Physical Data Independence: Query and Update Compilation

System description: A Compiler for Database Queries and Updates
Keywords: Automated Reasoning, Planning and Optimization

Abstract
We present a compiler and optimizer that compiles relational queries and updates with respect to a given database schema to source code in C. The schema itself is flexibly captured as a set of (range-restricted) first-order constraints that describe both the logical/conceptual understanding of the data and, with the help of minimal extra-logical annotations, the physical layout of the data in storage devices.

1 Introduction
We present a compiler with a novel architecture deriving from earlier work reported in (Toman and Weddell 2017) and (Hudek, Toman, and Weddell 2015) that compiles user requests to manipulate data expressed over logical or conceptual views of data to low level plans for executing the request in C.

The compiler incorporates an optimizer, and is based on a variation of the Craig Interpolation theorem, with reasoning realized via a modified analytic tableau proof procedure producing results that describe a space of candidate solutions for a given user request compilation problem. This enables a decoupled A*-based planner, with the help of an external cost model, to subsequently arbitrate among alternative candidate low level plans in C without the need to support backtracking in the proof procedure.

The tableau procedure is implemented as a virtual machine that uses a compiled and optimized byte-code generated from the database schema and a user query or update request. The result of the compilation is then linked with support libraries implementing the low-level data access interface to the concrete data structures of materialized relations, and then linked with the remainder of the user’s application. Additional extensions and infrastructure ensuring, among other things, that duplicate semantics is correctly accounted for, and that the resulting code respects binding patterns, are also incorporated.

Section 2 that follows introduces how information is organized by the compiler in a way that accommodates a very general form of physical data independence, and introduces a working example to illustrate this. An overview of the system architecture follows in Section 3. Section 4 then illustrates how standard physical design patterns are accommodated, and Section 5 how the compiler handles user requests that are updates. A further self-contained case study for a simple main memory system is given in Figure 4.

2 Physical Data Independence

Physical data independence is a property of database and knowledge representation systems that allows users, when formulating requests to manipulate data, queries and updates, to understand the data at a conceptual or logical level that is completely devoid of any knowledge of how concrete data structures encode data, or how such data structures enable more efficient algorithms for computing answers to some varieties of request patterns. Compiling queries and updates is the problem of reformulating them to alternative low-level plans invoking such algorithms via interfaces to concrete data structures in a way that realizes the most efficient possibilities for such plans.

In reviewing the compiler for solving this problem, we focus on illustrating how it works for queries and then for updates by appeal to a simple case study for a hypothetical student information system.

Example 1 (Graduate and Undergraduate Students). Students in this system are represented as pairs consisting of their student id and name and are classified as undergraduate or graduate. This situation is captured by the following declarations linking binary relations student, ugrad, and grad (in a computer-readable notation for first order formulae):

% disjoint coverage
student(x,y) <-> (ugrad(x,y) or grad(x,y)),
ugrad(x,y) and grad(x,z) -> bot,
% student id is a key
student(x,y) and student(x,z) -> y=z,

We will assume in this example that users intend/wish to interact with this system using the ugrad and grad relations. However, the actual data is stored in a (CSV) file astudent that can be viewed as a ternary relation with triples consisting of a record id (virtual, representing the position of the actual tuple in the file) and the student’s id and name. In addition the system keeps an ordered file, agrad, of record ids of graduate student records in the astudent file. This situation is captured by the following constraints formally describing a physical design of the system.

student(x,y) <-> ex(r,astudent(r,x,y)),
grad(x,y) <-> ex(r,astudent(r,x,y) and agrad(r)),

and agrad(r)),

and agrad(r)),
In addition, we mark the two relations, astudent and agrad as access paths (this signifies we can access data in instances of these symbols).

This expression is straightforwardly expanded to C code that implements an open-fetch-close cursor to communicate parameters/results between queries and user applications.

Observe that the plan for a subquery has complementation. An ability to horizontally partition data, called *sharding* in modern systems, is also straightforward with this architecture:

% access paths: hp{1-3} (actual partitions), sel{1-3} (key range checks, bp=3)
% coverage

Here, the *sel* built-in and fixed relations (e.g., range comparisons) that determine which tuples belong to which storage hpi partition. Note that the *sel* do not have to be

3 System Architecture

The architecture of the compiler is outlined in Figure 2 and consists of a translator from range restricted first-order formulae (Abiteboul, Hull, and Vianu 1995) (describing the queries and the logical and physical designs) to bytecode (Toman and Weddell 2017) that is the basis for implementing a high performance version of a split tableau prover (Hudek, Toman, and Weddell 2015). The result of the prover are *closing sets* indirectly defining a space of valid rewritings. An A*-based planner selects the resulting query plan from this space with the help of a cost model describing the performance characteristics of the individual access paths. A feature of this architecture is that the planner does not need to backtrack any steps taken by the theorem prover. The plan is then used to generate C source code.

