Cost-Based Query Optimization via AI Planning

Nathan Robinson
College of Engineering
and Computer Science
Australian National University
Canberra, Australia*

Sheila A. McIlraith
Dept. of Computer Science
University of Toronto
Toronto, Canada

David Toman
D.R. Cheriton School of
Computer Science
University of Waterloo
Waterloo, Canada

Abstract

In this paper we revisit the problem of generating query plans using AI automated planning with a view to leveraging significant advances in state-of-the-art planning techniques. Our efforts focus on the specific problem of cost-based join-order optimization for conjunctive relational queries, a critical component of production-quality query optimizers. We characterize the general query-planning problem as a delete-free planning problem, and query plan optimization as a context-sensitive cost-optimal planning problem. We propose algorithms that generate high-quality query plans, guaranteeing optimality under certain conditions. Our approach is general, supporting the use of a broad suite of domain-independent and domain-specific optimization criteria. Experimental results demonstrate the effectiveness of AI planning techniques for query plan generation and optimization.

1 Introduction

Informally, a *query execution plan* (*query plan* for short) is an algebraic expression composed of physical operations over data structures representing finite relations that are used to access and combine information. *Query optimization* endeavors to find a query plan that minimizes space, latency, or other properties associated with the efficiency of query plan execution (Ioannidis 1996; Chaudhuri 1998; Haas et al. 2009).

In this paper we revisit how AI automated planning can be applied to the query optimization problem for relational queries. In particular, we focus on the so called *join-order selection* for conjunctive queries. The join-order selection part of query optimizers is responsible for choosing which data structures to use to access disk-resident data, for the selection of the join algorithms to combine the retrieved data, and for arranging these operators into a query plan in a way that minimizes the cost of executing such a plan (with respect to a given *cost model*). The importance of the join-order optimization can hardly be overstated: the heart of query optimizers in commercial relational systems is the *join-order optimizer* for conjunctive queries. These are sometimes based on proprietary extensions of dynamic programming techniques developed in (Selinger et al. 1979), and sometimes based on time-limited branch and bound search. Complex queries are typically preprocessed by heuristic rules to so called *query blocks*—(maximal) conjunctive subqueries—that are optimized for join-order separately to reduce the overall search space (Chaudhuri 1998).

There is a body of previous AI planning research related to query planning by Knoblock, Ambite, Kambhampati, and others (e.g., (Kambhampati and Gnanaprakasam 1999; Nie and Kambhampati 2001; Kambhampati et al. 2004; Knoblock 1996; Ambite and Knoblock 1997; 2000; Barish and Knoblock 2008; Friedman and Weld 1997)) with Karpas and colleagues' work being the most recent (Karpas et al. 2013). Most of these works use planning to *rewrite* relational query expressions using rewriting rules based on textbook relational expression equivalences, which do not capture the critical challenges within commercial practice.

There are two main differences that set our approach apart from other research on AI planning for query optimization:

1. We provide a *novel encoding* of the conjunctive query optimization problem as an AI planning problem. Unlike many other approaches whose states essentially encode the generated query plan (Ambite and Knoblock 1997; 2000; Barish and Knoblock 2008; Karpas et al. 2013) in our approach states encode *what the partial plan has logically achieved to answer the query*; the plan itself is encoded by the (optimal) sequence of actions that reach the particular state. This way, different but equivalent query plans lead to the same state (with a different cost) and consequently the planner can prune suboptimal partial plans.
2. We integrate a *relational cost model* into the planning process. Unlike standard planning problems (including other approaches to query planning (Karpas et al. 2013)) in which the cost of executing actions is constant and additive (such as the shortest path in a road network), cost of relational operations crucially depends on properties of their inputs (such as the size of the inputs) and thus the cost of actions depends on the state they are executed in.

We base our approach on a slight extension of the seminal work on query optimization in IBM's System R (Selinger et al. 1979). In particular, we use the same physical plan operators for access paths and joins and the associated cost model developed in that paper. Also, we adopt their restriction to linear (left-deep) query plans for conjunctive queries

*Most of this work was carried out at the University of Toronto.

to simplify the presentation. For pipelined execution of conjunctive queries left-deep plans are fully general (as they describe the fine-grained execution of all other plans). For non-pipelined executions, the above is not true, but in practice the optimality degradation is about 2% compared to the more complex tree-like query plans (Bruno, Galindo-Legaria, and Joshi 2010). Our own experiments generating bushy plans (omitted for space) confirm these findings.

Our approach derives from our observation that optimized information gathering is a cost-optimizing delete-free planning problem. We develop three somewhat diverse planning algorithms to address our query optimization problem: a greedy delete-free algorithm, an optimal A* search, and a greedy best-first search, together with a suite of domain-specific heuristics. We analyze their properties and assess their computational effectiveness. Of particular note is our ability to generate query plans that are guaranteed optimal on problems that are highly competitive to those reputed to be solved to optimality by commercial systems.

Despite its maturity, query optimization remains a topic of research with recent advances targeting heuristics for query optimization (e.g., (Bruno, Galindo-Legaria, and Joshi 2010)). The work presented here similarly provides a unique and promising perspective on the development of search techniques for query optimization.

