the presence of cyclic dependencies in the schema description (e.g., cyclic embedded inclusion dependencies), this set is not finite. Moreover, there is often no upper bound on the size of the expansion for the general constraint families; in Section 4 we discuss restricted families of integrity constraints for which such a bound can be found.

However, unlike many theoretical approaches, we do not immediately abandon the general case. Rather, we use a *resource-bounded approach* to control the depth of the expansion by restricting the number of applications of the $\mathrm{Inf}(I)$ rules. We achieve this by assigning a *rank* to variables in the generated conjunctive query as follows: rank 0 is assigned to variables in the original query, and then, each time a *new* variable is introduced in the expansion of an embedded tuple-generating dependency, the variable is assigned rank one higher than the maximal rank of variables that appear in a precondition of the instance of the applicable dependency. Moreover, if two variables are made equal using a equality-generating constraint, the higher rank is assigned to the result.

We define $\mathrm{Exp}^n(Q, I)$ to be the set of formulas $P(x_1, \ldots, x_k) \in \mathrm{Exp}(Q, I)$ such that $\mathrm{rank}(x_i) \leq n$. It is easy to see that the ranking of variables guarantees that the expansion procedure terminates after finitely many applications of the $\mathrm{Inf}(I)$ rules. The result of expansion, $\mathrm{Exp}^n(Q, I)$, is a set of atoms that represents a *conjunctive query* (defined as a conjunction of all elements of $\mathrm{Exp}^n(Q, I)$). To obtain the original query, it is also necessary to remove by projection all extra variables introduced during the expansion; this can be simply achieved by projecting on the free variables of the original query Q.

Plan Generation. We use the depth-bounded expansion procedure to find a valid query execution plan under the required restrictions of accessibility to relations in the conceptual schema. Our first *naive algorithm* shows that the problem can be essentially reduced to finding a correct join-order among the accessible relations equivalent to the original query. We can therefore reuse off-the-shelf join-order selection and cost estimation algorithms for this step.

Algorithm 10 (Naive Plan Generation)

FindPlan(Query Q, IC I, BP B, integer n) =
  $E := \text{Exp}^n(Q, I)$;
  find min(c) and the corresponding plan P in
    $\{c : (P, c) = \text{CostJoinOrder}(E')$
      where E' subsumes E, $E' \subseteq B$, and $Q \subseteq \text{Exp}^n(E', I)\}$
  or fail if this set is empty;
  return(P);

where CostJoinOrder is a generic join-order selection algorithm.

In this algorithm, we exhaustively construct a plan for every subset of the candidate set E'. In practice, however, the selection of the candidate and the actual join-order selection can be interleaved, which makes the complexity of an exhaustive join-order selection dominate the combined complexity.

Lemma 11 (Soundness) *Let Q be a query, I a set of integrity constraints, B a set of accessible relations, and P a plan produced by* FindPlan(Q, I, B, n). *Then P implements Q on every database that satisfies I and B.*

When applied to our running example, the algorithm gives the following result:

Example 12 The expansion of the query $Q = \{\texttt{Task(P,\_,C,\_), Fd(F,N,P)}\}$ from Example 2 with respect to the integrity constraints in Example 5 is

$$\text{Exp}^1(Q, I) = \{\texttt{Task(P,\_,C,\_)}, \texttt{Fd(F,N,P)}, \texttt{Task\_int(A,P,\_,C,\_)}, \ldots\}$$

(the variables P, C, and N have rank 0, the rest of the variables have rank 1). Note the use of the equality generating constraint that matched the last attribute of `Fd_int` with the first attribute of `Task_int`. One of the sets E' that pass the conditions in Algorithm 10 is

$$E' = \{\texttt{Task\_int(A,P,\_,C,\_), Fd\_int(\_,F,N,A)}\}.$$

This corresponds to the final (and indeed optimal) plan

$$\pi_{\{N,C\}}\texttt{Fd\_int(\_,F,N,A)} \bowtie \texttt{Task\_int(A,P,\_,C,\_)}$$

that utilizes the ability to navigate the address field in the `Fd_int` structure (cf. section 3.2).

In this setting, plan P_1 from Example 12 is no longer feasible, and the only feasible plan is P_3 with the join ordering

$$\pi_{\{N,C\}}\texttt{Task\_int(\_,P,\_,C,A)} \bowtie \texttt{Fd\_int(\_,F,N,A)}$$

that utilizes the ability to scan all file descriptors associated with a particular task by traversing an array of `Fd_int` structures associated with a given task (indicated by the binding pattern). Indeed, this is the situation in an unmodified Linux kernel.

