Kalpa Publications in Computing

Volume 1, 2017, Pages 1–10

LPAR-21S: IWIL Workshop
and LPAR Short Presentations

An Interpolation-based Compiler and Optimizer for Relational Queries

(System Design Report)

David Toman and Grant Weddell

University of Waterloo, Waterloo, ON, Canada
{david,gweddell}@cs.uwaterloo.ca

Abstract

We outline the implementation of a query compiler for relational queries that generates query plans with respect to a database schema, that is, a set of arbitrary first-order constraints, and a distinguished subset of predicate symbols from the underlying signature that correspond to access paths. The compiler is based on a variant of the Craig interpolation theorem, with reasoning realized via a modified analytic tableau proof procedure. This procedure decouples the generation of candidate plans that are interpolants from the tableau proof procedure, and applies A*-based search with respect to an external cost model to arbitrate among the alternative candidate plans. The tableau procedure itself is implemented as a virtual machine that operates on a compiled and optimized byte-code that faithfully implements reasoning with respect to the database schema constraints and a user query.

1 Introduction

In [9], we introduced an approach to finding plans for *first order* (FO) queries based on Craig interpolation [5] and analytic tableau [7]. The main technical contributions of this earlier work were twofold. The first was the introduction of so-called *conditional tableau* that abstract reasoning common to all query plans for a given user query Q and a *physical design* given by a 2-tuple, $\langle \Sigma, \mathsf{Phys} \rangle$, in which Σ is a collection of range-restricted FO sentences comprised of constraints, dependencies, mapping rules, and so on, and in which Phys is a subset of predicate names occurring in Σ denoting predicate names that are access paths, that is, names of actual data structures storing the data.

The second contribution was the introduction of an architecture in which plan search is made an orthogonal process that interacts with the construction of conditional tableau via so-called *closing sets*. Such sets correspond to a selection of ground atoms with predicate names occurring in Phys, and signal the existence of at least one executable query plan based on data structures corresponding to these atoms. Closing sets and how plan search is conducted ensure any query plan found by plan search must then be logically equivalent to an interpolant that derives from a proof based on analytic tableau of

$$\Sigma^L \cup \Sigma^R \cup \Sigma^{LR} \models Q^L \rightarrow Q^R,$$

T.Eiter, D.Sands, G.Sutcliffe and A.Voronkov (eds.), LPAR-21S (Kalpa Publications in Computing, vol. 1), pp. 1–10

where Σ^L and Q^L (resp. Σ^R and Q^R) denote Σ and Q in which every occurrence of a non-logical symbol P has been replaced by P^L (resp. P^R), and where the connection to access paths in a physical design, hereon called *physical atoms*, is uniformly captured by Σ^{LR} given by

$$\{\forall \mathbf{x}.P^L(\mathbf{x}) \leftrightarrow P(\mathbf{x}), \forall \mathbf{x}.P^R(\mathbf{x}) \leftrightarrow P(\mathbf{x}) \mid P \in \mathsf{Phys}, \mathbf{x} = \mathsf{Fv}(P)\}.$$

Note that connecting Σ^L and Σ^R via Σ^{LR} in this way is a crucial first step in factoring tableau reasoning shared by alternative query plans for a user query, and is essential to the notion of conditional tableau mentioned above (see [9] for details; note, however, that the variant of the conditional tableau introduced in this paper—called henceforth *split tableau*—is a non-trivial refinement of [9]). The search for an interpolant outlined in [9] proceeds in two major steps:

1. Build tableau T^L using $\Sigma^L \cup \{Q^L(\mathbf{a})\}$, and tableau T^R using $\Sigma^R \cup \{Q^R(\mathbf{a}) \rightarrow \bot\}$; and
2. Generate query plan candidates, orthogonally, using Σ^{LR} and the above-mentioned closing sets for (T^L, T^R).

This paper outlines a number of refinements to each of these steps that were incorporated in the current implementation, in particular:

(i) only ground atoms are stored in the tableau (no complex formulae are created/stored);

(ii) complex formulae are compiled to bytecode common to both T^L and T^R that is then interpreted by a virtual machine;

(iii) tableau branching is compactly encoded in tags attached to these atoms;

(iv) complex terms are encoded by fixed-size constants (the terms can be reconstructed if needed); and

(v) plan generation is based on A* search [8] with a powerful heuristic derived from database cost models.