2 Preliminaries

We begin with a review of necessary relational database background and terminology. Conjunctive queries (CQ) in SQL take the form

$$\texttt{select } x_1,\ldots,x_k \texttt{ from } R_1\, r_1,\ldots,R_n\, r_n \texttt{ where } C$$

where C is a conjunction of equalities of the form $r_i.a = x_l$ with a an attribute (column) of the relation (table) R_i. It can be written as a *predicate calculus* comprehension:

$$\{x_1\ldots,x_k \mid \exists r_1,\ldots,r_k,x_{k+1},\ldots,x_m. \\ R_1(r_1)\wedge\ldots\wedge R_n(r_n)\wedge \bigwedge R_i a_j(r_i,x_l)\}$$

where, conceptually, $R_i(r_i)$ atoms bind the variables r_i to record id's of records in the instance of R_i and $R_i a_j$ are binary relations that represent attributes of R_i (attribute relations). Note that the *tuple variables* (r_i) are separate from the *value variables* (x_j). We allow some of the variables x_i in the `select` list to be designated as *parameters*.

This representation of queries (and other artifacts such as access path below) is essential to our encoding as it allows us not only to specify the goal of our planning problem, but also to capture partial progress towards that goal by utilizing the above atomic formulas as fluents. This way, a query plan can be incrementally constructed by simply recording the *effects* of actions in terms of these fluents (rather than as explicitly encoding the particular query plan).

2.1 Operators for CQ Query Plans

The *query plans* for conjunctive queries are responsible for accessing data relevant to the query answers that are stored in (possibly disk-based) data structures, called the *access paths*. The results of these primitive operations are then combined using *join* operators to form the ultimate query plan. Indeed, the crux of query optimization for conjunctive queries lies in the choice of appropriate access paths for the relations involved in the query and in the *ordering* of how the results of these operations are combined using joins—hence this part of query optimization is often dubbed *join-order selection*. Additional relational operators, such as selections and projections are commonly *not* considered at this time—either they are fully subsumed by joins (e.g., selections) or can be added in post-processing (projections[1]).

Access Paths. The primitive relational operations are the *access paths* (APs), operators responsible for retrieving the *raw* data from relational storage (e.g., disks). Every user relation (table) is typically associated with several access paths that support efficient search for tuples based on various conditions, e.g., find all Employee tuples in which the name attribute is "John". Note that the access paths used for search *expect* some of their attributes (the *inputs*) to be bound (i.e., associated with a constant value obtained earlier in the course of execution of the query plan). The access paths for a relation R are described by triples of the form

$$\texttt{name}(r,x_1,\ldots,x_k) : \langle R(r)\wedge C, \{x_{i_1},\ldots,x_{i_k}\}\rangle$$

where C is a conjunction of equalities (similar to those in conjunctive queries) only using attributes of R and variables r and $x_1,\ldots,x_k$ out of which $x_{i_1},\ldots,x_{i_k}$ denote the *input parameters* of this access method.

Base File (scan and record fetch): In the basic setting, for each relation R we always have two access paths:

$$R\texttt{Scan}(r,x_1,\ldots,x_k) : \langle R(r)\wedge Ra_1(r,x_1)\wedge\ldots\wedge Ra_k(r,x_k),\{\}\rangle$$
$$R\texttt{Fetch}(r,x_1,\ldots,x_k) : \langle R(r)\wedge Ra_1(r,x_1)\wedge\ldots\wedge Ra_k(r,x_k),\{r\}\rangle$$

where $a_1,\ldots,a_k$ are all the attributes of R; these two paths are used to retrieve all tuples of a relation R and to retrieve a *particular* tuple given its tuple id (note that the tuple id r is the *input* to the access path and has to be bound before a record can be fetched).

Indices: In addition to the basic access paths we typically have additional access paths, called *indices*, that are used to speed up lookups for tuples based on certain search conditions (that are again captured by specifying inputs for the access path). Note also that the indices typically store only a few attributes of the indexed relation (the remaining ones can be retrieved using the `Fetch` access path). We capture this by declaring an access path

$$R\texttt{xxxIndex}(r,Y) : \langle R(r)\wedge C, X\rangle$$

for each *index* on R (called generically `xxx` here) where C is a conjunction of attribute relations (for attributes of R), is X a set of names of variables that correspond to parameters of the index (the search condition), and Y is a set of variables that correspond to the attributes actually stored in the index (typically $X = Y$).

[1]While we do not explicitly deal with duplicates in this presentation, all the techniques are fully compatible with SQL's duplicate semantics for conjunctive queries.

