Projective Beth Definability and Craig Interpolation for Relational Query Optimization

(Material to Accompany an Invited Talk at SOQE 2021)

David Toman and Grant Weddell

Cheriton School of CS, University of Waterloo, Canada
{david,gweddel}@uwaterloo.ca

Abstract. Accessing information using a high-level data model or ontology has been a long-standing objective of research communities in several areas. The underlying idea of separating a logical/conceptual view of how information is understood by users from a physical view of the layout of data in data structures, called *physical data independence*, has been a focus of research for more than fifty years. Here, we explore the issues connected with optimizing and executing relational queries and updates in this setting. In particular, we consider how to find appropriate reformulations of user queries over the physical design, and show how these ideas naturally relate to first-order definability and interpolation.
The talk elaborates on how logic-based approaches can be used to capture both the high-level conceptual views of information and the low-level physical layout of data. Based on such a formalism, we present the design of a relational query optimizer based on Craig interpolation that allows users to compile both queries and updates to low-level code that operates directly over a physical encoding of the data. The ultimate objective of this design is to produce low-level code that performs comparably with hand-written code in low-level programming languages such as C.

1 The Problem

Abstraction and high-level approaches to software development have been among the most significant factors in increasing both the productivity of developers and the quality of the applications they develop. Efforts along these lines date back to the late 60s and early 70s when the concepts of *data independence* [1,2] and *abstract data types* (ADTs) [13] were introduced. The concepts share the goal of enabling application programmers to develop applications with respect to a purely *abstract* or *conceptual* understanding of the application's data or information, an understanding that is entirely *independent* of the concrete data structures and related algorithms that encode the data on physical storage devices. Indeed, modern file systems are examples: programmers manipulate files

via operations that are entirely devoid of any need to understand low-level disk layout issues.

The idea of data independence gained popularity in the 70s, both in the area of programming languages, e.g., with languages such as SETL [8,14,10], and in the area of information and database systems mainly due to the development of the *relational model* (RM) [5] with accompanying data manipulation language(s) [6] based on *first-order logic*. However, with the passing of time (50 years later), approaches that use lower levels of abstraction, such as C or the various recent NoSQL database systems, have often displaced approaches that promote data independence, often at the cost of increasing development time and/or lowering the quality of deployed systems. The most common reason for this phenomenon is the need for massive scaleability and flexibility, capabilities often missing in systems with high levels of abstraction such as RM.

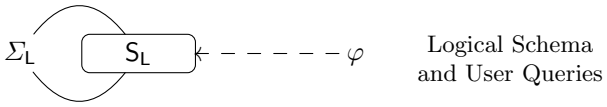
The goal of this presentation is to outline a direction of research that, in the realm of database and information systems, enables simultaneous high level abstractions at the user level and extreme flexibility at the *physical design* level, that is, in the choice of concrete data structures and their access algorithms. Indeed, our ultimate goal of this direction of research is to compete with *hand-written* code in low-level languages such as C, while providing the high level of abstraction in the original RM [5] that are not yet fully realized in existing relational database management systems.

2 Data Independence (through an Example)

We begin outlining how *data independence* can be understood more formally in terms of *first-order (relational) signatures* and integrity constraints (i.e., first-order sentences over these signatures).

2.1 The Logical Schema

The *logical schema* is a first order signature $\mathsf{S_L}$ and an accompanying set of integrity constraints Σ_L that are specific to the domain of the application (that require user familiarity). The situation can be depicted as follows:



The users interacting with the data use *queries*, in our case open first order formulae over $\mathsf{S_L}$, to formulate their requests (we will deal with modifying the data later in Section 4). There are two important observations that follow from this arrangement:

1. The user only requires familiarity with $\mathsf{S_L}$ and Σ_L to be able to develop applications; and

2. The user can assume that the actual data is a single interpretation on $\mathsf{S_L}$ that is a model of Σ_L over which her requests are evaluated (i.e., without the need to comprehend subtle issues related to logical entailment and/or belief revision). We call such interpretations *instance*s of the schema.