These refinements enable the current implementation to be several orders of magnitude faster than a semi-naive implementation of the original system reported in [9], and are the focus of the remainder of the paper.[1] In particular, our presentation of these refinements is organized as follows:

- we develop a *virtual* machine suitable for generating closing sets for a given query and schema (see Section 3);
- we develop a complementary query/constraint to byte-code compiler (see Section 4); and
- we propose and develop a *query planner* based on A* and a novel powerful heuristic guiding the search for optimal query plans (see Section 5).

We also outline how the current proposal interfaces with additional techniques and supporting infrastructure, including query plan post-processing which is essential to an accounting of so-called *binding patterns* and for efficiently managing duplicates in query results. While these techniques are not themselves described in this paper, we briefly discuss how the proposed compiler can take advantage of them.

[1] The authors are unaware of any benchmarks available or published that can be used to compare the performance of query compilers or optimizers in general, and thus cannot see any reasonable path towards truly meaningful experiments beyond summarizing, as we do, the performance benefit over a previous more naive implementation.

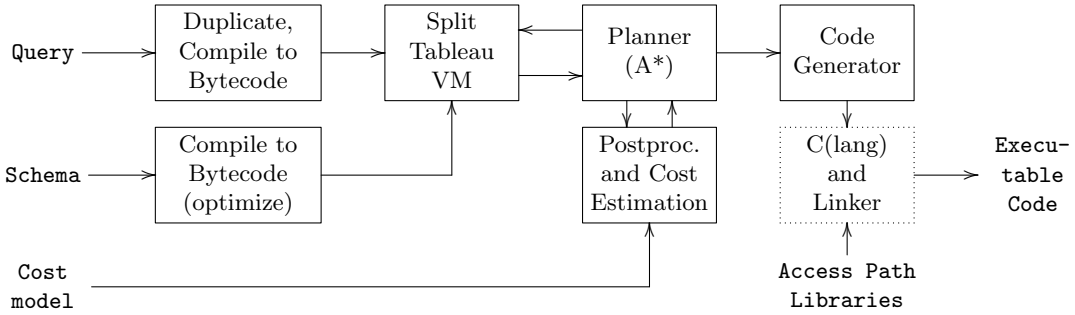

Figure 1: Compiler Architecture

In summary, we focus in this paper on salient issues that we have found to underlie any implementation of the split tableau-based query compiler originally presented in [9]. Note that formal correctness proofs are rather long and tedious, and, in the end, simply reduce the issues to the correctness of the more naive (but more compact) techniques in [9]; they are therefore omitted in this paper for sake of brevity.

2 System Architecture and Components

The overall architecture of the compiler is depicted in Figure 1, and consists of two main components:

1. A virtual machine that interprets a byte-code generated from the schema constraints and the user query, the *Split Tableau Virtual Machine* (VM), and produces the above-mentioned *closing sets*: information that is sufficient to find alternative query plans without any need for further reasoning with sentences comprising the physical design; and

2. An A*-based *Planner* that uses a *cost model* and a *plan post-processor* to choose an optimal plan among the alternatives.

Note that the *Planner* operates entirely independently of physical design sentences that capture dependencies, mapping rules, and so on, since all information needed to generate plans is encapsulated by closing sets computed by the *Split Tableau VM*. Note also that the planner can request the VM to deepen the tableau search. This is a necessary feedback because of the expressiveness of sentences comprising a schema/physical design (the constraint/query language is not recursive, and hence it is not possible to have an apriori bound on the tableau expansion that ensures finding an optimal query plan).

The two main modules are accompanied by a *Byte-code compiler* and a *Code Generator*. The *Byte-code compiler* converts range-restricted first order formulae (both schema constraints and user queries) and generates the appropriate byte-code used by the *Split Tableau VM*. The *Code generator* converts query plans (still expressed as annotated first-order formulae) to actual imperative code, e.g., in C or LLVM (see, e.g., code fragments in [11]). The following sections now describe each of the modules.

```
queue: a priority queue of (pointers to) conditional atoms;
iterators: an array of (pointers to) conditional atoms;
while (queue is not empty && tableau is not closed) do
    iterators[0] = pop(queue); execute(iterators[0].code);
```

Figure 2: Split Tableau Construction.

