Query Optimization over Crowdsourced Data*

Hyunjung Park
Stanford University
hyunjung@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

Deco is a comprehensive system for answering declarative queries posed over stored relational data together with data obtained on-demand from the crowd. In this paper we describe Deco's cost-based query optimizer, building on Deco's data model, query language, and query execution engine presented earlier. Deco's objective in query optimization is to find the best query plan to answer a query, in terms of estimated monetary cost. Deco's query semantics and plan execution strategies require several fundamental changes to traditional query optimization. Novel techniques incorporated into Deco's query optimizer include a cost model distinguishing between "free" existing data versus paid new data, a cardinality estimation algorithm coping with changes to the database state during query execution, and a plan enumeration algorithm maximizing reuse of common subplans in a setting that makes reuse challenging. We experimentally evaluate Deco's query optimizer, focusing on the accuracy of cost estimation and the efficiency of plan enumeration.

1. INTRODUCTION

Crowdsourcing [8] enables programmers to incorporate human computation into a variety of tasks that are difficult for computer algorithms alone to solve well, e.g., tagging images, categorizing products, and extracting sentiments from Tweets. However, to take advantage of crowdsourcing in practice, programmers have to write custom code using low-level APIs, which leads to some common challenges even for simple applications: improving data quality by resolving inconsistencies in crowdsourced data, integrating crowdsourced data with existing data, and optimizing crowdsourcing workflows for monetary cost and latency.

To address these challenges, we are developing *Deco* (for "declarative crowdsourcing") [18, 19, 20], a system that answers declarative queries posed over stored relational data together with data obtained on-demand from the crowd. In [18], we defined a data model and a query language for Deco: The data model was designed to be general, flexible, and principled; the query language extends SQL with simple constructs necessary for crowdsourcing. Based on the data model, we defined a precise semantics for arbitrary queries. In [20], we described Deco's query plans, and how the system executes them to minimize monetary cost while reducing latency. Deco's query execution engine uses several novel techniques to overcome the limitations of traditional query execution models in the crowdsourcing setting, including a hybrid execution model, incremental view maintenance inside query plans, two-phase query execution, and dynamic fetch prioritization.

In this paper, we describe Deco's cost-based query optimizer. Our goal is to find the best query plan to answer a query, where "best" means the least estimated monetary cost across all possible query plans. (Recall in [20] we described how to execute a *given* plan with least monetary cost. Now we seek to find the best overall plan.) Although Deco's query optimizer has similar overall structure to a traditional query optimizer, there are several fundamental differences in plan costing and selection that needed to be addressed in our setting, to reflect Deco's query semantics and plan execution strategies.

Distinguishing between existing vs. new data: To estimate monetary cost properly, Deco's cost model must distinguish between existing data obtained by past queries (or otherwise present in the database), versus new data to be obtained on-demand from the crowd. Existing data is "free", so all of the monetary cost is associated with new data. Deco's cost model must take into account the existing data that might contribute to the query result, in order to estimate the cardinality of new data required to produce the result. In our setting, the estimated cardinality of new data directly translates to the monetary cost to answer the query.

Estimating cardinality and database state simultaneously: As Deco executes a query, it also changes the state of database by storing new data obtained from the crowd. Cardinality estimation obviously must be based on some final database state, which needs to be estimated as well. Deco's cardinality estimation algorithm simultaneously estimates cardinality and the end-state of the database, using a top-down recursive process: Starting from the root operator, each operator passes a requirement for the end-state to its subplans; as the recursion unwinds, the subplans return their estimated cardinality based on the expected end-state. Note that Deco's cardinality estimation is holistic: the cardinality of a subplan depends on the entire plan, not just the subplan.

Exploiting limited subplan reuse opportunities: In Deco, different physical plans corresponding to the same logical plan may produce different query results. (We will see in our query semantics that many possible results are valid in a crowdsourcing setting.) Thus, unlike in traditional optimization, estimated cardinality is a property of physical plans rather than logical plans. As a result, in comparison with traditional plan enumeration, there are far

*This work was supported by the NSF (IIS-0904497), the Boeing Corporation, and a KAUST research grant.

fewer opportunities to avoid redundant computation across alternative plans or prune inferior subplans early. Combined with holistic cardinality estimation, Deco does need to explore a large number of plans, but we reuse common subplans to the extend possible.

Our experiments show that Deco's query optimizer succeeds in choosing inexpensive query plans for a wide variety of settings, within reasonable optimization time.

The rest of the paper proceeds as follows:

- We review Deco's data model, query language, and query execution engine (Section 2), summarizing material from [18, 20].
- We describe how Deco estimates the monetary cost of executing a given query plan (Section 3).
- We describe the search space of alternative Deco query plans (Section 4).
- We present Deco's plan enumeration algorithm, which explores the search space and applies the cost estimation algorithm to choose the best plan (Section 5).
- We experimentally evaluate Deco's query optimizer in terms of the accuracy of cost estimation and the efficiency of plan enumeration (Section 6).

Related work is covered in Section 7, and we conclude with future directions in Section 8.

2. PRELIMINARIES

We begin by reviewing Deco's data model and query language, then we briefly describe Deco's query execution engine. For more details see [18, 20], from which these summaries are drawn.

2.1 Data Model and Query Language

Conceptual Relation: *Conceptual relations* are the logical relations specified by the Deco schema designer and queried by end-users and applications. The schema designer also partitions the attributes in each conceptual relation into *anchor attributes* and *dependent attribute-groups*. Informally, anchor attributes typically identify "entities" while dependent attribute-groups specify properties of the entities.

As a running example, suppose our users want to query two conceptual relations with information about countries and cities:

Country(country, [language], [capital])
City(city, country, [population])

Each dependent attribute-group (single attributes in this case) is enclosed within square brackets.

Raw Schema: Deco is designed to use a conventional RDBMS as its back-end. The *raw schema*—the schemas for the data tables actually stored in the underlying RDBMS—is derived automatically from the conceptual schema, and is invisible to both the schema designer and end-users. For each relation R in the conceptual schema, there is one *anchor table* containing the anchor attributes, and one *dependent table* for each dependent attribute-group; dependent tables also contain anchor attributes.

For our example relation Country, we have the raw schema:

CountryA(country)
CountryD1(country, language)
CountryD2(country, capital)

Fetch Rules: *Fetch rules* allow the schema designer to specify how data can be obtained from humans. A fetch rule takes the form $A_1 \Rightarrow A_2 : P$, where A_1 and A_2 are sets of attributes from one conceptual relation (with $A_1 = \varnothing$ permitted), and P is a *fetch procedure* that implements access to human workers. When invoked, the fetch rule $A_1 \Rightarrow A_2$ obtains new values for A_2 given values for A_1, and populates raw tables using those values for attributes $A_1 \cup A_2$. The schema designer also specifies a fixed monetary cost for each fetch rule, to be paid to human workers once they complete the fetch rule.

Here are some example fetch rules for our running example:

- [Country] $\varnothing \Rightarrow$ country: Ask for a country name, inserting the obtained value into raw table CountryA.
- [Country] country $\Rightarrow$ capital: Ask for a capital given a country name, inserting the resulting pair into table CountryD2.

For a full description of the allowable fetch rules in Deco, see [18].

Resolution Rules: Suppose we've obtained values for our raw tables, but we have inconsistencies in the collected data. We use *resolution rules* to cleanse the raw tables—to get values for conceptual relations that are free of inconsistencies. For each conceptual relation, the schema designer can specify a resolution rule $\varnothing \rightarrow A : f$ for the anchor attributes A treated as a group, and one resolution rule $A \rightarrow D : f$ for each dependent attribute-group D. Resolution function f is a black-box that adheres to a simple API, taking as input a set of values for the right-hand side attributes (corresponding to a specific value for the left-hand side) and returning a "cleaned" set of values. If the empty set is returned, more input values are needed to produce an output. In addition, the schema designer should specify the minimum and average number of input values needed for f to produce an output value, which is used both for cost estimation and to optimize plan execution.

In our example, we might have the following resolution rules:

- [Country] $\varnothing \rightarrow$ country : *dupElim*
- [Country] country $\rightarrow$ language (or capital) : *majority-of-3*

Resolution function *dupElim* produces distinct country values for Country. Resolution function *majority-of-3* produces the majority of three or more language (or capital) answers for a given country. We assume a "shortcutting" version that can produce an answer with only two values, if the values agree. Note any resolution functions are permitted, not just the types used here for illustration.