Example 1 (Logical Schema)
We will use the following logical schema formulated in SQL as our running example.

```
CREATE TABLE employee (          CREATE TABLE department (
  num      INTEGER NOT NULL,       num      INTEGER NOT NULL,
  name     CHAR(20),               name     CHAR(50),
  worksin INTEGER  NOT NULL        manager INTEGER NOT NULL,
  PRIMARY KEY (num),               PRIMARY KEY (num),
  FOREIGN KEY (worksin)            FOREIGN KEY (manager)
        REFERENCES department            REFERENCES employee
)                                )
```

The (instances of) `employee` and `department` relation declarations are intuitively meant to store information about employee numbers, names and departments they work in, and about departments, their names and managers. In our formalism, this is simply a syntactic sugar for a signature

$$\mathsf{S_L} = \{\texttt{employee}/3, \texttt{department}/3\}$$

(where "$/i$" indicates predicate arity) and integrity constraints

$$\begin{aligned}\Sigma_\mathsf{L} = \{\, & \texttt{employee}(x, y_1, z_1) \wedge \texttt{employee}(x, y_2, z_2) \rightarrow y_1 = y_2 \wedge z_1 = z_2, \\ & \texttt{employee}(x, y, z) \rightarrow \exists u, v.\texttt{department}(z, u, v), \quad \ldots \qquad\qquad \}\end{aligned}$$

stating that employees are identified by their `num`ber, that they must work for exactly one department, and so on.

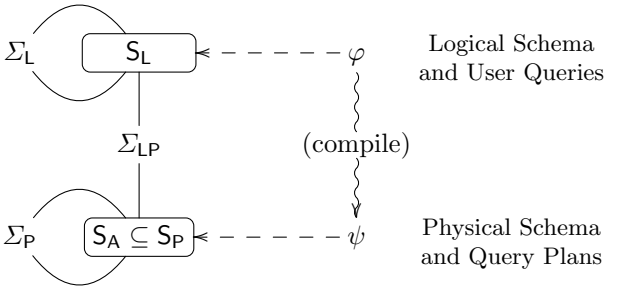
The ability of specifying integrity constraints in Σ_L allows one to go beyond what is available in typical implementations of the relational model, for example:

- managers are employees that manage a department (a view)
  $\texttt{manager}(x, y, z) \leftrightarrow \texttt{employee}(x, y, z) \wedge \exists u, v.\texttt{department}(u, v, x)$
- managers work in their own departnemts (business rule)
  $\texttt{employee}(x, y, z) \wedge \texttt{department}(u, v, x) \rightarrow z = u$
- workers and managers partition employees (partition)
  $\texttt{employee}(x, y, z) \leftrightarrow (\texttt{manager}(x, y, z) \vee \texttt{worker}(x, y, z))$
  $\texttt{manager}(x, y, z) \wedge \texttt{worker}(x, y, z) \rightarrow \bot$

Observe that this extends the signature of the logical schema with additional predicate symbols `manager`/3 and `worker`/3 that a user can now reference in queries.

2.2 The Physical Schema

We use a similar strategy to define the *physical schema* where we again use relational signatures and constraints for this purpose. However, these symbols will correspond to actual *data structures* that are called *access path*s in database literature. These access paths correspond to various ways to access data, ranging from dereferencing a pointer in main memory or extracting a field from a main memory record (abstracted by binary predicate symbols whose interpretations are address-value pairs) to using main memory data structures such as linked lists (again abstracted by appropriate predicate symbols) to reading data from external storage, and to communicating with other agents. The situation can be again depicted as follows:



Note that there can be additional *helper* predicate symbols in S_{P} in addition to the access paths S_{A}.

There are two issues with this strategy that must be addressed:

1. Will it suffice to associate access paths (data structures and their associated search algorithms) with predicate symbols?
2. Is it reasonable to also think about generated code using access paths as formulae (ψ above)?

To address the 1st question, we annotate the symbols in S_{A} with so called *binding patterns* [19] indicating which arguments of the particular access path must be *bound to a value* before the access path can be *executed*. We indicate this by an additional integer in the signature specification, for example "`pointer-nav`/2/1" indicates that the access path representing address-value pairs in main memory can be only used when we have a value for the first component (i.e., an address). The implementation then consists of a simple statement for dereferencing this address to produce the a value of the second argument. This observation also leads to restrictions on the form of ψ [16].