3 The *Split Tableau VM*

To begin, we introduce the virtual machine used to execute a byte code to simulate split tableau derivations. The state of the split tableau is captured simply as two sets of *conditional atoms*, one for T^L and one for T^R:

Definition 1 (Conditional Atoms)**.** *Let R be a ground atom (arguments are constant symbols), $\{P_1, \ldots, P_n\}$ a set of ground* physical atoms*, and $\{i_1 : j_1, \ldots, i_m : j_m\}$ a set of pairs of positive integers such that $i_{k_1} \neq i_{k_2}$ for $0 < k_1 < k_2 \leq m$. We call "$R[P_1, \ldots, P_n]\{i_1 : j_1, \ldots, i_m : j_m\}$" a* conditional atom *$R$ depending on ground physical atoms $P_1, \ldots, P_n$ (called* dependencies*) and belonging to tableau* branches *described by $i_1 : j_1, \ldots, i_m : j_m$.*

Note that the atom R can also be the symbol $\bot$ (arity 0) standing for falsehood. Intuitively, the ground atom R belongs to the tableau T's branch identified by the sequence of pairs $i_1 : j_1, \ldots, i_m : j_m$, where the first component of each pair identifies a branch point and the second component the number of a branch emanating from this branch point, and depending on the atoms $P_1, \ldots, P_n$ already existing in T in the same branch.

Our approach generates *conditional atoms* to account for variant tableau proofs of the same theorem: this is essential to facilitate subsequent plan generation and optimization. In the actual implementation, both atoms $P_1, \ldots, P_n$ and the branch identifiers $i_1 : j_1, \ldots, i_m : j_m$ are sorted arrays to facilitate subsequent set operations, such as set union. The conditional atoms are then stored in balanced search trees to facilitate their lookup by name and a prefix of their arguments. There are two search trees per predicate symbol, one for T^L and one for T^R.

Figure 2 gives the main control loop of the tableau construction. The priority queue is organized by the complexity of conditional atoms (i.e., by a weighted sum of the number of branches involved, of the number of conditional atoms, and of the age of the Skolem constants). The `code` field attached to conditional atoms contains the byte code executed whenever a new conditional atom (for a particular predicate) is created, and is defined in terms of the following *abstract instruction set* for the *Split Tableau VM* (and is common to both T^L and T^R):

Definition 2 (Abstract Instruction Set)**.**

`JOIN` P n i_1 a_1 ... i_n a_n: This instruction finds the first conditional atom with the predicate symbol P whose first n arguments match the arguments a_j of the (preceding) `iterators`$[i_j]$. If such an atom is found, it is added to the `iterators` array as the next element and the VM proceeds with the execution of subsequent code; otherwise the VM backtracks. If the `JOIN` instruction is reached by backtracking, the next conditional atom for P is looked up and the above procedure is repeated.

`MKATOM` P i_1 a_1 ... i_n a_n: This instruction creates a new conditional atom for P whose arguments are filled (left to right) with constants copied from arguments a_j of the (preceding) `iterator`$[i_j]$. The dependencies and branches for the conditional atom are defined as the unions (merges) of the dependencies and branches of the individual iterators, respectively. The created atom is then inserted in the appropriate search tree and, if it

had not already existed, also in the priority `queue`. Execution then continues with the next instruction. (The same applies to any new atoms encountered in the next three instructions.)

`MKOR` P j`:` The instructions creates a conditional atom P that is identical to the conditional atom `iterators`[0] but its branches contain additional branch identifier $i : j$ where i is a global value containing the number of branching points so far and is incremented when $j = 0$ (i.e., the leftmost branch of every branch point must be labeled $i : 0$).

`MKSKOLEM` P`:` This instruction creates a conditional atom P whose arguments consist of a new Skolem constant followed by the arguments of the conditional atom `iterators`[0]. The index of the new constant determines its *age*. The dependencies and branches are shared with `iterators`[0].

`MKPHYS:` This instruction creates a new (physical) conditional atom *in the other* tableau. This atom shares the predicate name and arguments of `iterators`[0], depends on itself and is associated with an empty set of branches.

`SEQ` s `byte-code-1 byte-code-2:` This instruction allows for composing two, independently backtracking parts of the bytecode: it first executes *byte-code-1* by transferring control to the next instruction, and then `byte-code-2` by transferring control to next+s byte code.