Example 1 Given a relation `Emp(Id, Name, Boss)` we will have the following access paths:

$\texttt{EmpScan}(r, x_1, x_2, x_3) : \langle \texttt{Emp}(r) \wedge$
$\quad \texttt{EmpId}(r, x_1) \wedge \texttt{EmpName}(r, x_2) \wedge \texttt{EmpBoss}(r, x_3), \{\}\rangle$
$\texttt{EmpFetch}(r, x_1, x_2, x_3) : \langle \texttt{Emp}(r) \wedge$
$\quad \texttt{EmpId}(r, x_1) \wedge \texttt{EmpName}(r, x_2) \wedge \texttt{EmpBoss}(r, x_3), \{r\}\rangle$
$\texttt{EmpNameIndex}(r, x_1, x_2) : \langle \texttt{Emp}(r) \wedge$
$\quad \texttt{EmpId}(r, x_1) \wedge \texttt{EmpName}(r, x_2), \{x_2\}\rangle$
$\texttt{EmpIdIndex}(r, x_1) : \langle \texttt{Emp}(r) \wedge \texttt{EmpId}(r, x_1), \{x_1\}\rangle$

that allow retrieving all employee records, finding a record by record id, finding record ids using employee id, and finding record ids using employee name, respectively. Note that `EmpNameIndex` has an extra variable x_1 for `Id`; we will see later how this can be used for so-called *index only* query plans. Note also how the right-hand sides, the *logical descriptions* of the access paths correspond to atoms in our conjunctive query representation.

Join Operators. To combine the results of access path invocations into query results, *join* operators ($\bowtie$) that essentially implement *conjunctions* are used. We consider the following two implementations of these operators:

Nested Loops Join (NLJ): The most basic join operator is based on the idea that for each tuple retrieved from its left argument it probes its right argument to find matching tuples. When the right argument is an access path with an input parameter present in the above tuple, the value is passed to the access path to facilitate search (in this case the join is often called the *IndexJoin*).

Merge Sort Join (MSJ): Another approach to implementing the join operator is to *sort* each of its arguments on the join attribute and then merge the results. While algorithmically preferable, the overhead of sorting often makes this method inferior to the plain NLJ. On the other hand, knowledge of *order properties* of the underlying access paths may allow the sorting step to be avoided.

Many other join implementations and algorithms have been investigated, such as the *Hash join* (based on creating a temporary hash-based index); these are handled similarly to the above two (and are omitted here).

Example 2 For the query:

$$\texttt{select } x_1, x_2 \texttt{ from Emp } e$$
$$\texttt{where } e.\text{Id} = x_1 \texttt{ and } e.\text{Name} = x_2$$

we expect the following query plans, based on whether x_1 or x_2 (or neither) is a supplied query parameter:

- None: $\texttt{EmpScan}(r, x_1, x_2, x_3)$
- x_1: $\texttt{EmpIdIndex}(r, x_1) \bowtie_{\text{NLJ}} \texttt{EmpFetch}(r, x_1, x_2, x_3)$
- x_2: $\texttt{EmpNameIndex}(r, x_1, x_2)$

Note that the last plan is an *index-only* plan. Also note that replacing *EmpFetch* access path that retrieves employee records based on record id by three paths, one for each employee attribute, would simulate how *column stores* execute queries. This relies on our representation of queries and access paths using tuple ids and attribute relations; indeed, this representation supports many advanced features that go far beyond textbook approaches and often generalizes hard-coded solutions present in production relational systems, such as distinguishing between clustered/unclustered indices, use of index-intersection plans, etc.

2.2 Cost Model

The optimality of a query plan is judged with respect to a *cost model* based on summary (statistical) information about the relations and the access paths. We collect:

- for every relation R (table): the number of tuples and, for each attribute, the number of distinct values;
- for each access path (index): the cost (disk pages read) of retrieving all the tuples that match the access path's input parameters (reading the whole data set if none);

These estimates are then combined, using arithmetic formulas associated with particular join algorithms, to estimate the cost and cardinality of partial query plans (in disk page reads and numbers of tuples). Accurate estimates of query costs are essential and are the key to the planners ability to find query plans that are optimal *in reality*, i.e., when query plans are executed. Our model is based on a System-R style cost model (Selinger et al. 1979) to illustrate the approach; more advanced cost models can be easily used as well.

3 Mapping into PDDL Actions

We map join-order selection to an automated planning problem by combining the choice of the next access path with the appropriate implementation of the join operation in a single PDDL action (note that this is sufficient for our left-deep plans). We use the fluents nds-R, has-R, and bound to capture the fact that the query needs to access a certain relation, that the current query plan has already accessed a certain relation, and that a variable has been bound to a value in the current plan, respectively.

3.1 Nested Loop Joins

First considering plans that use NLJ only (note that this also covers index-join based plans in the cases where NLJ is coupled with an index access path). For each AP

$$\langle R\text{AP}, R(r) \wedge Ra_1(r, x_1) \wedge \ldots \wedge Ra_k(r, x_k), \{x_{i_1}, \ldots, x_{i_l}\}\rangle$$

there is an action:

ACTION NLJ-RAP

pre: $\texttt{nds-}R(?r), \texttt{nds-}Ra_1(?r, ?x_1), \ldots, \texttt{nds-}Ra_k(?r, ?x_k),$
$\quad \texttt{bound}(?x_{i_1}), \ldots, \texttt{bound}(?x_{i_l})$
post: $\texttt{has-}R(?r), \texttt{has-}Ra_1(?r, ?x_1), \ldots, \texttt{has-}Ra_k(?r, ?x_k),$
$\quad \texttt{bound}(?x_1), \ldots, \texttt{bound}(?x_k)$