Example 2 (Physical Schema)
We illustrate the first issue by defining the *physical schema* for our running example. Our physical design consists of a linked list of `emp`loyee records that use pointers (references) to indicate `dep`artment records an employee works in. In a similar fashion, the `dep`artment records use a pointer to indicate which

employee is a manager. The records in a Pascal-like notation are as follows:

```
record emp of                    record dept of
       integer   num                    integer   num
       string    name                   string    name
       reference dept                   reference mgr
```

In our formalism this looks as follows: we define the following predicates to be associated with access paths (i.e., in $\mathsf{S_A}$):

- `empfile`/1/0: set of *addresses* of `emp` records; this access path abstracts navigating a linked list (of `emp` records) in main memory.
- `emp-num`/2/1: a set of address of `emp` records paired with the `emp` numbers; this access path corresponds to extracting a field (`num` in this case) from an `emp` record (given an address of such a record). The access paths `emp-name`/2/1 and `emp-dept`/2/1 and `dept-num`/2/1, `dept-name`/2/1, and `dept-mgr`/2/1 similarly abstract the field extraction of the remaining fields from the `emp` and `dept` records.

We also use two auxiliary predicates `emp`/1 and `dept`/1 to stand for the sets of addresses of `emp` and `dept` records. Integrity constraints ($\Sigma_{\mathsf{P}} \cup \Sigma_{\mathsf{LP}}$) then capture the properties of instances of the physical schema and how they relate to the logical schema. For example the fact that records have appropriate fields can be specified as follows:

$\texttt{emp}(e) \rightarrow \exists d.\texttt{emp-dept}(e,d)$	`emp` records have a `dept` field
$\texttt{emp-dept}(e,d_1) \wedge \texttt{emp-dept}(e,d_2) \rightarrow d_1 = d_2$	the `dept` field is functional
$\texttt{emp-dept}(e,d) \rightarrow \texttt{dept}(d)$	the value of the `dept` field is a pointer to a `dept` record

For full listing of the constraints see Appendix A. This completes our description of the physical schema for our example.

2.3 Queries and Plans

Now we are ready to give an answer to our 2nd question, how to interpret formulae as *query plans*. This is straightforward: atomic formulae are mapped to (the code associated with) access paths and logical connectives and quantifiers to "control flow code fragments" as follows:

atomic formula	$\mapsto$	access path (a `get-first` / `get-next` iterator)
conjunction	$\mapsto$	nested loops join
existential quantifier	$\mapsto$	projection (with optional duplicate information)
disjunction	$\mapsto$	concatenation
negation	$\mapsto$	simple complement

For a formula to correspond to a plan (i.e., executable code), it is also necessary to obey binding patterns [16]. While such a procedural interpretation of atoms

and logical connectives might seem over simplistic, we discuss in Section 3.2 below how this simple fine-grained interpretation suffices for most of the hard-coded solutions in other database systems.

Example 3
We illustrate this framework by worked examples of several user queries together with possible query plans for these queries over our running physical design case.

Q1: List employee numbers, names, and departments ($\texttt{employee}(x, y, z)$). We can show that this user query is *logically equivalent* under the integrity constraints to the following formula over $\mathsf{S_A}$:

$$\exists e, d.\texttt{empfile}(e) \wedge \texttt{emp-num}(e, x) \wedge \texttt{emp-name}(e, y) \\ \wedge \texttt{emp-dept}(e, d) \wedge \texttt{dept-num}(d, z)$$

Assuming our formulas as plans mapping, this formula would correspond to the following C-like code (with trivial simplifications and inlining of the $\texttt{ea-xxx}(x, y)$ access paths to y `:=` x`->xxx`):

```
for e in empfile do
        x := e->num;  y := e->name;
        d := e->dept; z := d->num; return (x, y, z);
```

Note also that the formula above satisfies the binding patterns associated with the access paths used as it retrieves the address of an `emp` record *before* attempting to extract the values of it's fields.