In the actual system, we use a simple greedy plan generation algorithm that can be abstracted by the following pseudo code:

Algorithm 13 (Basic Greedy Plan Generation in DEMO)

GreedyFindPlan(Query Q, IC I, BP B, integer n) =
  $E := \mathrm{Exp}^n(Q, I)$;
  $E' := \emptyset$;
  do
    $C :=$ GetCandidates($E - E', B$);
    if $C = \emptyset$ fail;
    $E' := E' \cup \{\mathrm{BestCandidate}(C)\}$;
  until ($Q \subseteq \mathrm{Exp}^n(E', I)$)

Note that by replacing the greedy *next goal* selection by a frontier generation, the greedy algorithm is converted to a branch-and-bound algorithm with only minimal changes to finding the feasible plans and the termination condition.

The actual algorithm used in **DEMO** avoids unnecessary recomputation of the $Q \subseteq \mathrm{Exp}^n(E', I)$ condition for the growing set E' using an incremental approach based on an efficient graph representation of conjunctive queries.

Code Generation. We now outline the final step in processing a query in the **DEMO** system: the generation of a low level target language code that implements an iterator protocol for a given query.

The plan generation algorithms guarantee that the plan given to the code generator is *feasible*—that all relations mentioned in this query plan satisfy the binding patterns. Consequently, at least one binding pattern must exist for every relation symbol mentioned in the plan.

Example 14 In our running example, the binding patterns are as follows:

$$B = \{\, \texttt{Task\_int}^{ffffff}, \texttt{Fd\_int}^{ffffb} \}.$$

These declarations lead to two iterators: the `Task_iter()` providing references to all tasks and `Fd_iter(A)` providing references to file descriptors for a particular task. We also consider the case in which an additional iterator `Fd_iter()` providing access to all files (independently of tasks) is available.

The system provides a standard library for common situations (e.g., array scans, linked list scans, various search trees, etc), that can often be used directly or with only minor customization.

Example 15 Now we can finish our example. The following pseudo-code is generated by the **DEMO** system[3]. First, assume that we are able to access all file descriptors directly:

```
Query(C,N) =
    for f ∈ Fd_iter do
        return(f.ProcA->Command, f.Name);
    od;
```

[3] In reality, the resulting C code must be "inverted" to form a proper iterator that follows **DEMO** iterator protocol.

The code illustrates the use of *scanning a base table* (2) and *stored field navigation* (3). However, if the file descriptors must be accessed through the corresponding process, the generated code is as follows:

```
Query(C,N) =
    for t ∈ Task_iter do
        for f ∈ Fd_iter(t) do
            return(t.Command, f.Name);
        od;
    od;
```

Additional features include the possibility of using indices (where available), short-circuited evaluation of selection conditions, etc.

4 Restricted Families of Integrity Constraints

While the plan generation algorithm works quite well in practice, it cannot guarantee to find a feasible execution plan, even if one exists. However, for several restricted families of integrity constraints, one can determine a resource limit $n \in N$ based on the structure and size of the set of integrity constraints describing the schema that is large enough for Algorithm 10 to always find a plan if one exists. Finding such families has been the target of intensive research both in the database and the theorem proving communities (cf. Section 6 for discussion of related work in this area). One of these families of constraints can be characterized as follows: All the *tuple-generating* constraints are inclusion dependencies that fall into one of the following categories:

1. Full inclusion dependencies (all variables in the consequent of a constraint also appear in one of the antecedents);
2. Unary inclusion dependencies (at most one variable in the consequent of a constraint appears in any of the antecedents); or
3. Acyclic inclusion dependencies (no embedded constraint depends on itself).

It is also necessary for the equality-generating constraints to be *functional dependencies* with non-empty left-hand sides. For this restricted families of integrity constraints, our approach is guaranteed to find a plan if one exists:

Lemma 16 (Completeness) *Let Q be a query. Then, for every set of restricted integrity constraints I and every set of binding patterns B, there is $n \in N$ such that exactly one of the following situations occurs:*

1. $\mathrm{FindPlan}(Q, I, B, n)$ *finds a plan for Q if there is a valid plan, or*
2. $\mathrm{FindPlan}(Q, I, B, n)$ *fails if there is no valid plan for Q.*

The value of n depends only on I and B.