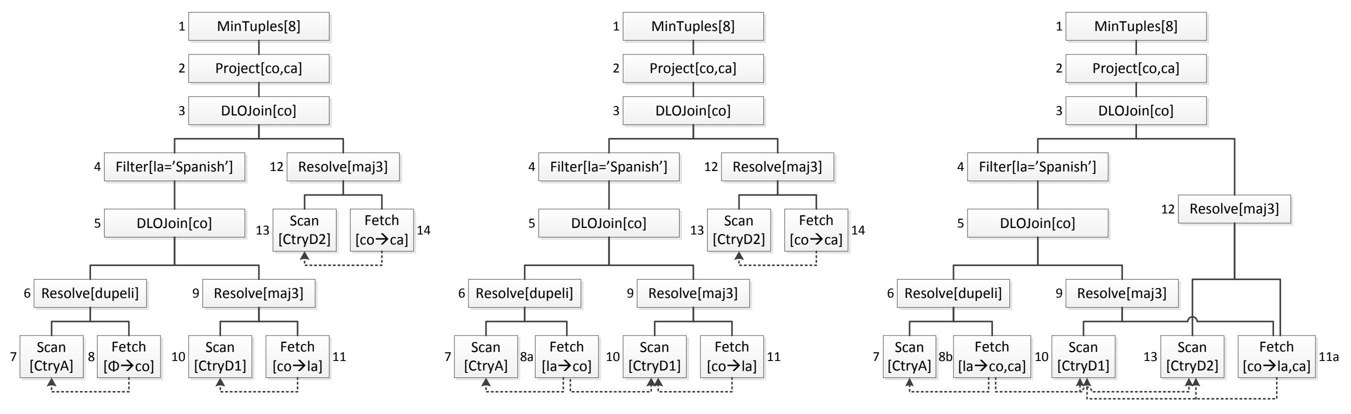
Data Model Semantics: The semantics of a Deco database is defined as a potentially infinite set of *valid instances* for the conceptual relations. A valid instance is logically defined by a *Fetch-Resolve-Join* sequence: (1) *Fetching* additional data for the raw tables using fetch rules; this step may be skipped. (2) *Resolving* inconsistencies using resolution rules for each of the raw tables. (3) *Outerjoining* the resolved raw tables to produce the conceptual relations.

Query Language and Semantics: A Deco query Q is simply a SQL query over the conceptual relations. Deco's query semantics dictate that the answer to Q must represent the result of evaluating Q over some valid instance of the database. Since by this semantics Q could always be answered correctly using the "current" valid instance (which may be empty), we add to our query language a "MinTuples n" constraint: The result of Q must be over some valid instance for which the answer has at least n tuples without NULL attributes. (As future work we will address other constraints such as "MaxCost c" and "MaxTime t" [20].)

In this paper we consider Select-Project-Join queries.

2.2 Query Execution

Suppose the query optimizer has selected a plan for a query with "MinTuples n" constraint. The query execution engine is designed with the primary goal of producing at least n result tuples while minimizing monetary cost. A secondary goal is to reduce latency by exploiting parallelism when accessing the crowd. Achieving both goals during query execution translates to the following over-



(a) Basic **(b) Reverse** **(c) Hybrid**

Figure 1: Query plans

all objective: Maximize parallelism while fetching data from the crowd (to reduce latency), but only when the parallelism will not waste work (to minimize monetary cost).

In our design of Deco, the query optimizer selects the plan with the least estimated monetary cost, without explicitly considering latency. Then the execution engine considers both cost and latency as outlined above. We could alternatively have used a multi-objective approach [4] in the optimizer, where it estimates both monetary cost and latency, and offers a tradeoff between them. We did not use this approach in our initial optimizer since latency is difficult to estimate in a crowdsourcing setting [10, 15], and empirically we found cost and latency to be correlated. For now, users can trade higher cost for lower latency by increasing the degree of parallelism when executing the chosen plan (as shown with experimental results in [20]). As future work, we may consider multi-objective optimization and compare the results with our current approach.

To meet our objective while respecting Deco's semantics, we incorporated several novel techniques into Deco's query execution engine: a hybrid execution model, incremental view maintenance inside query plans, two-phase query execution, and dynamic fetch prioritization. These techniques are discussed in detail in [20]. Here we summarize only those aspects that are relevant to the query optimization problem.

- Deco executes queries in two phases. In the *materialization* phase, the "current" result is materialized using the existing contents of the raw tables without invoking additional fetches. If this result does not meet the MinTuples constraint, the *accretion* phase invokes fetch rules to obtain more results. This second phase extends the result incrementally as fetch rules complete, and invokes more fetches as necessary until the MinTuples constraint is met.
- In certain cases, minimizing monetary cost is especially difficult because existing data can make some fetches more profitable than others. Deco's query execution engine incorporates an algorithm that identifies specific "good" fetches. Individual query operators do not always have enough information to choose the good fetches, so our approach is to invoke more fetches than needed in parallel, but prioritize them so the better fetches are more likely to complete first (thus minimizing cost). When enough data has been obtained, outstanding fetches are canceled.

We will see in Section 3 how Deco's cost estimation algorithm takes these aspects into account.

Example Query Plan: Figure 1a shows one possible query plan for the following query on the example database from Section 2.1:

```
SELECT country, capital  FROM Country
WHERE language='Spanish'  MINTUPLES 8
```

Abbreviations in the plan should be self-explanatory. Deco-specific query operators used in this plan are as follows:

- The *Fetch* operator corresponds to a fetch rule $A_1 \Rightarrow A_2 : P$. It receives values for A_1 from its parent and invokes procedure P. It does not wait for answers, so many procedures may be invoked in parallel. When values for $A_1 \cup A_2$ are returned by P, they are sent to one or more *Scan* operators that work with the *Fetch* operator. *Scan* operators insert the new tuples into raw tables and also pass them up to their parent.
- The *Resolve* operator corresponds to a resolution rule $A_1 \rightarrow A_2 : f$. It receives from its child tuples containing attribute values for $A_1 \cup A_2$. It applies function f based on groups of tuples with the same A_1 value, and passes up resolved values for $A_1 \cup A_2$.
- The *DLOJoin* (for Dependent Left Outerjoin [11]) operator is similar to a relational indexed nested-loop join. It receives attribute values from its outer child, which it passes to its inner child to obtain additional attributes that constitute join result tuples. In our plans, *DLOJoin* always obtains anchor attributes from its outer child and dependent attributes from its inner.
- The *MinTuples* operator determines when the answer is complete.

We describe the case where there are no existing tuples in any of the raw tables. First, the root operator sends eight *getNext* requests to its child operator (based on MinTuples). These requests propagate down the left (outer) side of the joins, and eventually invoke fetch rule $\varnothing \Rightarrow$ country eight times, without waiting for answers. At this point, there are eight outstanding fetches in parallel.

As these outstanding fetches complete, the new country values are inserted into raw table CountryA and passed up the plan by the *Scan* operator. Through the *DLOJoin*, new countries trigger invocations of fetch rule country $\Rightarrow$ language. For each country value, two instances of this fetch rule are invoked in parallel because the resolution function *majority-of-3* requires at least two language values as input. At this point, we may have many fetches going on in parallel: some to fetch more countries, and some to fetch languages for given countries.

Until the MinTuples constraint is met, the query plan invokes additional fetches as needed. For example, if the two instances of fetch rule country $\Rightarrow$ language for a given country return two different language values, the plan invokes another instance of the same fetch rule to obtain the third language value. Likewise, as soon as a resolved language value for a certain country turns out to not be Spanish, the plan invokes a new instance of fetch rule $\varnothing \Rightarrow$ country. For countries whose resolved language value is Spanish, the plan obtains capital values for the country, in parallel with other fetches similarly to how language values were obtained. Once the MinTuples constraint is met, the result tuples are returned to the client.

For further details of Deco's query execution engine, see [20].

3. COST ESTIMATION

We describe how Deco's query optimizer estimates the cost of executing a given query plan. As discussed in Section 1, our optimization metric is the total monetary cost incurred by fetches. Thus, Deco's cost estimation algorithm takes as input a query plan and statistics about data (both local data and crowdsourced data), and produces as output the estimated cost in dollars. In a sense our cost model estimates resource consumption in a similar fashion to traditional database systems, except the resource is money instead of CPU, I/O, and communication costs.

We assume there is a fixed monetary cost associated with each fetch rule, and this cost is specified by the schema designer through Deco's data definition language (DDL). For example, fetch rules country $\Rightarrow$ language and country $\Rightarrow$ capital may cost \$0.03 and \$0.05 per fetch, respectively. Although costs may differ across fetch rules, we assume the cost-per-fetch of one rule $A_1 \Rightarrow A_2$ does not depend on the specific values for A_1 (even though, conceivably, the level of difficulty to answer such questions may vary based on the values for A_1).