Q2: List worker numbers and names ($\exists z.\texttt{worker}(x, y, z)$). Again, this query is equivalent to the following formula over $\mathsf{S_A}$:

$$\exists e, d.\texttt{empfile}(e) \wedge \texttt{emp-num}(e, x) \wedge \texttt{emp-name}(e, y) \\ \wedge \texttt{emp-dept}(e, d) \wedge \neg\texttt{dept-mgr}(d, e)$$

Note that a negation, $\neg\texttt{dept-mgr}(d, e)$, is required, and that there is no negation in the query nor in the schema that provides any direct clue that it is needed. (We are not aware of any system that can synthesize this plan, that is, that compiles queries using this framwork.)

Q3: List all department numbers and their names ($\exists z.\texttt{department}(x, y, z)$). Finding a plan for this query is more difficult since we do not have a direct way to "scan" `dept` records. However, it is an easy exercise to verify that the following two formulae over $\mathsf{S_A}$ are logically equivalent to the query:

$$\exists d, e.\texttt{empfile}(e) \wedge \texttt{emp-dept}(e, d) \\ \wedge \texttt{dept-num}(d, x) \wedge \texttt{dept-name}(d, y)$$

(relying on the constraint that "departments have at least one employee")

$$\exists d, e.\texttt{empfile}(e) \wedge \texttt{emp-dept}(e, d) \\ \wedge \texttt{dept-num}(d, x) \wedge \texttt{dept-name}(d, y) \wedge \texttt{dept-mgr}(d, e)$$

(relying on the constraint that "managers work in their own departments")

Both correspond to plans. However, while the second might seem to be less efficient than the first, a query optimizer should prefer it on the grounds that, in this case, the quantified variables d and e are *functionally determined* by the answer variable x. Hence, the final projection generated for the second has no need to *eliminate duplicate answers*. This is not the case for the first of these formulae since it would return a copy of the department information for *every* employee of the department should duplicate elimination in the final projection not be performed.

Many other problems and issues in physical design and query plans can be revolved in this framework, including standard RDBMS physical designs (and more), access to search structures (index access and selection), horizontal partitioning/sharding, column store/index-only plans, hash-based access to data (including hash-joins), multi-level storage (aka disk/remote/distributed files), materialized views, etc., all without any need for coding in C beyond the need for the generic specifications of `get-first` / `get-next` templates for concrete data structures [16].

3 Interpolation and Query Optimization

Now we turn our attention to the description of a query compiler/optimizer that, given the logical and physical schemata and a user query, generates a query plan that correctly implements the user request.

3.1 What Queries *Make Sense*?? (to users)

However, before we begin, it is important to resolve what queries make sense to a user who presumes there is a *single interpretation* of symbols in $\mathsf{S_L}$ at any point in time, no matter how it is represented/stored physically. To satisfy to this expectation, the queries that make sense should have *the same answer* in *every* model of the overall physical design Σ in which the interpretation of $\mathsf{S_A}$ is fixed, that is, where the stored data is always the same. This arrangement also guarantees that artifacts facilitating efficient storage and retrieval of information *won't* be leaked in the results of queries (since they do not exist in the logical view of the data). The consequence of this observation is that either

1. there are situations in which a seemingly reasonable user query cannot be answered (that would be the case for *Q3* in Section 2.3, were the constraint "departments have at least one employee" absent from the schema), or
2. queries must adhere to syntactic restrictions in which, e.g., symbols corresponding to built-in operations cannot be used completely freely, and physical designs must also adhere to syntactic restrictions such as so-called *standard designs* (i.e., where an access path exists for every logical table in Σ_L, thus guaranteeing that every user query can be answered).

To make the definition of *sensible queries* more formal, we appeal to a well-known notion of *definability:*

Proposition 4 (Projective Beth Definability [4])
Let $\Sigma \cup \{\varphi\}$ be a FO theory over symbols in L and $L' \subseteq L$. Then t.f.a.e.:

1. For all M_1, M_2 models of Σ such that $M_{1|L'} = M_{2|L'}$, and all $\boldsymbol{a}$ tuples of individuals, it holds that $M_1 \models \varphi[\boldsymbol{a}]$ iff $M_2 \models \varphi[\boldsymbol{a}]$, and
2. φ is equivalent under Σ to some formula ψ in L'.