The value of n is bounded by a polynomial in the size of I for the family of full and unary inclusion dependencies, and is exponential in the size of I if one allows acyclic (non-unary) inclusion dependencies [1].

Assuming that only single-column statistics for join order selection are used (this disables the possibility of inferring that a particular binary relation is a contraction), the plan produced by Algorithm 10 is also optimal. As with completeness, this result cannot be generalized to the case of unrestricted integrity constraints. However, in those cases Algorithm 10 finds the best plan for the given resource bound n.

5 System Architecture

The overall architecture of the query processing component of the DEMO system is presented in Figure 5. There are essentially four components involved in the translation of a query to either C or Java code.

- The query is first parsed and translated to a canonical form in an extended relational algebra. Various standard simplifications to the query are performed at this time.
- The *complex query plan generator* then translates the high level algebraic expression to a low level form that expresses the chosen query plan. During this translation, the generator searches for conjunctive query subexpressions whose plan generation is delegated to the *conjunctive query plan generator*.
- The conjunctive query plan generator is a concrete realization of algorithm 13 outlined above. Optimized access plans for conjunctive subquery fragments of the general query are obtained by this module using the schema information. We elaborate on how this all works and on the interaction between the two plan generators below.
- The *code generator* finally converts the resulting query plan to the target language (currently, C or Java). In particular, the code generated from a plan consists of a pair of functions that realize an iterator protocol.

5.1 Plans for Complex Queries

The algorithms introduced in Section 3.2 focus on optimizing conjunctive queries. However, in a realistic setting, an optimizer must also handle negations and disjunctions in queries.

For general (first-order) queries, however, there is no hope for completeness of any planning algorithm (e.g., an equivalent of Lemma 16) due to the undecidability of satisfaction for relational calculus (there may be a feasible *empty* plan, but we are not able to detect it). Therefore, as with other approaches, we assume that for valid queries we must be able to find a plan for every atomic subformula (base table), or at least for every conjunctive subquery. In addition, the system uses several heuristics that propagate information between the conjunctive fragments of the original query to improve the chances of finding an execution plan.

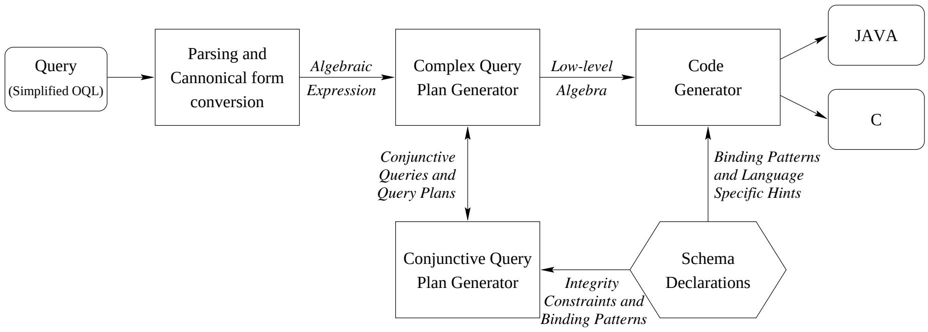

Fig. 5. DEMO Query Processing Subsystem Architecture.

The **DEMO** compiler uses a two-tier architecture to handle the general case. A complex query submitted to the *complex query plan generator* is processed inductively starting from the leaves and working towards the root as follows:

1. Find maximal conjunctive subqueries in the leaves of the given query;
2. Consult the *conjunctive query plan generator* and get plan(s) for these subqueries;
3. Generate subplans from these results for their immediate parents;
4. Replace the subqueries in the original query for which plans have been generated in Step 3 by views (that represent these plans); and
5. Repeat Steps 1 to 4 until the root of the original formula is reached.

In addition, information about variable bindings and other properties that hold in the answers to subqueries (and were determined by the plan generation procedures) are also recorded by the algorithms. This information is used to improve the plan in the upper levels of the query.

6 Summary Comments

In this paper, we have proposed a new application area for database systems, the class of *embedded control programs* (ECPs), and have presented an overview of how query processing is accomplished for such systems in the **DEMO** query optimizer. The focus of our discussion on query processing and our main technical contribution is the concept of *resource bounded plan generation.* Largely because of its ability to use binding patterns together with various families of tuple and equality generating dependencies, this new search paradigm has enabled us to deal effectively with two important problems:

1. The reverse engineering of existing legacy ECPs to the alternative architecture depicted in Figure 3; and
2. Competing effectively with expert programmers working directly in C.

