Query Optimization Revisited: An AI Planning Perspective

Nathan Robinson
Dept. of Computer Science
University of Toronto
Toronto, Canada

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

David Toman
School of Computer Science
University of Waterloo
Waterloo, Canada

Abstract

The generation of high quality query plans is at the heart of query processing in traditional database management systems as well as in heterogeneous distributed data sources on corporate intranets and in the cloud. A diversity of techniques are employed for query plan generation and optimization, many of them proprietary. In this paper we revisit the problem of generating a query plan using AI automated planning. Characterizing query planning as AI planning enables us to leverage state-of-the-art planning techniques, as well as supporting the longer-term endeavor of purposeful information gathering as part of a larger data-intensive, task-driven system. While our longterm view is broad, here our efforts focus on the specific problem of cost-based join-order optimization, a central 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 plan* or a *query execution plan* is an ordered set of physical operations used to access information. *Query optimization* endeavors to find a query plan that maximizes the efficiency of execution, where efficiency may be measured in terms of minimizing space, latency, or other properties associated with the execution of the plan (e.g., (Ioannidis 1996; Chaudhuri 1998; Haas et al. 2009)). Traditionally the information being accessed by a query plan has resided in a relational database management systems, but as information management has evolved, query optimization has broadened to address plans that are executed over network accessible federated databases. Most recently, with the preponderance of structured and unstructured data in distributed information sources, there has been increasing interest in querying information sources that exist over the web and in the cloud, and extending beyond relational databases to linked data (e.g., (Ladwig and Tran 2010)).

In this paper we examine whether AI automated planning has anything of substance to contribute to the generation and optimization of plans for information gathering in general, and specifically for relational queries. Indeed there is a body of previous AI planning research related to query planning, including (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)). Many of these works use planning or plan rewriting to construct query plans using simple physical operations, some relying on extensive processing outside the planner. Much of this (excellent) work is older work, little of it benefiting from advances in the state of the art in planning and plan optimization in the last decade. We were originally interested in revisiting this problem with the broadened perspective of advances in delete-free, cost-optimizing, and preference-based planning, and with a view to the integration of optimized information gathering into decision-making.

We are motivated by the task of the generating optimized information gathering plans in all of their guises, but for the purposes of this paper, our algorithms and empirical evaluation are tailored to the task of query optimization in relational database systems, and specifically to cost-based join-order optimization – the optimization of the ordering of join operations employed in conjunctive query evaluation.

The original 1979-published work on query optimization was with respect to System R, and used dynamic programming techniques (Selinger et al. 1979). Modern-day systems are proprietary and embedded within commercial systems, but reportedly some continue to use dynamic programming, while others use time-limited branch and bound search. Here we cast the general problem of information gathering as a delete-free planning problem, and the problem of optimizing the quality of information gathering as a cost-optimizing delete-free planning problem.

Unfortunately, the delete-free property of information gathering is not universally applicable. In particular, when we delve into the details of the particular relational database query optimization task of cost-based join-order optimization, we immediately observe two things. First, that the cost models that are employed in join-order optimization are context sensitive. The cost of an action is predicated on what has preceded it. Further, we observe that while information gathering is delete free, some of the physical operations employed to realize efficient query plans can have a component that deletes a property of our plan state. (The sorting of a table as part of some physical operations is one such example.)

We develop three somewhat diverse planning algorithms to address our query optimization problem: a delete-free algorithm, an optimal A* algorithm, and a greedy algorithm, 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.

The work presented here is in its early stages, but is sufficiently advanced that there are interesting results and lessons to share. It introduces an interesting application to the AI Planning community, and a challenging problem to those interested in applications for cost-optimal (delete-free) planning and the more specific unaddressed problem of context-sensitive cost-optmal (delete-free) planning.

2. Preliminaries

We begin with a review of necessary relational database background and terminology. Conjunction queries (CQ) in SQL take the following 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 $r_i.a = x_l$ for a an attribute (column) of the relation (table) R_i. This can be equivalently written as a *predicate calculus*-style comprehension of the form:

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

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*.