4 Standard Design Patterns

We now illustrate how this system architecture supports many features typically found in physical designs of information systems. A prototypical relation name *table* is used for user visible (logical) relations.

**Primary and Secondary Indices.** We start by creating a physical design that models a primary index for our data (basetable, assuming x is a primary key) enhanced by a secondary index, *index_y*, on the values of y). Note the use of the *record ids* in the primary index (not visible to users) to define the secondary index.

% access paths: basetable (primary index), index_y (secondary index)

% keys (lossless decomposition)

The result of the prover consists of a translator from range restricted formulae (Abiteboul, Hull, and Vianu 1995) (describing the queries and the logical and physical designs) to bytecode (Toman and Weddell 2017) that is the basis for implementing a high performance version of a split tableau prover (Hudek, Toman, and Weddell 2015). The result of the prover are *closing sets* indirectly defining a space of valid rewritings. An A*-based planner selects the resulting query plan from this space with the help of a cost model describing the performance characteristics of the individual access paths. A feature of this architecture is that the planner does not need to backtrack any steps taken by the theorem prover. The plan is then used to generate C source code.
instance of table
This design requires records to correspond to tuples in the
records(p,x,y,z) \rightarrow \text{pages}(p)
% two level design:
% records (read records from page; \text{bp}=1)
% access: \text{pages} (read disk pages)
e.g, by the following design:
architecture also allows one to capture such two-level store,
accessed via \text{pages}
Two-level Store
hp partitioning and replication among the disjoint, thus allowing for an arbitrary combination of data
partitioning and replication among the hpi.

**Two-level Store** Finally, information stores must accommodate *external storage*, such as disks, that must in turn be
accessed via pages typically containing 100s of tuples. The
architecture also allows one to capture such two-level store,
e.g, by the following design:
% access: pages (read disk pages)
% records (read records from page; \text{bp}=1)
table(x,y,z) \leftrightarrow \text{ex}(p, \text{records}(p,x,y,z))
% two level design:
records(p,x,y,z) \rightarrow \text{pages}(p)
This design requires records to correspond to tuples in the
instance of table, and requires that a plan *must* first read a page before accessing such records. This is ensured by a
mechanism called *binding patterns* (bp) that ensure, in this case, that records access path can only be used when there is an actual value for \text{p}; for details see, e.g. (Benedikt et al. 2016) or (Toman and Weddell 2011).

Given two tables with the 2-level design, one can formulate a query \{x, y \mid \exists z, u, v. \text{table1}(x, z, u) \land \text{table2}(y, z, v)\} that is compiled to the following plan:
query(v11, v12) \leftrightarrow
ex(v13, ex(v14, ex(v15, ex(v16, ex(v17, and ( and ( and (pages2(v16), pages1(v17) ), records2(v16, v12, v13, v14) , records1(v17, v11, v13, v15) ) ) ) ) ) )

Observe that the plan manifests the so-called *Block Nested
Loops Join* without any need for a special operator to do this.
Indeed, in all the above examples, the architecture makes it
unnecessary to write any additional C code beyond the code
for the access paths, such as scanning an array or reading a
page from a disk.

### 5 Updates
To accommodate user requests that are updates consisting of inserts and deletes to logical relations, while still supporting
physical data independence, the system must *translate* such requests to insertions and deletions that are executed against the access paths of the underlying physical design. This is accomplished by viewing such requests as a *query compilation task* as depicted in Figure 3. The left and right halves in the figure correspond to two copies of the original design that capture the state of the system before and after the update. The user request is then specified by sets of tuples to insert and delete for user relations ($U^+$ and $U^-$) and use constraints to relate the before and after states (i.e., $U^{old} \lor U^+ \leftrightarrow U^{new} \lor U^-$). The task is to synthesize plans for the relations $A^+, A^-$ that link the before and after states of access paths in the same way. Hence, the set of access paths in the update scenario are the original access paths, with the old superscript, and the relations corresponding to the user update request ($U^+, U^-$). Once these plans are synthesized, code is generated that modifies the access paths based on the results produced by the plans.