It turns out we can reduce the monetary cost estimation problem to the cardinality estimation problem. However, the notion of cardinality from traditional databases has to be adjusted, since Deco inserts new tuples into raw tables during query execution. In Deco, we estimate cardinality of a subplan or operator as the total number of output tuples expected in order to obtain a query result with a sufficient number of tuples. Since no Deco query operators except *Fetch* cost money, we have the following formula for estimated monetary cost:

$$monetary\ cost = \sum_{\textit{Fetch}\ \text{operator}\ F} F.cost\text{-}per\text{-}fetch \times F.card \quad (1)$$

where $F.cost\text{-}per\text{-}fetch$ and $F.card$ denote the cost-per-fetch and estimated cardinality of *Fetch* operator F, respectively.

Because the schema designer specifies $F.cost\text{-}per\text{-}fetch$, estimating the monetary cost amounts to estimating $F.card$, the cardinality of each *Fetch* operator F. Of course to estimate the cardinality of each *Fetch* operator, we need to estimate cardinality for other parts of the plan as well. The next section gives a high-level description of our challenges in cardinality estimation. Sections 3.2 and 3.3 specify the cardinality estimation algorithm and provide cardinality and cost estimation examples.

3.1 Challenges and Approach

Deco's query semantics gives us several new challenges in cardinality estimation.

Existing vs. New Data: Under the valid-instance semantics, Deco's query result must reflect all existing data in the raw tables. In addition, Deco obtains new data from the crowd until a sufficient number of result tuples are present. Since existing data is "free" and new data is not, Deco's cost model must distinguish between existing data and new data to estimate monetary cost properly. We will see in Section 3.2 how *Resolve* and *Fetch* operators take into account the amount of relevant existing data to estimate the cardinality of required new data.

Statistical Information: To estimate cardinality we require some statistical information about both existing data and new data. For existing data, we use statistical information maintained by the back-end RDBMS. Since we have the limited knowledge about resolution functions, we rely on the statistics of the resolved raw tables rather than the raw tables. For data obtained from the crowd, we primarily rely on information provided by the schema designer and/or end-user. We require a *selectivity factor* to be provided by the schema designer for each resolution function (further discussed below). For filters, we allow the end-user to provide a selectivity factor; if none is provided we resort to "magic numbers" [5]. As a simple alternative, the end-user can specify that new data is expected to follow the same statistics as existing data.

Selectivity Factors: The selectivity factor for a predicate has a slightly different meaning in our setting than in traditional databases. Essentially, a selectivity factor of σ for a predicate p over conceptual relation R says that a tuple of R (current or future) has a σ chance of satisfying p. For example, the selectivity of predicate language='Spanish' may be around 0.1 in the Country relation, because there are about 20 Spanish-speaking countries out of about 200 countries in the world. For resolution functions, the selectivity factor estimates how many output tuples are produced on average by each input tuple. For example, the selectivity factor of resolution function *majority-of-3* with shortcutting depends on how often shortcutting is expected to happen, ranging from 1/3 (when the first two values are never expected to agree) to 1/2 (when the first two values are always expected to agree).

Estimating Cardinality and Database State Simultaneously: As Deco executes a query, it also changes the state of database because it stores newly crowdsourced data. Thus, cardinality estimation obviously must be based on some estimated final database state, and our algorithm estimates cardinality and end-state simultaneously. Note that the end-state depends on the entire query plan: a subplan cannot estimate the end-state without considering the rest of plan. Therefore, Deco's cardinality estimation must be holistic.

Cardinality estimation is a top-down recursive process. Each operator calls its subplan(s) with a set of predicates, and the number of tuples it needs from its subplan that satisfy the predicates. The process begins with the "MinTuples n" operator calling its child with no predicates and n tuples needed. As the recursion unwinds, subplans return their estimated cardinality to their parent (with some complexities discussed below).

For example, in Figure 1a, when the recursion reaches *Fetch* operator 8, the operator receives predicate language='Spanish' and eight tuples needed. Since the selectivity of the predicate is 0.1, the *Fetch* operator returns an estimated cardinality of 80. In Figure 1b (note the only change from Figure 1a is operator 8a, which obtains countries with a certain language), *Fetch* operator 8a is also called with predicate language='Spanish' and eight tuples needed. However, since the fetch rule in this case is instantiated with left-hand side value "Spanish", the operator expects the predicate to be satisfied by all fetched data, and returns estimated cardinality of 8.

In some cases, Deco's cardinality estimation is similar to estimating "stopping cardinality" for traditional queries with Top (or Limit) clauses [2]: The system estimates how many tuples a particular subplan must produce in order to produce a sufficient number of result tuples satisfying the Top clause. However, Deco's valid-instance semantics forces us to use all existing data in the raw tables regardless of the MinTuples clause, making cardinality estimation somewhat more complex.

Propagating Cardinality: During query execution, multiple raw tables and fetch rules may feed a single *Filter* operator. Thus, a

Filter operator may process different classes of data: data from existing raw tables; data obtained by fetch rules with a predicate corresponding to the *Filter*; data obtained by fetch rules without a predicate corresponding to the *Filter*. As a result, during cardinality estimation, we cannot apply just one selectivity at a given *Filter* operator. Instead, as cardinality estimates are returned up a plan, they need to notate the breakdown of cardinality according to the set of predicates that were received from the parent in the top-down phase, so the right selectivity is applied.

For example, in Figure 1a, when *DLOJoin* operator 5 returns its estimated cardinality of 80, it indicates that only eight tuples are expected to satisfy predicate language='Spanish'. On the other hand, in Figure 1b, *DLOJoin* operator 5 returns an estimated cardinality of 8, with all eight tuples expected to satisfy the predicate. This breakdown provides *Filter* operator 4 enough information to distinguish among different classes of data.

3.2 Cardinality Estimation Algorithm

Now we describe a procedure EstimateCard that estimates cardinality for each operator *op* in a query plan. This procedure is specialized for each operator type and takes two parameters:

- preds: an array of k predicates (with their selectivities)
- target: a target number of output tuples (at operator *op*) satisfying the predicates in preds

Using these two parameters, operator *op* receives information necessary for estimating its cardinality (and the end-state of the database) from *op*'s parent operator. To estimate cardinality, operator *op* recursively calls the EstimateCard procedure on its children with appropriate parameters, and uses its own arguments. Before the EstimateCard procedure returns, operator *op* stores its output in the following three member variables:

- *op*.card: estimated cardinality of operator *op*
- *op*.cards: an array of 2^k elements representing the breakdown of *op*.card by evaluation results of the k predicates in preds
- *op*.distincts: an array containing the number of distinct values for each output attribute

To define *op*.cards[i] ($0 \leq i < 2^k$) precisely, let $b_0, b_1, \dots, b_{k-1}$ denote the binary representation of i (i.e., $i = \sum_{j=0}^{k-1} b_j 2^{k-j-1}$ and $b_j \in \{0, 1\}$). The binary value b_j encodes whether predicate preds[j] is satisfied (b_j=1) or not. Thus, *op*.cards[i] is the estimated cardinality corresponding to the combination of predicate evaluation results encoded by the index i. The following equation holds by definition: *op*.card $= \sum_{0 \leq i \leq 2^k-1}$ *op*.cards[i]. Note that the default cards array is exponential in size, i.e., 2^k. If k is expected to be large, the implementation could assume independence and represent cards by the sum of arrays of size k, capturing different distributions of existing and new data while reducing the space and time complexity. We have not implemented this approach yet, but it is a simple modification to our system.

For *op*.distincts, no array element can be larger than *op*.card. Note we only need certain elements involved in join predicates, although our notation is defined for all attributes. These outputs are stored rather than simply returned to the caller for two reasons. First, we need card values of all *Fetch* operators to compute the total monetary cost. Second, a card value of a *Fetch* operator may be updated when the EstimateCard procedure is called again through another parent. (Technically, except for *Fetch* operators, these outputs could be returned to the caller.) To initiate cardinality estimation for a plan, we call root.EstimateCard(0, ∅).