Note that the pre- and postconditions of these actions ensure that the precondition (as a conjunction) logically implies the actual access path (as a predicate) that, in turn, implies the postcondition. This condition must be satisfied by all actions corresponding to access paths. Next, for the query:

$$\{x_1 \ldots, x_k \mid \exists r_1, \ldots, r_k, x_{k+1}, \ldots, x_m. \\ R_1(r_1) \wedge \ldots \wedge R_n(r_n) \wedge \bigwedge Ra_i(r_i, x_l)\}$$

we have an initial state s_0 such that:

$\texttt{nds-}R_1(r_1), \ \ldots, \texttt{nds-}R_n(r_n) \in s_0$
$\texttt{nds-}Ra_i(r_i, x_l) \in s_0$ for all conjuncts in $\bigwedge Ra_i(r_i, x_l)$, and
$\texttt{bound}(x_j) \in s_0$ for all parameters of the query;

and a goal $\mathcal{G}$ such that:

$\texttt{has-}R_1(r_1),\ \ldots,\ \texttt{has-}R_n(r_n) \in \mathcal{G}$ and
$\texttt{has-}Ra_i(r_i, x_l) \in \mathcal{G}$ for all conjuncts in $\bigwedge Ra_i(r_i, x_l)$.

Observe that the initial state's set of nds- atoms is identical to the given query, that each action invocation only adds has- atoms whose conjunction is implied by the nds- atoms (cf. the invariant above required for all actions), and, when the goal is reached, the has- atoms contain all atoms of the original query. This yields the required equivalence.

Example 3 For the query in Example 2, assuming x_1 is a parameter, we have a possible plan:

$\langle \texttt{NLJ-EmpIdIndex}(r, x_1), \texttt{NLJ-EmpFetch}(r, x_1, x_2, x_3)\rangle$

Note that the initial NLJ "joins" with a single tuple of parameters, in this example with the value for x_1. This then corresponds to exploring the following sequence of states:

1. nds-Emp(r), nds-EmpId(r, x_1), nds-EmpName(r, x_2), bound(x_1)
2. nds-Emp(r), nds-EmpId(r, x_1), nds-EmpName(r, x_2), bound(x_1), has-Emp(r), has-EmpId(r, x_1), bound(r)
3. nds-Emp(r), nds-EmpId(r, x_1), nds-EmpName(r, x_2), bound(x_1), has-Emp(r), has-EmpId(r, x_1), bound(r) has-EmpName(r, x_2), has-EmpBoss(r, x_3), bound(x_2), bound(x_3)

Another plan is $\langle \texttt{EmpScan}(r, x_1, x_2, x_3)\rangle$. This produces the following sequence of states:

1. nds-Emp(r), nds-EmpId(r, x_1), nds-EmpName(r, x_2), bound(x_1)
2. nds-Emp(r), nds-EmpId(r, x_1), nds-EmpName(r, x_2), bound(x_1), has-Emp(r), has-EmpId(r, x_1), has-EmpName(r, x_2), has-EmpBoss(r, x_3), bound(x_2), bound(x_3)

This plan is however, less efficient given our cost model.

3.2 Adding Merge Sort Joins

While we could naively add MSJ to the above approach, we would miss opportunities arising from additional understanding of ordered properties of the access paths in order to avoid unnecessary sorting steps in the plan. We use the fluent asc(x) to indicate that the values of the variable x are sorted (ascending) in the output of the (current) query plan (again we use only single-variable orderings, but extending to other interesting orders is a mechanical exercise). Note that unlike, e.g., the bound fluent, the sorted property of a variable may disappear after executing the next join, causing the encoding to lose its delete-free character.

To take advantage of order of access paths and results of partial query plans we use the following three actions that correspond to sorting the result of the current query plan, to merge-joining with an appropriately ordered access path, and to merge-joining with an access path that was sorted prior to the join, respectively:

ACTION Sort-on-?x: (sort current result on x)

pre:	bound(?x)
post:	asc(?x), ¬asc(?y) for all other variables ?y

ACTION MJ-on-?x-AP: add a *merge-join* on variable x with access path AP, assuming AP is also sorted on x.

pre:	bound(?x), asc(?x)
post:	effects of AP as for NLJ, ¬asc(?y) for non-?x variables

ACTION MSJ-on-?x-AP: add a *sort-merge-join* on variable x with access path AP, when AP is not sorted on x.

pre:	bound(?x), asc(?x)
post:	effects of AP as for NLJ, ¬asc(?y) for non-?x variables

We also add asc(x) to the initial state for each bound variable x. (This is sound since there is only a single "tuple" of parameters and constants.)

Finally, for a given query problem, we take an initial description of the problem instance, together with the schemas described here, and generate a query-specific PDDL planning instance. In addition to the information in the schemas, each individual action has an action cost that is a computation that relies on the variable bindings in the current state and as such is *context specific*. While PDDL supports context-specific action costs, few planners actually accommodate them, we therefore solve the above described problems using domain-specific solvers presented in Section 4.

In particular, the cost of our actions (that represent joining the next access path to the current query plan) depends on the number of tuples so far, captured as $size(s)$ for the current state s, the size and structure of the relation to be joined, and the particular join algorithm. These values are used to estimate the cost and size in successor states.