**Example 3 (Updating Students)**. Updates in our introductory example with respect to the *update specification* corresponding to the one in Example 1 yields the following query computing the tuples that need to be removed from the astudent access path (file) in terms of the user request:

- **astudent_minus(v11, v12, v13)** \leftrightarrow
  and ( and ( and (astudent_cld(v11, v12, v13),
  and ( not ( ugrad_plus(v12, v13) ),
  not ( grad_plus(v12, v13) )
  ) or ( ugrad_minus(v12, v13),
  grad_minus(v12, v13) )
  ) ) )

Note that students that are moved between grades and undergrads are not affected. The insertion into this access path is similarly synthesized as follows:

- **astudent_plus(v11, v12, v13)** \leftrightarrow
  and ( or ( ugrad_plus(v12, v13),
  grad_plus(v12, v13) )
  student_cmp(v12, v11) )

Here, tuples to be inserted require a *record id*. The system uses so-called *constant complement* relations (Bancilhon and Spyratos 1981), in this case *student_cmp*, to provide such values. Operationally, the complement relation corresponds to allocating space for new tuples.
The physical design below describes a main-memory database that stores information about employees (their ids, names and numbers of departments they work for) and about departments (their numbers, names and manager ids). The physical design stores employee records as C structs in a linked list (ea), each of which contains an employee number and name fields and a pointer to the department record this employee works in (dept). The department records contain department number and name fields and a pointer to an employee record of the manager (mgr). Note that there isn’t a linked list (or an equivalent data structure) that allows to find all department records directly. The design is specified as follows:

% record layout of emp and dept records and fields: ea/da addresses,
% all attributes functional, "num" is a key; "dept" and "mgr" are pointers;
% access paths: ea/1/0 (linked list of employee records),
% ea_num, ea_name, ea_dept, da_num, da_name, da_mgr/2/1 (field extract "->" operator)
%
ea(e) -> ex(y,ea_num(e,y)), ea_num(y,x) and ea_num(z,x)-> y=z,
ea(e) -> ex(y,ea_name(e,y)), ea_name(e,y) and ea_name(e,z)-> y=z,
ea(e) -> ex(y,ea_dept(e,y)), ea_dept(e,y) and ea_dept(e,z)-> y=z, ea_dept(e,d) -> da(d),
da(d) -> ex(y,da_num(d,y)), da_num(y,x) and da_num(z,x)-> y=z,
da(d) -> ex(y,da_name(d,y)), da_name(d,y) and da_name(d,z)-> y=z,
da(d) -> ex(y,da_mgr(d,y)), da_mgr(d,y) and da_mgr(d,z)-> y=z, da_mgr(d,e) -> ea(e),
%
% user predicates
employee(x,y,z) <-> ex(e,baseemployee(e,x,y,z)), % record addresses
% ea(e) <-> ex([x,y,z],baseemployee(e,x,y,z)),
ea_num(e,x) <-> ex([y],baseemployee(e,x,y,z)),
ea_name(e,y) <-> ex([x,z],baseemployee(e,x,y,z)),
ex(d,ea_dept(e,d) and da_num(d,z)) <-> ex([x,y],baseemployee(e,x,y,z)),
% department(x,y,z) <-> ex(d,basedepartment(d,x,y,z)), % record addresses
da(d) <-> ex([x,y,z],basedepartment(d,x,y,z)),
da_num(d,x) <-> ex([y],basedepartment(d,x,y,z)),
da_name(d,y) <-> ex([x,z],basedepartment(d,x,y,z)),
ex(e,da_mgr(d,e) and ea_num(e,z)) <-> ex([x,y],basedepartment(d,x,y,z)),
%
% business logic: managers work for their own departments
% da_mgr(x,e) and ea_dept(e,y) -> x=y % pointer-based version
%
Plan for the \{ (e, n, d) | employee(e, n, d) \} query.
employee_num_name_dept(vl1, vl2, vl3) <-
ex(vl4, ex(vl8),
and ( and ( and ( and ( ea(vl4),
ea_name(vl4, vl12) ), ea_num(vl4, vl11) ),
ea_dept(vl4, vl18) ), da_num(vl18, vl3) )
)

The plan is translated to C yielding the following source code
for a ∈ ea return (a->num, a->name, a->dept->num).

In reality the code will be a cursor with an open-fetch-close protocol, but the actual code that executes will be essentially the same as above in the fetch part of the iterator.

Note that the plan contains a seemingly unnecessary atom da_mgr(vl4, vl17). This atom, however, guarantees that each department is returned only once (as we can deduce from equational constraints that there can be one one manager) and, in turn, avoids the need for de-duplicating the result (making the query much more efficient than the still correct and smaller query without this atom).

Figure 4: Main Memory Physical Design with Pointers.
References