`RET:` This instruction simply backtracks to the most recent `JOIN` or `SEQ`, or returns to the top-level control loop.

Note that, unlike the `MKATOM` instruction that genuinely depends on conjunctions of conditional atoms, the `MKOR` and `MKSKOLEM` instructions depend *only* on the original conditional atom (`iterators`[0]): this is intentional to reduce the number of new Skolem constants and branches due to eliminating conditional atoms derived multiple times (since these will never be reinserted in the tableau nor in the priority queue). Similarly, the `MKPHYS` instruction only makes sense for an *already existing* physical atom.

4 Transforming Schema and Query to VM Code

To use the instruction set defined above, we argue that (range-restricted) first order formulae are *compiled* to the above instructions in such a way that executing these byte codes (as defined by the algorithm in Figure 2) simulates correct tableau expansion.

It is straightforward to see that every (closed) range-restricted formula in first-order logic can be represented as an equivalent set of formulae in one of the following three forms necessary to consider for *compilation* (up to the introduction of new auxiliary predicate symbols).[2] The compilation of these three forms to abstract bytecode for the *Split Tableau VM* is as follows:

1. $A_1(\mathbf{x_1}) \wedge \ldots \wedge A_k(\mathbf{x_k}) \rightarrow B_1(\mathbf{y_1}) \wedge \ldots \wedge B_\ell(\mathbf{y}_\ell)$**:** For each $A_i(\mathbf{x_i})$, we generate the code fragment

$$\begin{array}{l}\texttt{JOIN}\ A_1\ n_1\ \mathit{args}(\mathbf{x_1})\ \ldots\ \texttt{JOIN}\ A_{i-1}\ n_{i-1}\ \mathit{args}(\mathbf{x_{i-1}})\\ \texttt{JOIN}\ A_{i+1}\ n_{i+1}\ \mathit{args}(\mathbf{x_{i+1}})\ \ldots\ \texttt{JOIN}\ A_k\ n_k\ \mathit{args}(\mathbf{x_k})\\ \qquad\qquad \texttt{MKATOM}\ B_1 \mathit{args}(\mathbf{y_1})\ \ldots\ \texttt{MKATOM}\ B_\ell\ \mathit{args}(\mathbf{y}_\ell)\ \texttt{RET}\end{array}$$

where n_j is the length of the prefix of variables in $\mathbf{x_j}$ that have appeared in one of $\mathbf{x_i}$, for $i < j$, and $\mathit{args}(\mathbf{x_j})$ is the list of pairs $(i\ a)$ describing the position in which the particular

[2] For range-restricted formulae that start with an existential quantifier we need to introduce an auxiliary unary predicate symbol and insert an appropriate atom to the initial tableau. In practice this case does not occur (and we don't cover it here).

variable has occurred (i corresponds to the subscript of $\mathbf{x_i}$ and a to the offset of the variable from the left).[3]

2. $A(\mathbf{x}) \rightarrow \exists y.B(y,\mathbf{x})$**:** We generate "`MKSKOLEM` B `RET`".

3. $A(\mathbf{x}) \rightarrow B_1(\mathbf{x}) \vee \ldots \vee B_k(\mathbf{x})$**:** We generate "`MKOR` B_1 `0` $\ldots$ `MKOR` B_k k `RET`".

In the above, A_is are predicate symbols, B_is are predicate symbols or $\bot$, and $\mathbf{x}$, and $\mathbf{y}$ are lists of variables. We also assume that the formulae do not contain constants and are closed universally. In addition, for every physical predicate we generate "`MKPHYS`". To generate the ultimate code for a predicate A, we collect all the code fragments for this predicate defined above and connect them in a single byte code stream using the `SEQ` instruction.

User Query. We assume that the user query is atomic and of the form $Q(\mathbf{x})$.[4] We then add the byte code "`JOIN` Q' k `0 0` $\ldots$ k k `MKATOM` $\bot$" to the byte code for Q, and insert a conditional atom $Q(\mathbf{a})[]\{\}$ in T^L and in `queue`, and $Q'(\mathbf{a})[]\{\}$ in T^R, where $\mathbf{a}$ are distinct constants substituted for $\mathbf{x}$ (this is justified by Skolemizing the negation of $Q^L \rightarrow Q^R$ in $\Sigma^L \cup \Sigma^R \cup \Sigma^{LR} \models Q^L \rightarrow Q^R$). In this way, whenever $Q(\mathbf{a})$ is derived in T^R, it will match the conditional atom Q' and yield $\bot$ (originating from the negation of Q for T^R). Correctness of the byte code and the overall procedure reduces to the case explored in [9] and is based on tedious case analysis.