4 Generating Query Plans

In the previous section, we saw how to encode the join-order query optimization problem in terms of a PDDL initial state, goal, and a set of PDDL action schemas that are translated, together with their cost model, into a set of instance-specific ground actions. We refer to the problem of generating a query plan with the NLJ PDDL encoding as a J-O *query-planning problem* and when augmented with MSJ APs as a J-O+ *query-planning problem*. By inspection:

Proposition 1 J-O *query planning is a delete-free planning problem.*

Proposition 2 J-O *query optimization is a context-sensitive cost-optimizing delete-free planning problem.*

Proposition 3 J-O+ *query planning is not a delete-free planning problem and as such,* J-O+ *query optimization is simply a context-sensitive cost-optimizing planning problem.*

There have been numerous studies on the complexity and parameterized complexity of planning including (Bylander 1994; Chen and Giménez 2010; Bäckström et al. 2012).

Theorem 1 *Finding a* J-O *query plan can be realized in polynomial time but* J-O *and* J-O+ *query optimization are NP-hard.*

The proof follows from the fact that monotonic planning is polynomial-time while delete-free optimal planning is NP-Hard (Bylander 1994). This is consistent with results for query optimization, which is recognized to be NP-hard in all but very restricted cases (Ibaraki and Kameda 1984).

Building on this correspondence to delete-free planning and in light of recent advances in cost-optimizing delete-free planning (e.g., (Gefen and Brafman 2012; Haslum, Slaney, and Thiébaux 2012; Pommerening and Helmert 2012)) we propose three algorithms together with a suite of domain-dependent heuristics for generating optimized query plans. The first algorithm, DF, exploits the delete-free nature of our problem, greedily generating cost-minimizing delete-free plans. The second is a classical A* algorithm, which we ran with three different admissible heuristics. The third, GR, is a greedy best-first search that does not consider partial plan cost in its evaluation function, but that uses an admissible heuristic, together with cost, in order to do sound pruning. The latter two algorithms can be guaranteed to produce optimal plans under certain conditions, which is notable relative to the state of the art in query planning.

4.1 Fast Delete-Free Plan Exploration

Algorithm DF computes plans that do not include sorting actions. For certain problems, this precludes DF from finding optimal solutions, but the costs of the plans it finds are guaranteed to be upper bounds on the optimal cost allowing them to be used as initial bounds for A* and GR.

The decision not to allow sorting actions means that in any state s, a subset of all actions can be efficiently determined that will move the planner towards the goal. We call such actions *useful* and denote the set of useful and applicable actions in state s as $A_u(s)$. In more detail, an action is useful in a state s if it adds a `has` or `bound` fluent which is not in s. The algorithm proceeds by heuristically generating sequences of useful actions which achieve a goal state. Throughout its fixed runtime, it remembers the best such plan generated. Clearly DF does not guarantee to compute

Algorithm 1 DF

```
$\pi_* \leftarrow \langle\rangle$; $c_* \leftarrow \inf$
**while** not time out **do**
    $s \leftarrow s_0$; $c \leftarrow 0$; $\pi \leftarrow \langle\rangle$
    **while** $\mathcal{G} \not\subseteq s$ **do**
        **if** random fraction of $1 \leq 0.9$ **then**
            $a \leftarrow a' \in A_u(s)$ with minimal resulting size,
                breaking ties with action cost
        **else**
            $a \leftarrow a' \in A_u(s)$ randomly selected with a
                likelihood inversely proportional to the
                resulting size and then action cost
        $c \leftarrow c + cost(a)$; $\pi \leftarrow \pi + a$
        **if** $c \geq c_*$ **then** break
        $s \leftarrow s \cup add(a)$
    **if** $c < c_*$ **then** $c_* \leftarrow c$; $\pi_* \leftarrow \pi$
**return** $\pi_*$ and $c_*$
```

an optimal query plan, but it has the capacity to generate a multitude of plans very quickly. In practice, our Python implementation of DF can generate hundreds to thousands of candidate plans per second.

4.2 A*

We use an eager A* with a variety of domain-dependent admissible heuristics. The code is based on the eager search algorithm in Fast Downward (Helmert 2006). The primary difference from existing heuristic-search planning algorithms is that, as our action costs are context-dependent, we compute them lazily when expanding a state. While this increases the cost of expanding each state, it eliminates the need to pre-compute actions with all possible costs, which is prohibitively expensive.

To reduce memory requirements, an bound can be used with the algorithm to prevent generated nodes with f values exceeding the bound from being added to the open list.

We now examine the three heuristics that were used with this algorithm and show their admissibility and consistency and therefore the optimality of the solutions returned by A*. The first heuristic, h_{blind} evaluates a state s as follows:

- $h(s) = 0$, if $\mathcal{G} \subseteq s$; and
- $h(s) = 1$, otherwise.

Proposition 4 *The heuristic h_{blind} is admissible and consistent for the query-planning problems we consider.*

The next heuristic, h_{admiss} evaluates a state s by counting the number of unsatisfied relations and assuming that $size(s)$ and all subsequent states is 1 to get a lower bound on the cost of achieving the goal.