A typical query compiler and optimizer in modern relational database systems performs several steps to produce a *query plan*, that is instructions to the *query execution phase* that is ultimately responsible for retrieving the data and answering the user's query. The query optimizer's phases range from parsing, type-checking, view expansion, etc., to rule-based query rewriting and cost-based *query optimization*. In the following we focus on *cost-based* optimization for *conjunctive queries*, that roughly correspond to the most common queries in SQL, the so called `SELECT`-blocks. Indeed, this part of optimizing queries is commonly considered the corner stone of relational query optimization since the original System R (Selinger et al. 1979).

2.1 Operators for CQ Query Plans

The *query plans* for conjunctive queries are responsible for accessing the 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 appropriate 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 (such as in the case of constant selections) or can be added in post-processing (projections[1]).

[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.

Access Paths The primitive relational operations are the *access paths* (APs), operators responsible for retrieving the *raw* data from relational storage (typically, 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). Formally, we can describe the access paths for a relation R as 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 the following 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, and Y is a set of variables that correspond to the attributes actually stored in the index (typically $X = Y$).

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

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

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_2 for `Id`; we will see later how this can be used for so-called *index only* query plans.

Join Operators To combine the results of access path invocations into query results, the *join* operators (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); for the purposes of this paper we focus on the above two joins without loss of generality.

To simplify the presentation we only consider left-deep query plans in this paper (this is similar to System R and is fully general for iterator-based plans that do not materialize intermediate results).

Example 2 For a query:

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

written as a comprehension as

$$\{x_1, x_2 \mid \texttt{Emp}(r) \wedge \texttt{EmpId}(r, x_1) \wedge \texttt{EmpName}(r, x_2)\}$$

we expect the following query plans, based on whether x_1 or x_2 (or neither) is a 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 unclustered indexing, index-intersection search, 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 (that store the actual data); we follow a simple System-R style cost model to illustrate the approach (however, more advanced cost models can be easily used as well). We collect the following:

- 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 (no. of 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 query plans (in disk page reads).

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 $\texttt{needs-}R$, $\texttt{has-}R$, and $\texttt{bound}$ 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

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

there is an action:

Action $\texttt{NLJ-}R\texttt{AP}$
pre: $\texttt{needs-}R(?r)$,
$\texttt{bound}(?x_{i_1})$, ..., $\texttt{bound}(?x_{i_l})$
post: $\texttt{has-}R(?r)$ $\texttt{has-}Ra_1(?r, ?x_1)$ ... $\texttt{has-}Ra_k(?r, ?x_k)$
$\texttt{bound}(?x_1)$ ... $\texttt{bound}(?x_k)$

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:

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

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

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

Example 3 For the query

$$\{x_1, x_2 \mid \texttt{Emp}(r) \wedge \texttt{EmpId}(r, x_1) \wedge \texttt{EmpName}(r, x_2)\}$$

with parameter x_1 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. In the planner this corresponds to exploring the following sequence of states:

1. $\texttt{needs-Emp}(r)$, $\texttt{needs-EmpId}(r, x_1)$, $\texttt{needs-EmpName}(r, x_2)$, $\texttt{bound}(x_1)$
2. $\texttt{needs-Emp}(r)$, $\texttt{needs-EmpId}(r, x_1)$ $\texttt{needs-EmpName}(r, x_2)$, $\texttt{bound}(x_1)$, $\texttt{has-Emp}(r)$, $\texttt{has-EmpId}(r, x_1)$, $\texttt{bound}(r)$
3. $\texttt{needs-Emp}(r)$, $\texttt{needs-EmpId}(r, x_1)$, $\texttt{needs-EmpName}(r, x_2)$, $\texttt{bound}(x_1)$, $\texttt{has-Emp}(r)$, $\texttt{has-EmpId}(r, x_1)$, $\texttt{bound}(r)$ $\texttt{has-EmpName}(r, x_2)$, $\texttt{has-EmpBoss}(r, x_3)$, $\texttt{bound}(x_2)$, $\texttt{bound}(x_3)$

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

1. $\texttt{needs-Emp}(r)$, $\texttt{needs-EmpId}(r, x_1)$, $\texttt{needs-EmpName}(r, x_2)$, $\texttt{bound}(x_1)$
2. $\texttt{needs-Emp}(r)$, $\texttt{needs-EmpId}(r, x_1)$, $\texttt{needs-EmpName}(r, x_2)$, $\texttt{bound}(x_1)$, $\texttt{has-Emp}(r)$, $\texttt{has-EmpId}(r, x_1)$, $\texttt{has-EmpName}(r, x_2)$, $\texttt{has-EmpBoss}(r, x_3)$, $\texttt{bound}(x_2)$, $\texttt{bound}(x_3)$

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 sorting steps in the plan.