Having defined the signature of the EstimateCard procedure, we now go through the actual implementation for each operator type.

```
MinTuples.EstimateCard(target, preds)
1 child.EstimateCard(this.minTuples, ∅)
2 this.card ← child.card
3 this.cards ← child.cards
4 this.distincts ← child.distincts
```

Since the root of every Deco query plan is a *MinTuples* operator, *MinTuples*.EstimateCard is the entry point of our cardinality estimation algorithm as a whole. It calls EstimateCard recursively on its child, with parameter target set to the number of tuples in the MinTuples clause, and no predicates for parameter preds. (Note that output tuples of the child operator of the root are guaranteed to satisfy all predicates in the query.) When the recursive call returns, the cardinality estimation algorithm terminates, and each F.card stores estimated cardinality for *Fetch* operator F, from which we calculate the estimated monetary cost using Equation (1).

```
Project.EstimateCard(target, preds)
1 child.EstimateCard(target, preds)
2 this.card ← child.card
3 this.cards ← child.cards
4 this.distincts ← $\Pi$ child.distincts
```

For the *Project* operator, the EstimateCard procedure simply continues the recursion because the cardinality does not change at all.

```
Filter.EstimateCard(target, preds)
1 child.EstimateCard(target, preds ∪ {this.pred})
2 this.card ← 0
3 this.cards ← $\{0, \dots, 0\}$
4 for $i = 0$ to $2^{\text{len(preds)}}-1$ do
5     this.card ← this.card + child.cards[$2i+1$]
6     this.cards[$i$] ← child.cards[$2i+1$]
7 end for
8 this.distincts ← $\min_{\text{elementwise}}$(
      child.distincts, {this.card, ..., this.card})
```

The *Filter* operator recursively calls EstimateCard on its child operator with its own predicate as well as the k predicates received from its parent operator. When the recursive call returns, child.cards contains 2^{k+1} elements. Then, the *Filter* computes its estimated cardinality by summing up the 2^k elements whose indexes indicate that its own predicate is satisfied.

```
DLOJoin.EstimateCard(target, preds)
1  outer.EstimateCard(target, preds)
2  $d$ ← outer.distincts[this.pred.left]
3  if outer.cards[$2^{\text{len(preds)}}-1$] > target then
4      inner.EstimateCard(
          $\alpha \times$ target $\times\ d$ / outer.card + $(1-\alpha) \times d$, ∅)
5  else
6      inner.EstimateCard($d$, ∅)
7  end if
8  this.card ← outer.card
9  this.cards ← outer.cards
10 this.distincts ← outer.distincts ∪ inner.distincts
```

For the *DLOJoin* operator, we first call EstimateCard on the outer child operator. We intentionally pass all predicates in preds to the outer, even though some dependent attributes are obtained from the inner. (We will see shortly how *Resolve* and *Fetch* operators use these predicates to estimate the number of required anchor values.) Note that we do not pass the left outerjoin predicate, which is always satisfied by definition.

Once the recursive call on the outer returns, we call EstimateCard on the inner child to eventually estimate the number of new dependent values required to produce a query result. Without considering dynamic fetch prioritization [20], parameter target would be simply $d =$ outer.distincts[this.pred.left]. However, when there are more anchor values than needed due to existing data, fetch prioritization takes effect as described in Section 2.2: Deco prioritizes those fetches filling in dependent attributes so that a sufficient number of result tuples are produced as soon as possible. As a result, some anchor values are not expected ever to be joined.

Since it is very difficult to predict the exact outcome of fetch prioritization due to its heuristic approach, we discount parameter target using a configurable weight $0 \leq \alpha \leq 1$. With $\alpha = 0$, we overestimate the number of new dependent values, because joining all d anchor values may eventually produce far more result tuples than needed. On the other hand, with $\alpha = 1$, we assume optimal fetch prioritization as well as no unfavorable correlations in the existing data, so we underestimate the number of dependent values. In Section 6.1, we empirically determine a good range for α.

```
*Resolve*.EstimateCard(target, preds)
compute this.card, this.cards and this.distincts based on the
    existing data in the resolved raw tables (using the back-end
    RDBMS)
$t \leftarrow$ this.cards[$2^{\text{len(preds)}}-1$]
fetch.EstimateCard(
    $\max(0$, target - $t)$, preds $\cup$ {this.resolution_function})
card $\leftarrow$ 0
**for** $i$ = 0 **to** $2^{\text{len(preds)}}-1$ **do**
    card $\leftarrow$ card + fetch.cards[$2i+1$]
    this.card $\leftarrow$ this.card + fetch.cards[$2i+1$]
    this.cards[$i$] $\leftarrow$ this.cards[$i$] + fetch.cards[$2i+1$]
**end for**
this.distincts $\leftarrow$ this.distincts + {card, ..., card}
```

The *Resolve* operator first computes card, cards and distincts based on the existing data in the resolved raw tables, without considering parameter target. Our current approach is to exploit the available statistics provided by the back-end RDBMS. Once card, cards and distincts are computed based on the existing data, the *Resolve* operator calls EstimateCard procedure on its child *Fetch* operator to estimate the cardinality of required new data. Since existing data contributes to $t =$ cards[$2^{\text{len(preds)}}-1$] output tuples satisfying all predicates in preds, parameter target is set to $\max(0$, target$-t)$. Also, the *Resolve* operator passes the selectivity of its resolution function as part of parameter preds.

```
*Fetch*.EstimateCard(target, preds)
card $\leftarrow$ target
**for each** pred $\in$ preds **do**
    **if** pred.left $\notin$ this.lhs_attrs **then**
        card $\leftarrow$ card / pred.selectivity
    **end if**
**end for**
**if** this.card < card **then**
    this.card $\leftarrow$ card
    this.cards $\leftarrow$ {card, ..., card}
    **for** $i$ = 0 **to** $2^{\text{len(preds)}}-1$ **do**
        **for** $j$ = 0 **to** len(preds)$-1$ **do**
            **if** preds[$j$].left $\in$ this.lhs_attrs **then**
                selectivity $\leftarrow$ 1.0
            **else**
                selectivity $\leftarrow$ preds[$j$].selectivity
            **end if**
            **if** $i$ & $2^{\text{len(preds)}-1-j}$ == 0 **then**
                this.cards[$i$] $\leftarrow$ this.cards[$i$] $\times$ (1$-$selectivity)
            **else**
                this.cards[$i$] $\leftarrow$ this.cards[$i$] $\times$ selectivity
            **end if**
        **end for**
    **end for**
**end if**
```

As a base case of our recursive process, the *Fetch* operator estimates its cardinality based on its associated fetch rule and the parameters target and preds. We assume that new data obtained by the *Fetch* operator always satisfies those predicates in preds that are used to instantiate the left-hand side of the fetch rule. (For example, in Figure 1b, new countries obtained by *Fetch* operator 8a satisfy predicate language='Spanish'.) Moreover, we assume the other predicates in preds are independent: the probability of a new tuple satisfying a predicate p is the selectivity of p.

Unlike all other operator types, a *Fetch* operator may have more than one parent operator, when the right-hand side of its fetch rule contains dependent attributes spanning multiple raw tables and no anchor attributes. (*Fetch* operator 11a in Figure 1c is one such example.) In this case, the EstimateCard procedure is called on the *Fetch* operator as many times as the number of its parent *Resolve* operators. Since the distribution of new data remains the same, the *Fetch* operator simply retains the maximum estimated cardinality across all calls.

```
*DepJoin*.EstimateCard(target, preds)
outer.EstimateCard(target, preds $\cup$ {this.pred})
inner.EstimateCard(outer.distincts[this.pred.left], $\varnothing$)
this.card $\leftarrow$ 0
this.cards $\leftarrow$ {0, ..., 0}
**for** $i$ = 0 **to** $2^{\text{len(preds)}}-1$ **do**
    this.card $\leftarrow$ this.card + outer.cards[$2i+1$]
    this.cards[$i$] $\leftarrow$ outer.cards[$2i+1$]
**end for**
this.distincts $\leftarrow$ $\min_{\text{elementwise}}$(
    outer.distincts $\cup$ inner.distincts, {this.card, ..., this.card})
```

The *DepJoin* operator [20] is another Deco-specific query operator that supports joining conceptual relations with predicates explicitly specified in Where clauses. (Note the example plans in Figure 1 do not have *DepJoin* operators.) As a variant of traditional dependent join, *DepJoin* sends join values extracted from outer tuples to its inner child to receive matching inner tuples. However, unlike in *DLOJoin*, the inner tuples returned do not necessarily match outer tuples and constitute join result tuples, because we cannot guarantee the results of fetch rules and resolution functions that may feed join predicates.