In a given state s let $\mathcal{R}(s)$ be the unsatisfied relations and $\mathcal{R}_I(s)$ be the (partially) unsatisfied relations for which we have a bound tuple id – i.e. those relations for which a partial index action has been executed, which can be satisfied with a fetch action.

h_{admiss} evaluates a state s as follows:

- $h(s) = 0$, if $\mathcal{G} \subseteq s$, and
- $h(s) = |\mathcal{R}_I(s)| + \sum_{r \in \mathcal{R}(s) \setminus \mathcal{R}_I(s)} max(1, ceil(log_{200}(pages(R))))$

Proposition 5 *The heuristic h_{admiss} is admissible and consistent for for the query-planning problems we consider.*

To prove Proposition 5, note that the minimum cost of satisfying any relation R is 1 when we can execute a fetch action on R (i.e., $R \in \mathcal{R}_I(s)$) and otherwise the cost is at least $max(1, ceil(log_{200}(pages(R))))$. h_{admiss} is monotone because whenever we execute an action a, resulting in state s', $|\mathcal{R}_I(s')| \leq |\mathcal{R}_I(s)|$ and $|\mathcal{R}(s')| \leq |\mathcal{R}(s)|$.

The final heuristic that we used, $h_{admissLA}$ builds upon the h_{admiss} heuristic by performing one step of look ahead to take into account the size of the current state s. Let $A_u(s)$ be the set of applicable and useful actions in state s, including sort actions. $h_{admissLA}$ evaluates a state s as follows:

- $h(s) = 0$, if $\mathcal{G} \subseteq s$, and
- $h(s) = min_{a \in A_u(s)} c(a) + h_{admiss}(s \cup add(a))$

Proposition 6 *The heuristic $h_{admissLA}$ is admissible and consistent for the query-planning problems we consider.*

To see that Proposition 6 holds, see that $h_{admissLA}$ has a value for non-goal states that is the minimum over all successors of the cost to reach that successor s' and the admissible and monotone estimate given by h_{admiss} from s'.

Theorem 2 *If* A* *terminates on a query-planning problem of the type we consider, then it returns an optimal solution.*

We observe that our dynamic cost can be seen just as a proxy for a large number of ground actions with fixed-costs, hence the theorem follows as the heuristics are consistent.

4.3 Greedy Best-First Search

This algorithm GR is an eager greedy best-first search that uses the same heuristics and code-base as A*. The main difference between GR and A* is that GR orders the states on its open list solely on the basis of their heuristic values, that is by the function $f(s) = h(s)$. Expanded states s are pruned when $g(s) + h(s)$ exceeds the current bound. Since this algorithm is greedy, the first solution it finds may not be optimal. However, in this case, as the search progresses better solutions will be found and the current bound tightened.

Theorem 3 *Pruning an expanded state s when $g(s) + h(s)$ exceeds the current bound does not rule out optimal solutions.*

The proof is based on observing that the bounds are sound and heuristics admissible. Therefore every state s in the sequence of states visited by an optimal plan $s_0, ..., s_k$, will have $g(s) + h(s)$ less than or equal to the current bound and not be pruned. The next theorem follows as Theorem 3 implies that if the search space is exhausted an optimal solution will have been found (or none exists).

Theorem 4 *If* GR *terminates on a query-planning problem of the type we consider, then it returns an optimal solution.*

5 Evaluation

Our experiments attempt to address the quality of plans generated and the performance of the proposed algorithms. Unfortunately, there are no benchmarks that measure the performance of query *optimizers* and comparing runtimes of the plans generated on database benchmarks does not contribute to addressing this issue as our plans are optimal or nearly optimal. Moreover, algorithms used in commercial query planners and the associated cost models are proprietary, nor is there an interface allowing execution of alternative plans, making a meaningful comparison impossible.

With this information in hand, the purpose of our experiments was 1) to evaluate the relative effectiveness of the different approaches to query plan search and plan optimization that we examined , and 2) to get some sense of whether AI automated planning techniques held some promise for cost-based join order optimization, in particular, relational database query optimization more generally, and beyond that to the general problem of optimizing the quality of information gathering from disparate sources.

We evaluated 3 different algorithms: DF, our delete-free planning algorithm, A*, our A* search algorithm, and GR, our greedy search algorithm. The latter two algorithms were each evaluated with three different heuristics: h_{blind}, h_{admiss}, and $h_{admissLA}$. Our specific purpose was twofold. First, we aimed to determine how many problems each of A* and GR could solve optimally. Second, and more pragmatically, we aimed to determine the quality of the plans found by DF and A* as a function of time.

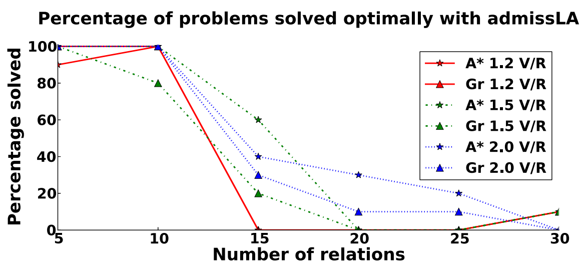

Figure 1: Percentage of problems solved optimally by A* and GR with $h_{admissLA}$ (2GB, 30 min time out).