We use the fluent $\texttt{asc}(x)$ 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, however, that unlike, e.g., the `bound` fluent, the sorted properties of variables 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 $\texttt{Sort-on-?}x$: (sort results of the current plan on x)
pre: $\texttt{bound}(?x)$
post: $\texttt{asc}(?x)$ $\neg\texttt{asc}(?y)$ for all other variables $?y$

Action $\texttt{MJ-on-?}x\texttt{-}AP$: (add a *merge-join* on variable x with the access path AP, assuming AP is also sorted on x)
pre: $\texttt{bound}(?x)$ asc$(?x)$
post: effects of AP as for NLJ
$\neg\texttt{asc}(?y)$ for all other variables $?y$

Action $\texttt{MSJ-on-?}x\texttt{-}AP$: (add a *sort-merge-join* on variable x with the access path AP, assuming AP is not sorted on x)
pre: $\texttt{bound}(?x)$ $\texttt{asc}(?x)$
post: effects of AP as for NLJ
$\neg\texttt{asc}(?y)$ for all other variables $?y$

We also add $\texttt{asc}(x)$ 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 an 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 previously described problems with the domain specific solvers presented in Section 5.

In more detail, 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. For example, the cost of executing $\texttt{NLJ-}R\texttt{Scan}$ in state s is $pages(R)ceil(size(s)/\texttt{buf-sz})$, that is we must join every page of tuples in the current store with every page of tuples in R (where `buf-sz` is the number of tuples we buffer before scanning R). For brevity, we omit a full description of the cost functions we employ, noting instead that they are closely based on those used in System R.

4. Query Planning as AI Planning

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, we make the following observations:

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

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

Obs 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.*

We highlight these seemingly straightforward observations because they suggest the potential for exploiting advances in delete-free cost-optimizing delete-free planning techniques (e.g., (Gefen and Brafman 2012; Haslum, Slaney, and Thiébaux 2012; Pommerening and Helmert 2012)). Perhaps less encouraging is the observation that delete-free planning owes much of its computational advantages to the property that partial plans expand monotonically and a final ordering of groupings of actions can be extracted quite easily. Unfortunately, with a *context sensitive* cost model, order matters, and many of the gains afforded by cost-optimizing delete-free planning are lost at least with respect to current implementations. Nevertheless on the positive side, as we move beyond relational cost-based join-order optimization and consider other criteria for defining plan quality and other physical operators for realizing a query/information gathering plan, we see that the models of cost are often context *independent* and thus, again by inspection, we make the following observation:

Obs 4 *A number of query and information-gathering optimization problems are context-independent cost-optimizing delete-free planning problems.*

This bodes well for the application of delete free planners to the generation of some classes of optimized information-gathering plans.

5. Generating Query Plans

Following from the observations in Section 4, 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. Note that while the heuristics have elements that are specific to the domain of join-order optimization, the general structure of the algorithms can be used for a diversity of information gathering/query plan optimization tasks simply by changing the cost function and the heuristic.

5.1 Fast Delete-Free Plan Exploration

Algorithm DF computes plans that do not include sorting actions. This prevents actions that merge on variables other than input variables. 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 the algorithms 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 $A_u(s)$. 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. Details of the algorithm follow.