Roughly, our implementation of EstimateCard for *DepJoin* combines EstimateCard of *DLOJoin* and *Filter*. One simplifying assumption we make is that new outer tuples produced by the crowd do not match existing inner tuples. This assumption will hold if queries typically join the same combinations of conceptual relations, since past queries will tend to have materialized either both or neither of a joining pair of tuples. Of course this assumption may not always hold. For example, if the inner relation is queried more frequently than the outer, the assumption can be violated to a large degree, resulting in large error. Note in the case where we have a bound on the number of distinct values for a join attribute, we could potentially remove this assumption.

```
MinTuples-1.EstimateCard(0, ∅)
    DLOJoin-3.EstimateCard(8, ∅)
        Filter-4.EstimateCard(8, ∅)
            DLOJoin-5.EstimateCard(8, {Spanish})
                Resolve-6.EstimateCard(8, {Spanish})
                    Fetch-8.EstimateCard(5.9, {Spanish, dupElim})
                        └ card=59, cards={0, 53.1, 0, 5.9}
                    └ card=63, cards={55, 8}
                Resolve-9.EstimateCard(63, ∅)
                    Fetch-11.EstimateCard(60, {majority-of-3})
                        └ card=150, cards={90, 60}
                    └ card=63
                └ card=63, cards={55, 8}
            └ card=8
        Resolve-12.EstimateCard(8, ∅)
            Fetch-14.EstimateCard(7, {majority-of-3})
                └ card=17.5, cards={10.5, 7}
            └ card=8
    └
└
```

(a) Basic plan

```
MinTuples-1.EstimateCard(0, ∅)
    DLOJoin-3.EstimateCard(8, ∅)
        Filter-4.EstimateCard(8, ∅)
            DLOJoin-5.EstimateCard(8, {Spanish})
                Resolve-6.EstimateCard(8, {Spanish})
                    Fetch-8a.EstimateCard(5.9, {Spanish, dupElim})
                        └ card=5.9, cards={0, 0, 0, 5.9}
                    └ card=9.9, cards={1.9, 8}
                Resolve-9.EstimateCard(9.9, ∅)
                    Fetch-11.EstimateCard(6.9, {majority-of-3})
                        └ card=17.25, cards={10.35, 6.9}
                    └ card=9.9
                └ card=9.9, cards={1.9, 8}
            └ card=8
        Resolve-12.EstimateCard(8, ∅)
            Fetch-14.EstimateCard(7, {majority-of-3})
                └ card=17.5, cards={10.5, 7}
            └ card=8
    └
└
```

(b) Reverse plan

Figure 2: Trace of EstimateCard

3.3 Cost Estimation Examples

Let us walk through two simple example runs of our cost estimation algorithm. Suppose we execute the example query in Section 2.2, starting with the following resolved raw tables:

CountryA

country
Chile
Korea
Peru
Spain

CountryD1

country	language
Korea	Korean
Peru	Spanish
Spain	Spanish

CountryD2

country	capital
Korea	Seoul
Spain	Madrid

We assume the selectivity factor of predicate language='Spanish' is 0.1. Also, we assume the selectivity factors of resolution functions *dupElim* and *majority-of-3* are 1.0 and 0.4, respectively. Finally we assume each fetch costs \$0.05, for all fetch rules.

Basic Plan: Figure 2(a) shows a trace of our cardinality estimation algorithm for the query plan in Figure 1a. Starting with calling EstimateCard(0, ∅) on the *MinTuples* operator 1, we recursively call EstimateCard until we reach *Resolve* operator 6. At this point, target is eight, and preds has predicate language='Spanish'. Considering existing data in resolved CountryA and CountryD1, there are two tuples satisfying the predicate (Peru and Spain), one tuple not satisfying the predicate (Korea), and one tuple unknown (Chile). Since the unknown tuple has 10% chance of satisfying the predicate, we have card $= 4$ and cards $= \{1.9, 2.1\}$. To produce eight tuples satisfying the predicate, we need $8 - 2.1 = 5.9$ more tuples, so *Resolve* operator 6 calls EstimateCard(5.9, {language='Spanish', *dupElim*}) on *Fetch* operator 8. Since the fetch rule of operator 8 is ∅ ⇒ country, we have card $= 59$ and cards $= \{0, 53.1, 0, 5.9\}$ at *Fetch* operator 8.

As recursion unwinds, *Resolve* operator 6 and *DLOJoin* operator 5 have estimated cardinality of 63. Since resolved CountryD1 has three existing tuples, *Resolve* operator 9 needs to produce $63 - 3 = 60$ more tuples, and *Fetch* operator 11 has estimated cardinality of $60/0.4 = 150$. Similarly, *Filter* operator 4 has estimated cardinality of eight, and *Resolve* operator 12 needs to produce $8 - 1 = 7$ more tuples. (Note only the Spain tuple in resolved CountryD2 is "relevant" at this point.) Thus, *Fetch* operator 14 has estimated cardinality of $7/0.4 = 17.5$. The final estimated monetary cost is $\$0.05 \times (59 + 150 + 17.5) = \$11.325$.

Reverse Plan: Figure 2(b) shows a trace of our cardinality estimation algorithm for the same query but using the plan in Figure 1b. Notice that the trace is exactly same until we reach *Fetch* operator 8a with target of 5.9 and preds being {language='Spanish', *dupElim*}. Since the fetch rule of operator 8a is language ⇒ country, we assume all new tuples satisfy predicate language='Spanish'. Thus, we have card $= 5.9$ and cards $= \{0, 0, 0, 5.9\}$ at *Fetch* operator 8a. As recursion unwinds, *Resolve* operator 6 and *DLOJoin* operator 5 have estimated cardinality of 9.9. Thus *Resolve* operator 9 needs to produce $9.9 - 3 = 6.9$ more tuples, and *Fetch* operator 11 has estimated cardinality of $6.9/0.4 = 17.25$. The rest of the trace is the same as above, and the estimated monetary cost is $\$0.05 \times (5.9 + 17.25 + 17.5) = \$2.0325$.

Our cost estimation indicates that the basic plan is expected to be about six times as expensive as the reverse plan for this setting.

4. SEARCH SPACE

We describe the search space of alternative plans that Deco's query optimizer considers. Even for Select-Project-Join (SPJ) queries, Deco's Fetch-Resolve-Join semantics enables interesting plan alternatives. Specifically, our plan alternatives are defined by selecting a join tree and a set of fetch rules.

(1) **Join tree**: A join tree is a binary expression tree whose operators (internal nodes) are either cross-products or left outer-joins, and whose operands (leaf nodes) are resolved raw tables. In Section 4.1 we discuss the space of join trees we consider.

(2) **Fetch rules**: For each raw table, a Deco query plan requires one fetch rule assigned to obtain additional tuples for the raw table. Section 4.2 discusses the space of possible fetch rules.

In our setting, we construct a logical query plan based on a selected join tree, and expand the logical plan into a set of physical plans by selecting fetch rules. As it happens, all three plans in Figures 1a–1c correspond to the same join tree; however, different fetch rules were chosen. We will see different join trees in Section 4.1.

In this section we first describe the join trees, then we describe the selection of fetch rules given a join tree. Section 5 addresses how to explore the search space described in this section, and find the best plan using the cost estimation algorithm described in Section 3.

4.1 Join Tree

Given a SPJ query Q over conceptual relations, our first goal is to find all join trees over raw tables that evaluate the cross product of all conceptual relations in the From clause of the query Q. Based on a join tree, we construct a complete logical plan by placing the query's join and local predicates. Because we aim to minimize monetary cost, the only sensible logical plan based on a join tree is the one that evaluates all predicates as early as possible. Thus, once we have all join trees for Q, it is straightforward to construct all equivalent logical plans for Q in our search space.

Let us first consider a Deco query containing only one conceptual relation R in its From clause. Here we are interested in finding all join trees equivalent to the following left-deep tree:

$$R = ((A \rtimes D_1) \rtimes D_2) \rtimes \cdots \rtimes D_k$$

where A and $D_1, \ldots, D_k$ denote the resolved anchor table and dependent tables for R, respectively. Although left outerjoins do not commute in general, we can reorder them using the following equation when neither outerjoin predicates are between Y and Z [22]:

$$(X \rtimes Y) \rtimes Z = (X \rtimes Z) \rtimes Y \qquad (2)$$

By repeatedly applying Equation (2) to the left-deep join tree above, we obtain all $k!$ left-deep join trees with A as the first (left-most) operand and $D_1, \ldots, D_k$ in any order after that.

Now consider the general case where a query Q contains n conceptual relations $R_1, \ldots, R_n$ in its From clause. The goal is to find all join trees equivalent to the following default tree:

$$R_1 \times \cdots \times R_n = (((A_1 \rtimes D_{11}) \rtimes D_{12}) \rtimes \cdots \rtimes D_{1k_1}) \times \cdots \times (((A_n \rtimes D_{n1}) \rtimes D_{n2}) \rtimes \cdots \rtimes D_{nk_n}) \qquad (3)$$

where A_i and $D_{i1}, \ldots, D_{ik_i}$ ($k_i \geq 0$) denote the resolved anchor table and dependent tables for R_i, respectively. In addition to Equation (2), the following equation holds when the outerjoin predicate is between Y and Z [22]:

$$X \times (Y \rtimes Z) = (X \times Y) \rtimes Z \qquad (4)$$

This equation allows us to "interleave" resolved raw tables from different conceptual relations in a join tree, opening up more opportunities to push down predicates (thus reducing monetary cost).