We say that φ *is explicitly definable w.r.t.* Σ *and* L'.

Definability (over $\mathsf{S_A}$ w.r.t. Σ) formally captures the idea of (physical) data independence, the illusion of a single interpretation of the logical schema that satisfies integrity constraints that is presented to the users, and therefore provides the means of determining which queries can be answered over a particular physical design.

The first question is how to test for definability. The following observation reduces this test to *determining* whether a particular formula constructed from the user query is entailed by a theory constructed from the schema: φ is explicitly definable (w.r.t. Σ and over $\mathsf{S_A}$) if and only if

$$\Sigma \cup \Sigma' \models \varphi \rightarrow \varphi' \tag{1}$$

where Σ' (φ') is Σ (φ) in which symbols *NOT in* $\mathsf{S_A}$ are *primed*, respectively.

The next question is how to find a plan for a given query. Our observations on how formulae can be interpreted as query plans in Section 2.3 then mostly reduces query compilation to a search for the formula ψ in Proposition 4(2). To find ψ, we rely on a variant of the following result [7]:

$$\text{If } \Sigma \cup \Sigma' \models \varphi \rightarrow \varphi' \text{ then there is } \psi \text{ s.t. } \Sigma \cup \Sigma' \models \varphi \rightarrow \psi \rightarrow \varphi'$$

where $\mathcal{L}(\psi) \subseteq \mathcal{L}(\mathsf{S_A})$. Here, ψ is called the *Craig interpolant*. Moreover, we can extract any such ψ from a Tableau proof of (1) in linear time [9].

3.2 Architecture

The above discussion might seem to solve the query compilation problem. However there are additional issues that need to be addressed:

1. The search for interpolants and their implied query plans must consider that alternative but logically equivalent plans might have vastly different performance characteristics.[1] Hudek *et al.* [11] introduce an approach that separates the tableau-based search for interpolants from the cost-based[2] exploration of alternative query plans.

[1] This holds even for conjunctive formulae: hence database literature often focuses on the so-called join-order problem [15].

[2] Cost-based query optimization is the cornerstone of relational systems [15]; advancements in the area of query plan cost estimation are easily incorporated in this framework.

2. *Binding patterns* for access paths (see Section 2.2) further restrict the space of executable query plans (and in turn of *sensible* queries). Benedikt *et al.*[3] have shown how the binding patterns can be accommodated in the search for interpolants (i.e., in the search for proofs of definability).
3. In addition, during the search for optimal query plans, we consider the impact of duplicate elimination as illustrated by plans for *Q3* in Section 2.3. A detailed account for this facet of query compilation can be found in [16,18].

Figure 1 sketches an architecture of a query compilation/optimization system that addresses the above concerns. The compiler preprocesses the given schema

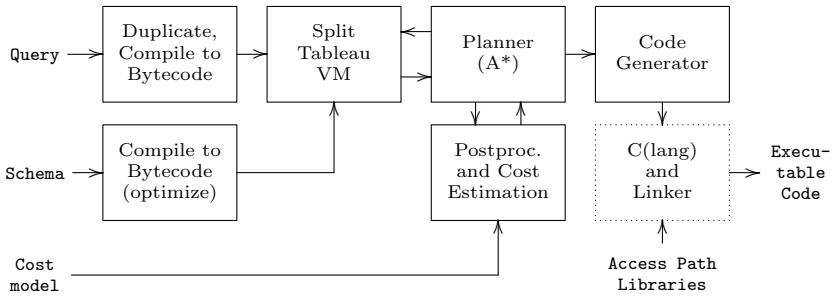


Fig. 1: Compiler Architecture

and the user query into a normal form and generates a *bytecode* that drives a *virtual machine-based* (VM) tableau theorem prover [17]. Unlike standard theorem provers, including those that can generate interpolants [12], the tableau VM generates an intermediate representation of a *space of equivalent interpolants* called *closing sets* [11]. Closing sets are then explored by an A*-based planner to find a query plan with the lowest estimated cost. The planner also explores ways to avoid duplicate elimination in the process. The planner is then followed by a code generator that produces the ultimate query plans in a form of C source.