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 $c_*$
```

Clearly DF does not guarantee to compute 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. In Section 6 we examine how effective this approach is empirically, at finding a high-quality plan. As noted below, this algorithm is also useful in providing a quality pruning bound for pruning of partial plans in our algorithms A* and GR.

5.2 A*

This algorithm A* is an eager A* search that uses a number of domain-dependent admissible heuristics. The code for A* 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.

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

The consistency and admissibility of the heuristic h_{blind} follow from the fact that 1 is a lowed-bound on the cost of any action and that the heuristic is clearly monotone.

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

To prove Proposition 2, we require to show that h_{admiss} is guaranteed to not over-estimate the cost of reaching the goal from s and is monotone. The former point can be seen by noting that the minimum cost of satisfying any relation R is 1 when we can currently execute a fetch action on R, that is when $R \in \mathcal{R}_I(s)$ and otherwise the cost is at-least $max(1, ceil(log_{200} R.pages))$, from the System R-based cost model. h_{admiss} is monotone because whenever we execute an action a we either reduce the size of the sets $\mathcal{R}(s')$ and $\mathcal{R}(s')$ by 1 or leave them unchanged and if $|\mathcal{R}_I(s')| < |\mathcal{R}_I(s)|$ then $|\mathcal{R}(s')| < |\mathcal{R}(s)|$.

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

To see that Proposition 3 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 1 *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. Note that all of our heuristics are admissible. We explored the development of more informative inadmissible heuristics, which could be used with admissible heuristics for sound pruning. Unfortunately, as of this writing, we have found no superior inadmissible heuristic.

5.3 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 sole difference between GR and A* is that GRorders 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.

Theorem 2 *An expanded state s can be safely pruned when $g(s) + h(s)$ exceeds the current bound without sacrificing optimality.*

The proof idea is based on observing that the bounds are sound and heuristics admissible. The next Theorem follows as either we consider successors of a node or the node is pruned; lack of further nodes implies that an optimal solution was found (or none exists).

Theorem 3 *If* GR *terminates on a query planning problem of the type we consider, then it returns an optimal solution.*

6. Evaluation

The question we'd like to address with our experimental evaluation is how AI planning techniques perform relative to the state of the art in relational database query planning. In particular, we'd like to know whether AI planning techniques have the potential to compute higher quality query plans faster than the state of the art. Unfortunately, the state of the art in relational database technology is embedded within proprietary systems, precluding systematic comparison. Similarly there are no suitable benchmarks for objective comparison of underlying algorithms. Instead what we do know and can leverage is that typical relational database systems employ time-limited planning algorithms and commonly trade plan optimality for reducing the time to find a reasonable query plan. For example, the classical System R query optimizer only considers plans that utilize Cartesian products as a last resort despite the fact that there are well-known examples where this heuristic leads to suboptimal query plans (this happens, e.g., when two relations with small cardinality are joined with a large relation indexed on attributes retrieved from both of these small relations). Other commercial systems utilize variants of time-constrained branch-and-bound algorithm in which the system attempts to allocate approximately the same time quota to exploring the "upper part" of the tree of possible plans as to the "lower part" (that is commonly much larger). Also, commercial query optimizers commonly take advantage of additional schema information, such as the key and foreign key constraints, that can be used to improve the cost estimation (this, however, makes the optimization problem easier as the additional information tends to make the cost difference between varying query plans more pronounced).