Finally we can use the commutativity and associativity of cross-products:

$$X \times Y = Y \times X, \quad X \times (Y \times Z) = (X \times Y) \times Z \qquad (5)$$

Now we are interested in every join tree derived by repeatedly applying Equations (2), (4), or (5) to the default join tree. We obtain a space of join trees satisfying the following properties:

- All raw tables appear in the leaf nodes exactly once.
- Each outerjoin node has some dependent table D_{ij} as its right child and the corresponding anchor table A_i in its left subtree.
- Each dependent table D_{ij} is the right child of some outerjoin node.

We prove in our extended technical report [21] that the space of join trees we obtain satisfies the properties above, and furthermore any join tree satisfying these properties is equivalent the default tree. Thus, these properties exactly characterize our valid join trees.

So far we defined the entire space of join trees that are equivalent to the default tree. To reduce the search space somewhat, we heuristically prune some join trees when a query contains three or more conceptual relations in its From clause. Specifically, we prohibit a cross-product node from having another cross-product in its right subtree. This restriction is analogous to considering left-deep join trees only in traditional optimizers, and makes sense in our setting of dependent joins.

4.2 Fetch Rule Selection

Given a logical plan corresponding to a selected join tree, the next step is to create a set of physical plans. Currently there is a one-to-one mapping between logical and physical operators for all components of any plan, except the leaves. Thus, translating a logical plan into a physical plan amounts to selecting a fetch rule to use for each raw table.

To select fetch rules for a particular logical plan, we first compute a set of candidate fetch rules for each raw table, described shortly. Selection of one fetch rule each from all candidate sets yields a physical plan. Depending on the logical plan and set of available fetch rules, some raw tables may have an empty set of candidate fetch rules; in this case there are no physical plans corresponding to the logical plan, so we discard the logical plan. (If every logical plan is discarded, the database does not include enough fetch rules to execute the query.)

Let us consider a *Fetch* operator F that is a child of *Resolve* operator R, whose *Scan* child is associated with raw table T. (See examples in the query plans in Figures 1a–1c.) Now consider the set of candidate fetch rules for *Fetch* operator F. In Deco's query execution engine, *Resolve* operator R requests tuples from *Fetch* operator F to feed its resolution function. To make this mechanism work, the fetch rule $A_1 \Rightarrow A_2$ deployed in *Fetch* operator F must satisfy the following conditions:

- **Condition 1:** Each attribute in A_1 is instantiated by either a value from an outer tuple passed to *Fetch* operator F by a dependent join operator, or a constant value in a local Where clause predicate.
- **Condition 2:** The attributes in $A_1 \cup A_2$ cover all attributes in raw table T, and some attributes in A_2 are attributes of T.

The first condition allows *Fetch* operator F to invoke the fetch rule by instantiating all attributes in A_1. The second condition allows new tuples obtained using the fetch rule to be inserted into T and passed up to *Resolve* operator R.

Note the first condition is specific to the logical plan, while the second one is not. Thus one possible algorithm to compute a candidate set would first find the available fetch rules satisfying the second condition, then discard those violating the first condition.

As an example, our original Figures 1a–1c show three different physical plans expanded from the same logical plan. It can be easily verified (and is intuitive) that the two conditions are satisfied for the selected fetch rules. The search space for the example query includes many more physical plans deploying different combinations of fetch rules.

5. ENUMERATION ALGORITHM

We now consider how to efficiently enumerate query plans in the search space and apply the cost estimation algorithm to find the predicted best query plan. As briefly discussed in Section 1, Deco's specific setting invalidates some key assumptions behind enumeration in traditional query optimizers:

- In Deco, different physical plans corresponding to the same logical plan may produce different query results: the fetch rules selected in a physical plan determine the valid instance over which the query result is evaluated. Thus, estimated cardinality is a property of physical plans rather than logical plans, violating a typical assumption exploited by extensible query optimizers such as Volcano [13] and Cascades [12].
- Deco's cardinality estimation is holistic: the cardinality and cost of a subplan depend in part on the rest of the plan. Thus, a bottom-up enumeration strategy as in the System R optimizer [23] is not applicable to Deco.

Given these constraints, our goal is to devise a plan enumeration algorithm for Deco that generates complete physical plans in the search space while maximizing reuse of common subplans. This strategy is expected to make Deco's query optimizer slower than

```
FindBestPlanNaive(ast)
 1  bestPlan ← NULL
 2  minCost ← ∞
 3  for each joinTree do
 4      for each fetchRuleSelection do
 5          plan ← BuildPlan(ast, joinTree, fetchRuleSelection)
 6          if plan.isValid() then
 7              plan.EstimateCard(0, ∅)
 8              cost ← plan.EstimateCost()
 9              if cost < minCost then
10                  bestPlan ← plan
11              end if
12          end if
13      end for
14  end for
15  return bestPlan
```

Figure 3: Naive plan enumeration algorithm

traditional query optimizers for queries with similar complexity; however, we will see in Section 6 that the optimization time tends to be insignificant compared to typical query execution time in Deco.

In this section we first describe a naive exhaustive enumeration, then we describe a more efficient version of the same strategy. In Section 6.2 we compare the performance of the more efficient enumeration against the naive one.

5.1 Naive Enumeration

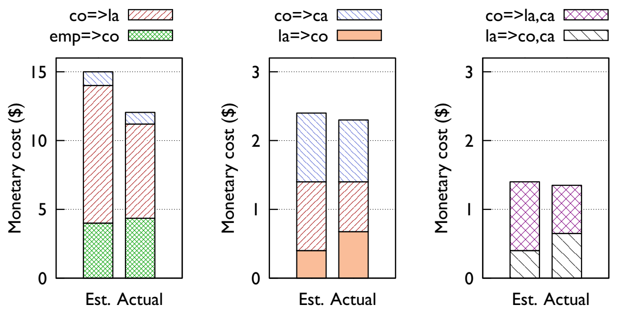
In its simplest form, Deco's plan enumeration algorithm generates all query plans in the search space and applies the cost estimation algorithm to each plan to find the predicted best plan. The simple FindBestPlanNaive procedure in Figure 3 illustrates the entire query optimization process at a high level.

To enumerate all possible join trees (line 3), we use a straightforward recursive algorithm based on the properties of join trees from Section 4.1. Given a join tree, we build a plan with each possible fetch rule selection satisfying Condition 2 in Section 4.2 (lines 4–5). In this step, The BuildPlan procedure places the query's predicates so that they are evaluated as early as possible. Finally, we retain those plans satisfying Condition 1 in Section 4.2 (line 6), and choose the plan with the least estimated cost (lines 7–11).

5.2 Efficient Enumeration

The FindBestPlanNaive procedure in Figure 3 handles each alternative plan independently, so it may perform some redundant computation across iterations of the inner loop (lines 5–12). Specifically, the EstimateCard procedure in Section 3.2 may be called multiple times on the same subplans with the same arguments for target and preds. Since those calls produce the same result (card and cards), we "memo-ize" [13]: the first call computes and stores the result, and subsequent calls reuse it. (Even if the order of predicates in parameter preds in a subsequent call is different from the original order of predicates used to compute the stored result, we can reuse the result by shuffling cards appropriately; recall the definitions of preds and cards in Section 3.2.)

It turns out that we can maximize reuse of common subplans by storing only one plan at a time (with its cardinality estimates), if we iterate over the alternative plans in a particular order. This order is determined based on two properties. First, given a left (outer) subplan S, all complete plans containing the subplan S must appear consecutively in the order, so that we can reuse the subplan S and its estimated cardinality. Moreover, we further order those plans containing the subplan S by the first raw table T joining with S



(a) Basic plan **(b) Reverse plan** **(c) Hybrid plan**

Figure 4: Accuracy of cost estimation: no existing data

so that the *Resolve* operator corresponding to T can reuse the estimated cardinality of the relevant existing data. Note that these properties apply recursively to different layers of subplans. As a result, we enumerate physical plans by alternately selecting raw tables and their corresponding fetch rules. Note in contrast the naive enumeration first selects a join tree and then selects a fetch rule for each raw table in the tree.