Example 3. A database schema $\Sigma = \{S(x) \leftrightarrow U(x) \vee \exists y.G(x,y), G(x,y) \rightarrow \neg U(x)\}$ stating that Students are either Undergraduate or Graduate (with an advisor), but not both, normalizes to $\{G(x,y) \rightarrow S(x), U(x) \rightarrow S(x), S(x) \rightarrow U(x) \vee G_1(x), G_1(x) \rightarrow \exists y.G_2(y,x), G_2(y,x) \rightarrow G(x,y), U(x) \wedge G(x,y) \rightarrow \bot\}$ (utilizing auxiliary symbols G_1 and G_2). The code generated is:

S: `SEQ 2 MKPHYS RET MKOR` U `0 MKOR` G_1 `1 RET`
G: `SEQ 2 MKPHYS RET SEQ 5 MKATOM` S `0 0 RET JOIN` U `1 0 0 MKATOM` $\bot$ `RET`
G_1: `MKSKOLEM` G_2 `RET`
G_2: `MKATOM` G `0 1 0 0 RET`
U: `SEQ 5 MKATOM` S `0 0 RET JOIN` G `1 0 0 MKATOM` $\bot$ `RET`

Note that only the S and G predicates are assumed to be in Phys (that our physical design does *not* explicitly store the set of undergraduate students).

Tableau Closure. The next issue to consider is when to stop expanding the tableau (i.e., running the bytecode) and consider generating query plans. This crucially depends on the above-mentioned notion of *closing sets* which we now define:

Definition 4 (Closing Sets)**.** *A* closing set S *for a tableau* T *is a smallest set of ground physical atoms such that if* $R[P_1,\ldots,P_n]\{B\}$ *is used to close branch* B *then (i)* $\{\neg R, P_1, \ldots, P_n\} \subseteq S$*, and (ii) if* $i : j \in B$ *then a conditional atom belonging to branch* $i : j'$ *for all* $j' \neq j$ *applicable to branch point* i *must also be a part of* S*.*

Closing sets for (T^L, T^R) *is a pair of sets* $\mathsf{ClosSet}(T^L)$ *and* $\mathsf{ClosSet}(T^R)$*, the first containing all closing sets for* T^L*, and the second containing all closing sets for* T^R*.*

[3] We require that a *prefix* of arguments of $\mathbf{x_j}$ variables appears in the previous atoms. This allows us to efficiently utilize the search trees for conditional atoms in the `JOIN` instructions at the cost of having to duplicate certain atoms with varying argument orderings, which is preferable to quadratic searches needed otherwise.

[4] Given an non-atomic query φ with free variables $\mathbf{x}$, we simply add the formula $\forall \mathbf{x}.Q(\mathbf{x}) \leftrightarrow \varphi$ to Σ.

P :	L_P	R_P
$R(\mathbf{a})$:	$\{\{\neg R(\mathbf{a})\}\}$	$\{\{R(\mathbf{a})\}\}$
$P_1 \wedge P_2$:	$L_{P_1} \cup L_{P_2}$	$\{S_1 \cup S_2 \mid S_1 \in R_{P_1}, S_2 \in R_{P_2}\}$
$P_1 \vee P_2$:	$\{S_1 \cup S_2 \mid S_1 \in L_{P_1}, S_2 \in L_{P_2}\}$	$R_{P_1} \cup R_{P_2}$
$\neg P_1$:	R_{P_1}	L_{P_1}
$\exists x.P_1[x/a]$:	L_{P_1}	R_{P_1}

Figure 3: Plan Fragments and their Closing sets.

Example 5. *Closing sets for our running example with respect to the query $U(x)$ (asking for all undergraduate students) are as follows:*

$$\mathsf{ClosSet}(T^L) = \{\{\neg S(s0)\}, \{G(s0, s2)\}\} \quad \text{and} \quad \mathsf{ClosSet}(T^R) = \{\{\neg G(s0, s2), S(s0)\}\}$$

and are constructed using conditional atoms $B(s0)[]\{\}$ and $\bot[G(s0, s2)]\{\}$ that appear in T^L, and $\bot[S(s0)]\{2:0\}$ and $G(s0, s2)[S(s0)]\{2:1\}$ that appear in T^R.