As the time to optimize queries is quite constrained in existing database systems the size of join-order problems that relational database optimizers currently solve (to optimality), in terms of the number of relations in a query, and the ratio of the number of variables to number of relations is rather small (again, in the absence of additional schema information). The actual algorithms and their performance in most commercial optimizers is a closely guarded business secret and only anecdotal evidence—that optimality is only guaranteed for about up to 10 way joins—is available.

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, with the general objective of determining the relative merits and shortcomings of the algorithms and heuristics, and 2) to get some sense of whether AI automated planning techniques held some longer-term promise for cost-based join order optimization, in particular, relational database query optimization and 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. Proprietary database systems usually allocate a short period of time for query planning (on the order of seconds) and for our algorithms to be practically useful they must find high-quality plans in this time frame.

In the absence of existing query planning benchmarks, we tested our planning systems on randomly generated database schemata and queries. Each generated schema consists of tables with between 2 and 10 attributes (not greater than half of the number of variables in the associated query). 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. We performed the following experiments.

6.1 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.

The results of this experiment can be seen in Figure 1. The number of problems that can be solved to optimality quickly drops off for both planners when the number of re-

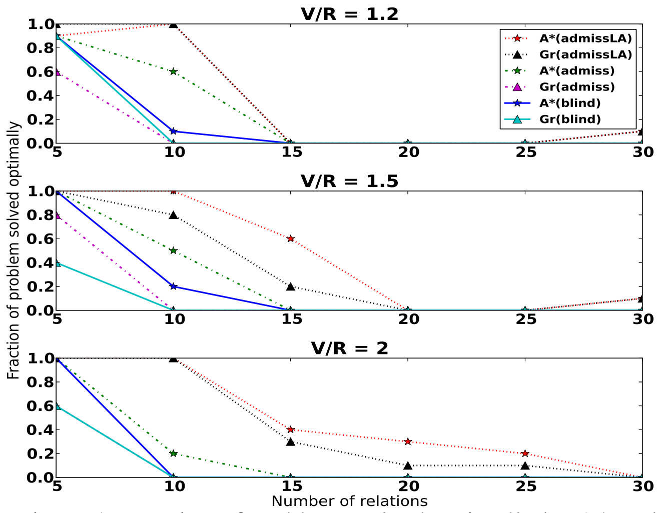

Figure 1: Fraction of problems solved optimally by A* and GR with different heuristics (2GB, 30 min time out).

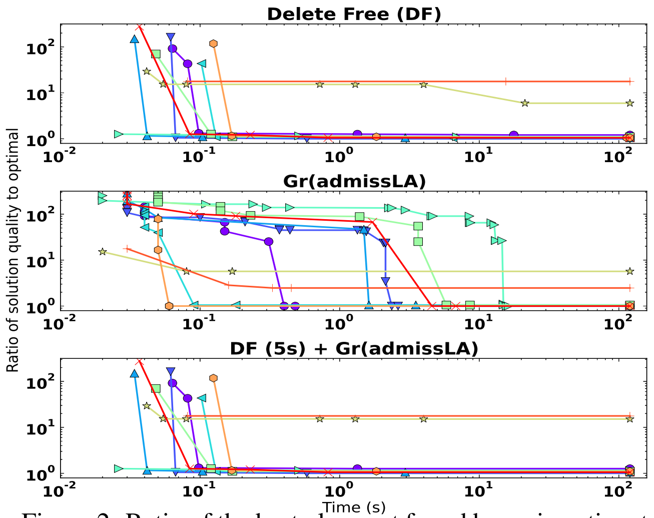

Figure 2: Ratio of the best plan cost found by a given time to the known optimal cost for problem instances with $R = 10$ and $V = 20$ for DF, GR, and DF run for 5 seconds, followed by GR with the resulting bound. GR used the $h_{admissLA}$ heuristic and all approaches had 2GB, 2 min time out.

lations is 15 or higher with A* and heuristic $h_{admissLA}$ performing the best. Problems with a higher V/R ratio are somewhat easier than those with a low ratio. As expected, the different heuristics all give significantly different performance. h_{blind} heuristic, due to its lack of guidance leads to the planners running out of memory on all but the smallest problems. h_{admiss} leads to considerably better performance than h_{blind}, as it returns a value proportional to the number of relations left to satisfy and directs the planner towards the goal. However, as it ignores the size of the current and all intermediate states, it fails to distinguish between many plans. $h_{admissLA}$ provides improved performance by at least considering the size of the state that results from executing each action, even though it ignores the sizes of all subsequent states. An important remaining challenge is to develop an admissible heuristic that can take into account the sizes of states beyond one step ahead without doing full k step lookahead, which was found to be too costly to be an effective heuristic.

In general, the planners solved those problems for which they could quickly find a sequence of cheap actions that bind a large portion of the variables while keeping the size small. Experiments show that those problems that can be solved optimally can be solved quickly, usually within a few seconds. On many of the remaining problems, the algorithms run out of memory exploring plateaus close to the goal.

6.2 Experiment 2: Fast High-Quality Plans

Given that our optimal algorithms do not scale acceptably for real world use, where an acceptable plan must be found within a few seconds, we also explored the use of several sub-optimal, any time algorithms.

In these experiments we ran DF and GR with no initial bounds and also GR with an initial bound generated by running DF for 5 seconds. Each of these algorithms was run for a total time of 2 minutes and plans were recorded as they were produced.