6. EXPERIMENTAL EVALUATION

In this section we present our experimental evaluation of Deco's query optimizer. We first evaluate the accuracy of our cost estimation algorithm in a variety of settings. Then, we evaluate the efficiency of our plan enumeration algorithm.

For all experiments, we used an Intel Core i5 laptop running Mac OS X, but any commodity machine should produce similar results. Currently Deco uses Python 2.7 and PostgreSQL 9.2.

6.1 Accuracy of Cost Estimation

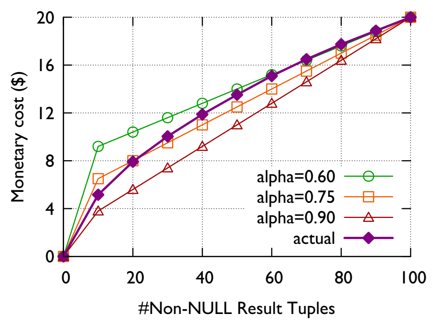
To evaluate the accuracy of Deco's cost model, we compare the estimated costs against the actual costs for three different scenarios: no existing data in the raw tables (Experiment 1), existing data with fetch prioritization (Experiment 2), and existing data with little effect of fetch prioritization (Experiment 3).

Experiment 1: No Existing Data In [18], we reported the monetary cost and latency of executing the example query in Section 2.2 using Mechanical Turk, for the following query plans:

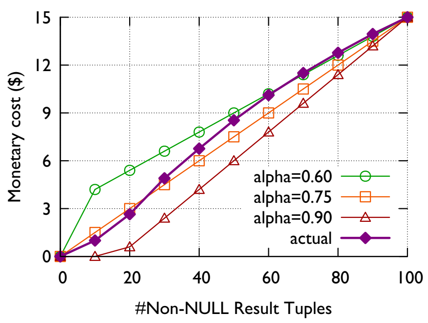
- Basic plan (Figure 1a): $\emptyset \Rightarrow$ country, country $\Rightarrow$ language, and country $\Rightarrow$ capital
- Reverse plan (Figure 1b): language $\Rightarrow$ country, country $\Rightarrow$ language, and country $\Rightarrow$ capital
- Hybrid plan (Figure 1c): language $\Rightarrow$ country,capital and country $\Rightarrow$ language,capital

The cost of each fetch was set to \$0.05 for all fetch rules. Starting with empty raw tables, the actual costs of the example query were \$12.05, \$2.30, and \$1.35 for the basic, reverse, and hybrid plans, respectively. For details see [18].

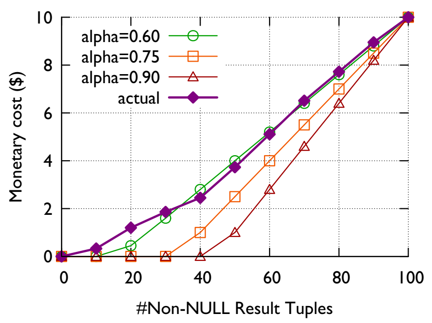
For cost estimation, we set the selectivity factor of predicate language='Spanish' to 0.1. Also, we set the selectivity factors of resolution functions *dupElim* and *majority-of-3* to 1.0 and 0.4, respectively. Given these settings, Deco's cost estimation algorithm produces estimated costs of \$15.00, \$2.40, and \$1.40 for the basic, reverse, and hybrid plans, respectively. Figure 4 shows the estimated and actual costs for the three query plans. The different portions of the bars show the costs incurred by the different fetch rules in each setting. Even though our overall estimated costs were reasonably close to the actual costs with a mean percentage error of 11%, estimated costs for individual *Fetch* operators were less accurate. We now explain why.



(a) Empty dependent tables

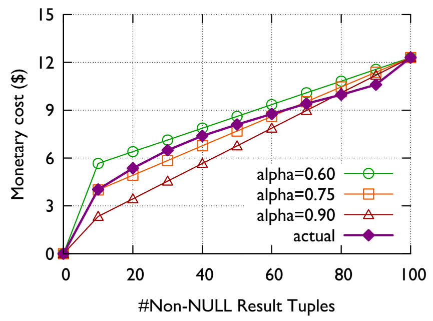


(b) 100 tuples in dependent tables

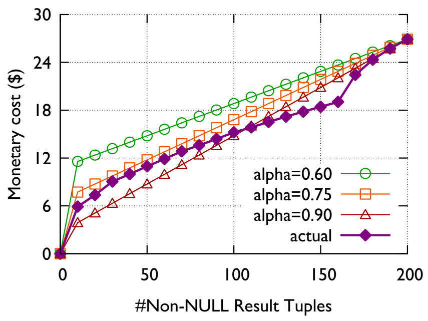


(c) 200 tuples in dependent tables

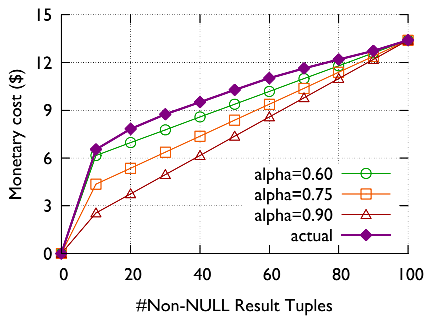
Figure 5: Accuracy of cost estimation: existing data, fetch prioritization (part 1)



(a) 100 European cities



(b) 200 World cities



(c) 100 Synthetic cities (worst case)

Figure 6: Accuracy of cost estimation: existing data, fetch prioritization (part 2)

First, our selectivity factor settings were not accurate enough. Using the basic plan, for example, resolution function *majority-of-3* did not need the third input value as often as expected, and we observed an actual selectivity factor of 0.47. Also, we ended up collecting 64 distinct countries with 87 answers, translating to actual selectivity factors of 0.74 and 0.125 for resolution function *dupElim* and predicate language=‘Spanish’, respectively. In general, selectivity factors for the different operators in a particular plan have different impact on the accuracy of estimated cost. As an example, the cost of the reverse plan is most sensitive to the selectivity of *majority-of-3*: a difference of 0.1 in the selectivity translates to about 20% error in the estimated cost. We ran experiments (not reported in detail due to space constraints) to explore how selectivity errors affect estimated cost. In most cases, the selectivity error needs to be quite high before a poorer plan is selected.

Second, our cardinality estimation algorithm makes some simplifying assumptions when handling “reverse” fetch rules (i.e., fetch rules obtaining anchor values given some dependent values). For example, we assume new countries obtained using language $\Rightarrow$ country satisfy predicate language=‘Spanish’, but in reality workers may return values that do not satisfy this predicate. We could remove this assumption from the *Fetch*.EstimateCard procedure, if we are given a “selectivity factor” for the fetch rule. In addition, a reverse fetch rule inserts new tuples into some dependent tables, thus effectively decreasing the additional number of required values to resolve the dependent attributes. To address this problem, we could adjust the selectivity factors of those resolution functions that have interactions with reverse fetch rules.

Experiment 2: Existing Data, Fetch Prioritization For this experiment and the next one, we used our *crowd simulator*, which enables a large number of experiments without significant latency and dollar cost. (Note that repeating these experiments on a real crowdsourcing platform would either be extremely costly, or would mean far fewer trials.) The simulator responds to fetch requests by selecting values from a predetermined set; we can either set our simulator to always give “correct” answers, or specify a fixed probability for each fetch rule that “incorrect” answers are given. We first built the simulator for the experiments in [20], which also give us the actual costs used here.

For cost estimation, recall from Section 3.2 that our cost model includes a configurable weight α to emulate the effectiveness of fetch prioritization, with a larger α indicating better prioritization. For each experiment, we computed the estimated costs using α values of 0.6, 0.75, and 0.9, to empirically determine a good value for α. In addition, we assume that statistics about the existing data available in the back-end RDBMS are accurate, so that we can evaluate Deco’s cost model in isolation.

Figure 5 shows the estimated and actual costs of obtaining X result tuples for the following simple query. We vary MinTuples X on the x-axis.

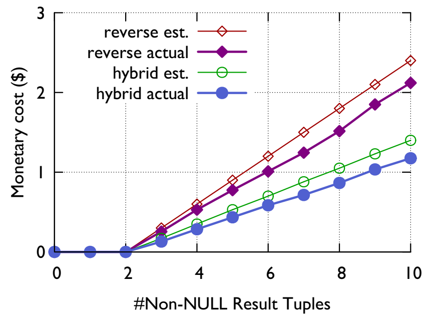
```
SELECT country, language, capital  FROM Country
MINTUPLES X
```

As in [20], we seed anchor table CountryA with 100 different country names, and the two dependent tables with 0, 100, and 200 randomly chosen tuples (across both tables) for Figures 5a, 5b, and 5c, respectively. The query plan uses two fetch rules, country $\Rightarrow$ language and country $\Rightarrow$ capital, to produce resolved language and capital values. Overall, our cost estimates are reasonably close to the actual costs: With α=0.75, we observed mean absolute percentage errors of 6.7%, 12.4%, and 44.4%, for Figures 5a, 5b, and 5c, respectively. Note the large error for Figure 5c is mainly due to the 100% error for the $X \leq 30$ data points.