In the first case, the variety of dependencies and binding patterns supported by our tools is sufficient to capture access to an arbitrary collection of C types, and to relate this kind of information to a conceptual schema. In the second case, we realized early on in the **DEMO** project that extensive semantic query optimization and "fine grained" plan algebras would be essential features of any query processing tools that could compete with expert programmers.

6.1 Related Work

Perhaps the most relevant ongoing work relates to various efforts to provide tools for information integration [2, 8, 10, 13–15]. Of particular relevance is more recent work along this line that assume binding patterns limit how information sources can be accessed [7, 17]. (See [20] for an excellent overview of this line of work, circa 1997.) Although the query semantics may differ (e.g. "find as much as you can" semantics), efforts along this line share the common goal of attempting to provide a uniform conceptual interface to existing legacy information sources. In particular, [6, 16] approach the integration problem, including physical index selection, using similar techniques to those presented in this paper. Our work, however, differs in scale (e.g. we worry about accessing an array of C structs; others worry about accessing a file system or web document) and in the need, we think, to worry as much as we do about performance. In addition, the techniques described in this paper can be easily extended to handling duplicate semantics of queries [12, 11] (which is beyond the scope of this paper).

Also of relevance is the extensive body of work on compiling logic programs [5, 9, 18, 19, 22]. Although a large focus of this work is on recursive query optimization (recursive queries do not seem particularly useful for ECPs) and there is only a limited effort to handle navigating legacy data structures, this work shares our own goal of effectively raising the level of programming to languages such as Datalog.

Acknowledgments. The authors gratefully acknowledge support by Natural Sciences and Engineering Research Council of Canada, by Communication and Information Technology of Ontario, and by Nortel Networks Ltd.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. Yigal Arens and Craig A. Knoblock. Sims: Retrieving and integrating information from multiple sources. In *SIGMOD*, pages 562–563, 1993.
3. Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Harold Bohme. *Linux Kernel Internals*. Addison-Wesley, 1997.
4. Remy Card, Eric Dumas, and Franck Mevel. *The Linux Kernel Book*. John Wiley and Sons, 1998.
5. Danette Chimenti, Ruben Gamboa, Ravi Krishnamurthy, Shamim A. Naqvi, Shalom Tsur, and Carlo Zaniolo. The ldl system prototype. *Transactions on Data and Knowledge Engineering*, 2(1), 1990.

6. Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *25th International Conference on Very Large Data Bases, VLDB'99*, pages 459–470, 1999.
7. Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD Conference*, 1999.
8. Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In *SIGMOD Conference*, pages 539–542, 1997.
9. Parke Godfrey, John Grant, Jarek Gryz, and Jack Minker. Integrity constraints: Semantics and applications. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, chapter 9. Kluwer, 1998.
10. Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *International Conference on Very Large Data Bases*, pages 276–285, 1997.
11. Vitaliy L. Khizder, David Toman, and Grant Weddell. On Decidability and Complexity of Description Logics with Uniqueness Constraints. In *International Conference on Database Theory*, pages 54–67, 2000.
12. Vitaliy L. Khizder, David Toman, and Grant Weddell. Reasoning about Duplicate Elimination with Description Logic. In *Computational Logic 2000 (DOOD,'00)*, pages 1017–1032, 2000.
13. Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *ACM Symposium on Principles of Database Systems*, pages 95–104, 1995.
14. Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *International Conference on Very Large Data Bases*, pages 251–262, 1996.
15. Yannis Papakonstantinou, Ashish Gupta, and Laura M. Haas. Capabilities-based query rewriting in mediator systems. *Distributed and Parallel Databases*, 6(1):73–110, 1998.
16. Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *ACM SIGMOD International Conference on Management of Data*, pages 273–284, 2000.
17. Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *ACM Symposium on Principles of Database Systems*, pages 105–112, 1995.
18. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL—control, relations and logic. In *International Conference on Very Large Data Bases*, pages 238–250, 1992.
19. Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD International Conference on Management of Data*, pages 442–453, 1994.
20. Jeffrey D. Ullman. Information integration using logical views. In *International Conference on Database Theory*, pages 19–40, 1997.
21. Moshe Y. Vardi. *Fundamentals of Dependency Theory*, pages 171–224. Computer Science Press, 1987.
22. David H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA, October 1983.
23. Grant Weddell. Main memory database management for embedded control applications, 1998. CITO Research Project, Government of Ontario.