An important question to answer about these algorithms is how the quality of the plans that they find 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 shows a representative sample of these results using the $h_{admissLA}$ heuristic. It shows that for most problems, the sub-optimal algorithms find solutions that 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.

There was a small subset of problems for which these approaches are unable to find solutions within 10 times the cost of the optimal, even after 2 minutes. From the available evidence, this issue increases at larger problem sizes. Further analysis of the structure of these difficult problems is needed to determine what prevents the greedy approaches from finding high-quality solutions.

The approach of using DF to produce an initial bound for GR did not lead to a significantly better solution quality (after similar or longer run-times) and, due to the time required to find the initial bound, is impractical. This is not particularly surprising as GR very quickly finds upper bounds on plan cost anyway. We therefore omit this approach from further discussion here.

As well as comparing DF and GR to the optimal solutions, we performed extensive experiments to compare them to each other. Figure 3 shows the costs of the best plans found by DFand GR with all three heuristics 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.

From looking at the plots, it is clear that 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 can be 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.

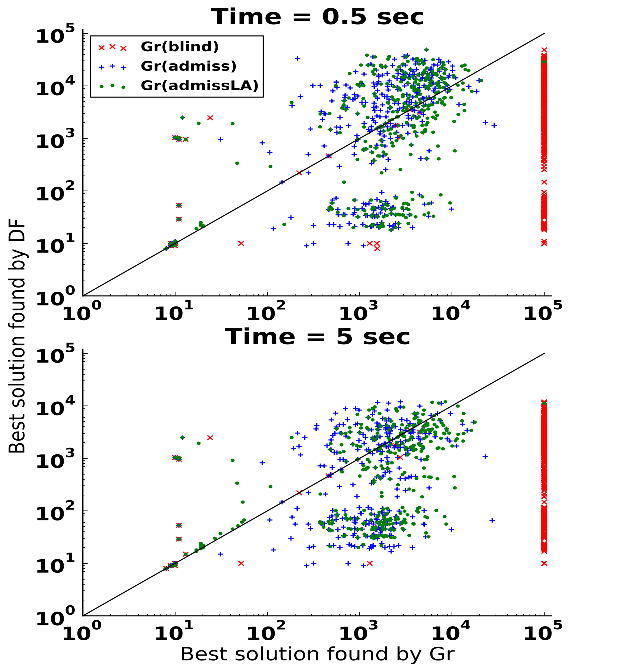

Figure 3: The costs of the best plans found by GR and DF after 0.5 and 5 seconds for all problem instances in our experiment set. Problems that could not be solved by an algorithm were assigned a cost of 10^5 in that case.

This pattern of performance is not particularly surprising. GR initially performs better than DF because the $h_{admissLA}$ heuristic is considerably more informative than the action selection heuristic employed by DF when generating delete-free plans. This allows GR to very quickly find reasonably good plans. 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. This behaviour can explain the cluster of problems on which GR consistently performs worse than DF.

7. Summary

This paper reports on preliminary findings relating to the applicability of AI automated planning techniques to the optimization of information gathering in general and to the generation of high quality cost-based join-order optimized query plans in particular. We observed that join-order query planning is a delete-free planning problem, and that query optimization is a context-sensitive cost-optimal delete-free planning problem. However, when we considered the broader problem that includes merge-joins, the delete free nature is lost. We developed delete-free, A*, and greedy planning algorithms which we combined with domain-dependent heuristics for generating optimized query plans. The latter two algorithms were guaranteed to produce optimal plans under certain conditions. Note that while the heuristics have elements that are specific to join-order optimization, the general structure of the algorithms can be used for a diversity of information gathering/query plan optimization tasks simply by changing the cost function and the heuristic. Experimental results are promising. Perhaps most notably, our planners could generate *optimal* query plans of a size that is highly competitive with those reputed to be solved by commercial systems. This work presents an interesting and challenging application domain for AI planning technology and a promising approach to solving a diversity of problems related to optimized information gathering.

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.

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.

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

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.

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.

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.

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.

Ladwig, G., and Tran, T. 2010. Linked data query processing strategies. In *Proceedings of The 9th International Semantic Web Conference*, 453–469.

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.