We tested our planning systems on randomly generated database schemata and queries. Each generated schema consists of tables with between 2 and 10 attributes. Each table has a random size of between 10k and 500k tuples and 200 tuples are assumed to fit into a page of memory. The first attribute of every table is assumed to be the primary key and has a number of distinct values equal to the table size. Every other attribute has a random number of distinct values up to 10% of the table size.

Every query has a given number of relations $R = 5, 10, ..., 60$ and a given number of variables $V = 1.2, 1.5$, or 2 times R. Every query has 3 variables set as constants and 10 other variables selected (less if there is not enough variables). For each relation in the query there is a 10% chance of reusing an existing table, otherwise a new table is used. Variables are randomly assigned to relations and we ensure that queries are connected. Ten problem instances were generated for each R and V. All experiments were run on a 2.6GHz Six-Core AMD Opteron(tm) Processor with 2GB of memory per experiment.

In addition to the experiments that follow, we ran experiments with typical shaped queries such as star queries, chain queries, etc. and saw consistent results to those reported below. We also ran a suite of experiments with simplified cost models where we quantized the costs into fixed-cost "buckets," producing a standard PDDL that we could run with an off-the-shelf planner. Unfortunately, the error propagation during query cost estimation was so poor, or alternatively the number of buckets we had to create to maintain accuracy so large, that such an approach proved infeasible. Finally, we developed methods for generating so-called bushy plans. Details of these algorithms are beyond the scope of this paper, but results showed little optimality degradation.

5.1 Experiment 1: Optimal Plans

An upper bound B on plan cost was produced by running DF for 5 seconds and then A* and GR were run with the initial bound B with a time limit of 30 minutes. Running DF for longer than 5 seconds, to get tighter initial bounds, did not allow more problems to be solved optimally.

Summarizing the results of this experiment, h_{blind} was ineffective in finding optimal solutions beyond 5 relations. h_{admiss} was somewhat effective up to 10 relations with both A*, and GR, solving up to 50% of instances optimally.

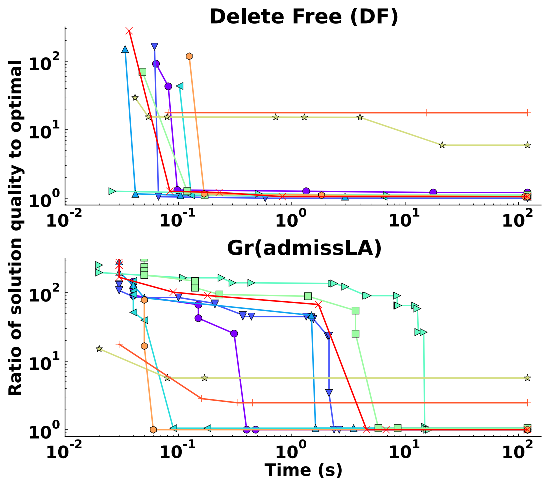

Figure 2: Ratio of the best plan cost found by a given time to the known optimal cost for problem instances with $R = 10$ and $V = 20$ for DF and GR. GR used the $h_{admissLA}$ heuristic and all approaches had 2GB, 2 min time out.

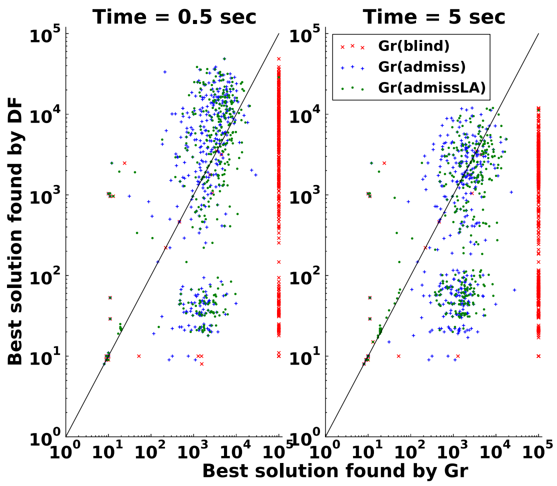

Figure 3: The costs of the best plans found by GR and DF after 0.5 and 5 seconds for all problem instances in our experiment set. Problems that could not be solved by an algorithm were assigned a cost of 10^5 in that case.

$h_{admissLA}$ was more successful, optimally solving most instances with 10 relations and some larger instances. Problems with a higher V/R ratio are somewhat easier than those with a low ratio. The performance of A*, and GR with this heuristic is summarized in Figure 1 (in color). Notably, our planners generate *optimal* query plans of a size competitive with those reputed to be solved by commercial systems.

While our heuristics h_{admiss} and $h_{admissLA}$ effectively direct the planner towards a goal state, they ignore the size of states between the current state and the goal. Unfortunately, our experiments showed that a full k-step lookahead heuristic was too costly to be effective.

5.2 Experiment 2: Fast High-Quality Plans

In many settings, a query plan must be found in seconds. Given that our optimal algorithms do not scale acceptably, we also experimented with several sub-optimal, any time algorithms. In these experiments we ran DF and GR with no initial bounds. Each algorithms was run for a total time of 2 minutes and plans were recorded as they were produced.