4 Updates

In this section, we sketch how the problem of compiling updates on a logical design can be translated to the problem of compiling queries on a related logical design, thus enabling the same framework above to also be used to compile inserts, updates and deletes on logical tables.

As already mentioned in Section 2.1, user updates are formulated with respect to the logical schema ($\mathsf{S_L}$ and Σ_L). Moreover, physical data independence presents the user with a illusion that he is modifying an instance of $\mathsf{S_L}$ by adding/removing ground tuples to/from the interpretations of symbols in $\mathsf{S_L}$. This process can be formalized in three parts as follows:

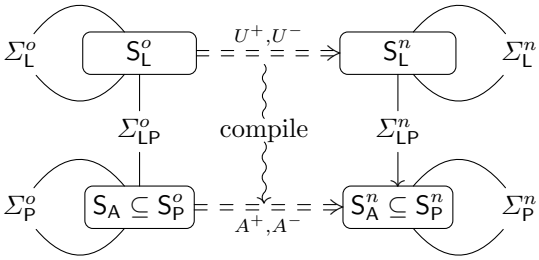
1. For every symbol $R \in \mathsf{S_L}$, introduce two additional symbols, R^+ and R^-, whose (disjoint) interpretations correspond to the ground tuples the user wants to add or remove to/from the current instance;
2. The *updated* instance is then defined by executing an simultaneous assignments $R := (R \cup R^+) - R^-$ for all $R \in \mathsf{S_L}$; and
3. At the end of the assignment the new interpretation must be a model of Σ_L.

The symbols R^+ and R^- are commonly called the *delta relation*s and Part 3 of this process on user updates ensures so-called *consistency preserving transactions*.

To convert the update problem to the problem of synthesizing plans for queries, consider two copies of the schema Σ, in which all symbols are superscripted by ^{o} and ^{n}, respectively. The intuition is that the ^{o} and ^{n} symbols correspond to the interpretations of $\mathsf{S_L}$ before and after the update. The actual assignment (Part 2 of the above process) can be then captured as additional formulae

$$R^o(\boldsymbol{x}) \vee R^-(\boldsymbol{x}) \leftrightarrow R^n(\boldsymbol{x}) \vee R^+(\boldsymbol{x})$$

for each $R \in \mathsf{S_L}$ as depicted below.



In the same way, the changes to access paths in $\mathsf{S_A}$ can be captured by analogous constraints, as depicted in the lower half of the figure. Thus, user inserts, updates and deletes on logical tables (comprising a transaction) are mapped to a *definability* question of the following form:

> Is A^n (or A^+, A^-) definable in terms of A^o_i and U^+_j, U^-_j (user updates) for every access path $A \in \mathsf{S_A}$, given the instance of all access paths in $\mathsf{S_A}$ and of all *delta relations* for $\mathsf{S_L}$?

A positive answer to this question yields a *update plan* that applies the delta relations corresponding to the access paths to their current interpretations.

5 Summary

We have outlined how projective Beth definability can be used in database and information systems to facilitate physical data independence. Moreover, we have

shown how a variation on Craig interpolation can be used to compile and optimize user queries and user updates that are formulated over a logical schema to an executable plan over a fine-grained physical design. There are many avenues for further research and development, including: (1) admitting more powerful languages for user requests, such as languages with aggregation; (2) enhancements to the tableau provers, as well as alternatives such as superposition-based provers; and (3) improvements to the planning component of query compilation responsible for exploring the search space of alternative query plans.