Deriving the closing sets every iteration would be expensive: we use a heuristic to make this test infrequently, typically after thousands of tableau expansions. Also, existence of the closing sets alone does not guarantee that the split tableau can be closed with the help of Σ^{LR}. We use the following construction (inspired by propositional resolution):

$$(C, D, R), (C', D', R') \in \mathcal{S} \text{ and } L \in C, \bar{L} \in D' \text{ then } (C - \{L\}, D' - \{\bar{L}\}, R \cup R' \cup \{L\}) \in \mathcal{S} \ (*)$$

where L and $\bar{L}$ are complementary literals, C, C', D, D', R, R' are sets of ground literals, and $\mathcal{S}$ is a set of triples of such sets. Then the following proposition gives the needed condition:

Property 6 (Closure)**.** *A split tableau (T^L, T^R) can be closed using Σ^{LR} if applying $(*)$ on $\{(C, \emptyset, \emptyset) \mid C \in \mathsf{ClosSet}(T^L)\} \cup \{(\emptyset, C, \emptyset) \mid C \in \mathsf{ClosSet}(T^R)\}$ yields $(\emptyset, \emptyset, R)$ for some R.*

The intuition behind this construction is that every step of $(*)$ corresponds to using a physical atom (or its negation) together with the associated formula in Σ^{LR} in the interpolant/plan for the query. In our running example, these are the two literals $S(s0)$ and $\neg G(s0, s2)$.

5 Query Planning and Optimization

Figure 3 associates two sets of literals with every range-restricted formula constructed from physical atoms (called henceforth a *plan fragment*).

Definition 7 (Admissible Fragments and Plans (variation on [9]))**.** *Let (T^L, T^R) be a split tableau for a query $Q(\mathbf{x})$. A fragment P is admissible if, for every $C \in L_P$, there is $C' \in \mathsf{ClosSet}(T^L)$ such that $C \subseteq C'$, and, for every $C \in L_R$, there is $C' \in \mathsf{ClosSet}(T^R)$ such that $C \subseteq C'$. We say that P is a* plan *for Q if (i) the above inclusions are set equalities, and (ii) no Skolem constants except those corresponding to the $\mathbf{x}$ variables appear in P.*

In this way, the sets L_P and R_P restrict which fragments are considered as parts of the query plans and thus define the search space for the planning process. It also defines the *initial fragments*: fragments of the form $R(\mathbf{a})$ and $\neg R(\mathbf{a})$ that satisfy the above condition. To guide the search, we assume that we are given a function cost that maps plan fragments to their (estimated) cost that is monotone (fragments are more costly than their parts). For a

```
queue: a priority queue of fragments with priorities determined by g(f) + h(f);
for all initial fragments f do insert(quantify(f),queue);
while (queue not empty && (f1=top(queue)) is not a plan) do
  if (there is a fragment f2 that has not been combined with f1)
    then for all f3 ∈ quantify(admissible-combination(f1,f2)) do
      if (not dominates(f4,f3) by any fragment f4)
        then insert(f3,queue); for all f4 dominates(f3,f4) do remove(f4,queue);
    else pop(queue);
```

Figure 4: Plan Search.

particular example of such a function see, e.g., [1]. We use this function to direct A* search for query plans. As usual in A*, the search is directed by the sum of the cost of the current state, g, in our case, the cost of the fragment under consideration, and a heuristic estimate of the cost of reaching the goal, f, in our case, a query plan. The heuristic f is based again on the closing sets for (T^L, T^R) for a given query, and is given as follows:

1. We define *reduced closing sets* to be the closing sets for (T^L, T^R) in which L_P (R_P) are removed from one of the closing sets for T^L (T^R, resp.).
2. We find the smallest value of $\sum_{L \in R} \mathsf{cost}(L)$ over all triples $(\emptyset, \emptyset, R)$ derived from the reduced closing sets by $(*)$.