An important question about these algorithms is how the quality of the plans found compares to the optimal. We only have optimal solutions for the smaller problems instances that could be optimally solved by DF and GR. Figure 2 (in color) shows a representative sample of these results using the $h_{admissLA}$ heuristic. Encouragingly, it shows that for most problems, the sub-optimal algorithms closely approach the optimal quality within a second. On the smaller problems, for which we could generate optimal solutions, DF more quickly approached high-quality solutions.

As well as comparing DF and GR to the optimal solutions, we performed extensive experiments to compare them to each other. Figure 3 (in color) shows the costs of the best plans found by DF and GR with all three heuristics after 0.5 and 5 seconds for all instances in our experiment set.

When the runtime is short, GR generally finds plans that are better than those found by DF, often by several orders of magnitude. As the experiment time increases, the quality of best plans found by DF improve relative to those found by GR. As expected, GR with the h_{blind} heuristic fails almost all problems, usually running out of memory before any solutions are found. Over all run times there is a significant group of problems for which GR fails to compete with DF.

This pattern of performance is not surprising. GR initially performs better than DF because the $h_{admissLA}$ heuristic is more informative than the action selection heuristic employed by DF. However, the greedy nature of GR means that expansions made early in the search can commit the algorithm to low quality parts of the search space.

6 Summary

We proposed a novel encoding that supported the characterization of join-order query planning as a (near) delete-free planning problem. A challenge of join-order optimization is that the cost models are context sensitive: the cost of an operation crucially depends on properties of its (often intermediate) inputs (E.g., the cost of a join is a function of the *size* its inputs). Sacrificing context sensitivity results in inaccuracy that confounds the optimization. We developed delete-free, A*, and best-first search planning algorithms and domain-dependent heuristics for generating optimized query plans. Our experimental results were promising. Our planners could generate *optimal* query plans of a size that is highly competitive with those reputed to be solved by commercial systems. Further, our sub-optimal algorithms found near-optimal plans within few seconds. The work presented here is an interesting application for AI planning and one that pushes the boundary of (largely absent) techniques for context-sensitive cost-based planning.

Acknowledgements: The authors gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) and from the NSERC Business Intelligence Network program (BIN).

References

Ambite, J. L., and Knoblock, C. A. 1997. Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 706–713.

Ambite, J. L., and Knoblock, C. A. 2000. Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence Journal* 118(1-2):115–161.

Bäckström, C.; Chen, Y.; Jonsson, P.; Ordyniak, S.; and Szeider, S. 2012. The complexity of planning revisited - a parameterized analysis. In *Proc. of the 27th AAAI Conference on Artificial Intelligence (AAAI)*.

Barish, G., and Knoblock, C. A. 2008. Speculative plan execution for information gathering. *Artificial Intelligence Journal* 172(4-5):413–453.

Bruno, N.; Galindo-Legaria, C. A.; and Joshi, M. 2010. Polynomial heuristics for query optimization. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, 589–600.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.

Chaudhuri, S. 1998. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 34–43.

Chen, H., and Giménez, O. 2010. Causal graphs and structurally restricted planning. *J. Comput. Syst. Sci.* 76(7):579–592.

Friedman, M., and Weld, D. S. 1997. Efficiently executing information-gathering plans. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 785–791.

Gefen, A., and Brafman, R. I. 2012. Pruning methods for optimal delete-free planning. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 56–64.

Haas, P. J.; Ilyas, I. F.; Lohman, G. M.; and Markl, V. 2009. Discovering and exploiting statistical properties for query optimization in relational databases: A survey. *Statistical Analysis and Data Mining* 1(4):223–250.

Haslum, P.; Slaney, J. K.; and Thiébaux, S. 2012. Minimal landmarks for optimal delete-free planning. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 353–357.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Ibaraki, T., and Kameda, T. 1984. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.* 9(3):482–502.

Ioannidis, Y. E. 1996. Query optimization. *ACM Computing Surveys* 28(1):121–123.

Kambhampati, S., and Gnanaprakasam, S. 1999. Optimizing source-call ordering in information gathering plans. In *Proceedings of the 16th International Joint Conference on Artificial Workshop on Intelligent Information Integration*.

Kambhampati, S.; Lambrecht, E.; Nambiar, U.; Nie, Z.; and Gnanaprakasam, S. 2004. Optimizing recursive information gathering plans in EMERAC. *Journal of Intelligent Information Systems* 22(2):119–153.

Karpas, E.; Sagi, T.; Domshlak, C.; Gal, A.; Mendelson, A.; and Tennenholtz, M. 2013. Data-parallel computing meets strips. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2012, Bellevue, Washington, USA*.

Knoblock, C. A. 1996. Building a planner for information gathering: A report from the trenches. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, 134–141.

Nie, Z., and Kambhampati, S. 2001. Joint optimization of cost and coverage of query plans in data integration. In *Tenth International Conference on Information and Knowledge Management*, 223–230.

Pommerening, F., and Helmert, M. 2012. Optimal planning for delete-free tasks with incremental lm-cut. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 363–367.

Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; and Price, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 23–34.