References

1. Charles W. Bachman. CODASYL data base task group: October 1969 report, 1969.
2. Charles W. Bachman. Summary of current work - ANSI/X3/SPARC/Study Group–Database Systems. *FDT Bull. ACM SIGFIDET SIGMOD*, 6(3):16–39, 1974.
3. Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. *Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2016.
4. Evert Willem Beth. On Padoa's method in the theory of definition. *Indagationes Mathematicae*, 15:330–339, 1953.
5. E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
6. E. F. Codd. Relational completeness of data base sublanguages. *IBM Research Report*, RJ987, 1972.
7. William Craig. Three uses of the Herbrand-Genzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 1957.
8. Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Program. Lang. Syst.*, 1(1):27–49, 1979.
9. Melvin Fitting. *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science. Springer Publishers, 1996.
10. Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Trans. Program. Lang. Syst.*, 5(1):26–45, 1983.
11. Alexander K. Hudek, David Toman, and Grant E. Weddell. On enumerating query plans using analytic tableau. In *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21-24*, pages 339–354, 2015.
12. Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2009.
13. Barbara H. Liskov and Stephen N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.
14. Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data structures in SETL programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, 1981.

15. Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.
16. David Toman and Grant E. Weddell. *Fundamentals of Physical Design and Query Compilation*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
17. David Toman and Grant E. Weddell. An interpolation-based compiler and optimizer for relational queries (system design report). In *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, Maun, Botswana, May 7-12, 2017*, 2017.
18. David Toman and Grant E. Weddell. Using feature-based description logics to avoid duplicate elimination in object-relational query languages. *Künstliche Intell.*, 34(3):355–363, 2020.
19. Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, 1985.

A Constraints for the Running Example

The following listing is a complete specification of constraints needed for our running example. Note that some of the constraints in Section 2.1 are *entailed* by the constraints below (and are thus omited).

```
%
% logical schema (entailed constraints omited)
%
% a (virtual) view for managers
manager(x,y,z) <-> (employee(x,y,z) and ex(n,department(z,n,x))),
%
% disjoint partition of employees to managers and workers
employee(x,y,z) <-> (manager(x,y,z) or worker(x,y,z)),
manager(x,y,z) and worker(x,u,v) -> bot,
%
% businness logic: managers work for their own departments
(department(x,y,z) and employee(z,u,w)) -> x=w,
%
% physical schema and mappings
%
% design of emp and dept structs; emp/dept addresses, fields functional
emp(e) -> ex(y,emp_num(e,y)),  emp_num(e,y) and emp_num(e,z)-> y=z,
                               emp_num(y,x) and emp_num(z,x)-> y=z,
emp(e) -> ex(y,emp_name(e,y)), emp_name(e,y) and emp_name(e,z)-> y=z,
emp(e) -> ex(y,emp_dept(e,y)), emp_dept(e,y) and emp_dept(e,z)-> y=z,
                               emp_dept(e,d) -> dept(d),
%
dept(d) -> ex(y,dept_num(d,y)), dept_num(d,y) and dept_num(d,z)-> y=z,
                                dept_num(y,x) and dept_num(z,x)-> y=z,
dept(d) -> ex(y,dept_name(d,y)), dept_name(d,y) and dept_name(d,z)-> y=z,
dept(d) -> ex(y,dept_mgr(d,y)), dept_mgr(d,y) and dept_mgr(d,z)-> y=z,
                                dept_mgr(d,e) -> emp(e),
```

```
%
% linked list for ea's and record attributes
%
empfile(x) <-> emp(x),
%
% user predicates and mappings
%
employee(x,y,z) <-> ex(e,baseemployee(e,x,y,z)),
%
emp(e)                              <-> ex([x,y,z],baseemployee(e,x,y,z)),
emp_num(e,x)                        <-> ex([y,z],baseemployee(e,x,y,z)),
emp_name(e,y)                       <-> ex([x,z],baseemployee(e,x,y,z)),
ex(d,emp_dept(e,d) and dept_num(d,z)) <-> ex([x,y],baseemployee(e,x,y,z)),
%
department(x,y,z) <-> ex(d,basedepartment(d,x,y,z)),
%
dept(d)                            <-> ex([x,y,z],basedepartment(d,x,y,z)),
dept_num(d,x)                      <-> ex([y,z],basedepartment(d,x,y,z)),
dept_name(d,y)                     <-> ex([x,z],basedepartment(d,x,y,z)),
ex(e,dept_mgr(d,e) and emp_num(e,z)) <-> ex([x,y],basedepartment(d,x,y,z))
```