Intuitively, we underestimate the cost of the remainder of the plan by simply adding the costs of the physical atoms that still need to appear in the plan, assuming each is used just once (rather than, e.g., in a *join* operator (that implements conjunctions). This yields an *admissible heuristics*[5] for A* search. The A* algorithm used is outlined in Figure 4 and uses the following auxiliary functions:

admissible-combination(f_1,f_2): Generate all admissible binary combinations of f_1 and f_2 using "$\wedge$" and "$\vee$"[6];

quantify(f): Replace all Skolem constants that occur in atoms of f but not in the rest of the closing sets oldest to youngest (the constant ids implicitly carry this information); constants introduced in T^L are replaced by an existential quantifier, those from T^R by a negation of the existential quantifier applied on $\neg f$ (in combination with the heuristic for "admissible-combination"); eager use of this function corresponds to the standard "pushing of quantifiers" database query optimization heuristic;

dominates(f_1,f_2): For fragments with $L_{f_1} = L_{f_2}$ and $R_{f_1} = R_{f_2}$ (i.e., that *achieve* the same effect), we return the Boolean value $g(f_1) < g(f_2)$, and false otherwise.

Soundness of the fragment construction reduces to the interpolant extraction procedure in [7].

Example 8. The fragments considered for our running example are $S(s0)$, $\neg G(s0, s2)$, $\neg\exists y.G(s0, y)$, and $S(s0) \wedge \neg\exists y.G(s0, y)$, which yields the ultimate plan for the query (still to be handed off to the actual code generator).

The code for the plan is then a C get-first/get-next iterator representing a nested loops join applied on S and a simple complement of G, as outlined in [11].

[5] The heuristic is *admissible* with respect to the cost model due to the monotonicity assumption. Note, however, that the cost model only approximates the actual execution cost, which can be lower than the estimate.

[6] For "$\vee$", we use a heuristic that applies De-Morgan rule to disjunctions in which one of the atoms is negated.

6 Beyond Basic Interpolation and Summary

The paper develops a practical path to implementing a query optimizer based on the theoretical foundations in [9]. There are additional issues addressed in the implementation, a full exposition of which is beyond the scope of this paper:

Dealing with Equality. To achieve an acceptable level of performance, a paramodulation-style inference for equalities via specialized instructions in our VM is used in combination with the ability to efficiently locate first occurrences of Skolem constants (due to our restrictions on the use of the MKSKOLEM instruction).

Range-restrictions and Binding Patterns. [4] shows that for range restricted schemas and queries, if an interpolant exists, it will be domain-independent. However, in practice physical atoms are often equipped with access restrictions (called binding patterns [11]) and additional care needs to be taken to satisfy these.

Duplicate semantics. Implementation of quantifiers and connectives (e.g., existential quantifiers by relational projection) often leads to undesirable *duplication of answers* (that need to be eliminated). We apply techniques [10, 11] (that require additional schema reasoning) to alleviate this issue in post-processing.

On related work, query reformulation under constraints has been studied extensively in the database literature. The two most prominent current approaches are based on *chase&backchase* [6] and Craig interpolation [2, 4, 9, 11]. Detailed comparison of the approaches is beyond the scope of this paper and can be found in [3, 11].

References

[1] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[2] Michael Benedikt, Balder ten Cate, and Efthymia Tsamoura. Generating low-cost plans from proofs. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 200–211, 2014.

[3] Michael Benedikt, Balder ten Cate, and Efthymia Tsamoura. Generating plans from proofs. *ACM Trans. Database Syst.*, 40(4):22:1–22:45, 2016.

[4] Alexander Borgida, Jos de Bruijn, Enrico Franconi, Inanç Seylan, Umberto Straccia, David Toman, and Grant E. Weddell. On finding query rewritings under expressive constraints. In *Proceedings of the Eighteenth Italian Symposium on Advanced Database Systems, SEBD*, pages 426–437, 2010.

[5] William Craig. Three uses of the Herbrand-Genzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 1957.

[6] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 459–470, 1999.

[7] Melvin Fitting. *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science. Springer Publishers, 1996.

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[9] 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 TABLEAUX*, pages 339–354, 2015.

[10] Vitaliy L. Khizder, David Toman, and Grant E. Weddell. Reasoning about duplicate elimination with description logic. In *Computational Logic - CL 2000*, pages 1017–1032, 2000.

[11] David Toman and Grant E. Weddell. *Fundamentals of Physical Design and Query Compilation*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.