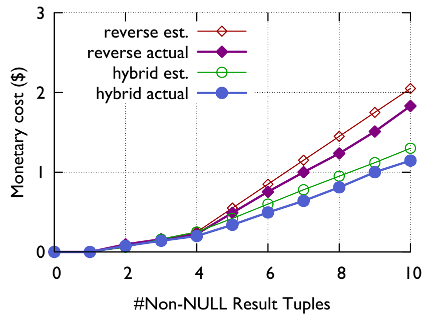
Figure 6 shows the estimated and actual costs of obtaining X result tuples for the following join query.

```
SELECT city, country, population, language
FROM Country, City
WHERE City.country = Country.country  MINTUPLES X
```

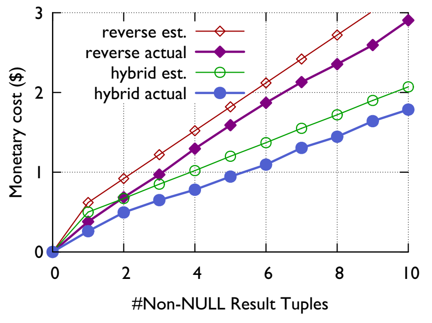
Anchor tables CountryA and CityA are populated from real datasets for Figures 6a and 6b, and a synthetic dataset for Figure 6c to



(a) Existing data state 1

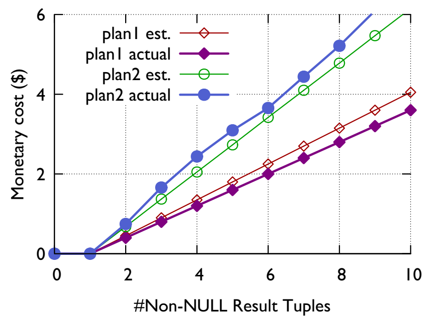


(b) Existing data state 2

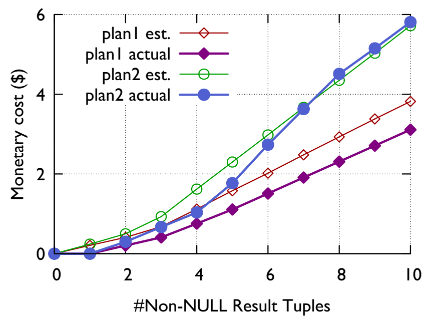


(c) Existing data state 3

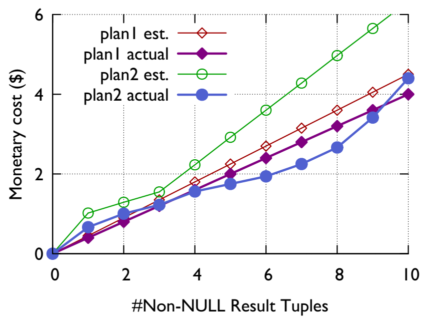
Figure 7: Accuracy of cost estimation: existing data, no fetch prioritization (part 1)



(a) Existing data state 1



(b) Existing data state 2



(c) Existing data state 3

Figure 8: Accuracy of cost estimation: existing data, no fetch prioritization (part 2)

demonstrate the worst case. All dependent tables are initially empty and populated on-demand using fetch rule country $\Rightarrow$ language for relation Country and fetch rule city,country $\Rightarrow$ population for relation City. In Figures 6a and 6b, our cost estimates are reasonably accurate as in Figure 5: With α=0.75, the mean absolute percentage errors are 4.8% and 11.1% for Figures 6a and 6b, respectively. In Figure 6c, we deliberately generated a synthetic dataset to make fetch prioritization work as poorly as possible. (Details were given in [20].) Hence, the estimated costs (even with α of 0.6) are smaller than the actual costs for the entire range of X. This result is consistent with the definition of α in Section 3.2: a smaller α means less effective prioritization.

Based on our experiments, we believe by setting α=0.75 as a rule of thumb, our cost model will produce acceptable estimates for many cases. Note that the best value for α depends on the heuristic approach for fetch prioritization; we would need to use a larger α for some more sophisticated heuristics described in [20].

Experiment 3: Existing Data, No Fetch Prioritization Now we consider a similar scenario but without fetch prioritization. Figure 7 shows the estimated and actual costs of obtaining X result tuples for the example query from Section 2.2. Again, we vary MinTuples X on the x-axis.

```
SELECT country, capital  FROM Country
WHERE language='Spanish'  MINTUPLES X
```

We start the query with three different initial states of the raw tables, resulting in the three graphs in Figure 7. For each initial state, we measured the actual costs of executing the reverse plan (Figure 1b) and the hybrid plan (Figure 1c), with the crowd simulator set to correspond to the selectivities observed using Mechanical Turk in Experiment 1. (Note the simulator flips a coin to produce data, and we report the average of ten trials.) For cost estimation, we use the same selectivity setting as in Experiment 1. Overall, our estimated costs were reasonably accurate across all three initial states and both plans, with a mean absolute percentage error of 18.6%. This result is comparable to Experiment 1 (no existing data) and implies that our cost model is able to distinguish between existing data versus new data.

Figure 8 shows the estimated and actual costs of obtaining X result tuples for the following query.

```
SELECT capital, population, language
FROM Country, City
WHERE City.country = Country.country AND
    City.city = Country.capital
MINTUPLES X
```

Again we start the query with three different initial states of the raw tables, resulting in the three graphs in Figure 8. For each initial state, we execute the following query plans:

- Plan 1: Join tree (((CountryA $\bowtie$ CountryD2) $\bowtie$ CityA) $\times$ CountryD1) $\bowtie$ CityD1, with "basic" fetch rules
- Plan 2: Join tree (((CityA $\times$ CountryA) $\bowtie$ CountryD2) $\bowtie$ CountryD1) $\bowtie$ CityD1, with "basic" fetch rules

Note despite the simplicity of the original query, these plans are 5-way joins over the raw tables. For Plan 2, we set the crowd simulator to produce capital cities with a probability of 0.3 (for fetch rule $\varnothing \Rightarrow$ city,country), and also used the corresponding selectivity setting for cost estimation. In Figures 8a and 8b, our cost estimates are reasonably close to the actual costs with a mean absolute percentage error of 20.3%. Figure 8c illustrates a case where our cost model often fails to predict the better plan: We populated relation City only with capital cities, resulting in large discrepancy between the actual selectivity of join predicate City.city=Country.capital and the provided selectivity setting.

6.2 Efficiency of Plan Enumeration

To evaluate the efficiency of plan enumeration, we compare the efficient enumeration from Section 5.2 against the naive enumeration from Section 5.1, in terms of the overall optimization time, i.e., time taken to find the predicted best plan given a parsed query.

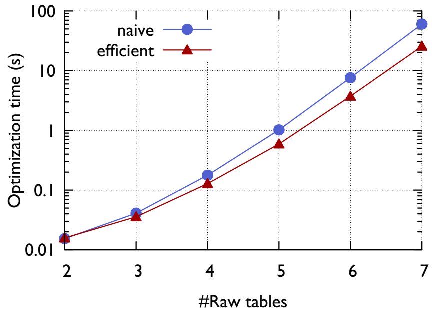


Figure 9: Efficiency of plan enumeration

Experiment 4: Varying Number of Raw Tables Because the size of the search space heavily depends on the number of raw tables m, we generated queries with a varying number of raw tables based on their From clauses. For each query, we created a set of fetch rules so that each raw table has exactly one candidate fetch rule. (Here we use a conservative setting: more fetch rules make the efficient enumeration even faster than the naive one.) Figure 9 shows the optimization times for the naive and efficient enumeration in logarithmic scale, for m=2..7. Not surprisingly the efficient enumeration performs better than the naive enumeration for the entire range of m values. Moreover, the percent improvement tends to increase as m increases, because the amount of redundant computation also increases. With m=7, the efficient enumeration is 2.35 times faster than the naive enumeration.

7. RELATED WORK

Several recent data-oriented systems have used a declarative approach to leverage crowdsourced data [1, 7, 10, 16, 17]. Among those systems, CrowdDB [10] bears the closest similarity to Deco in terms of the data model and query language; however, Deco opts for more generality and flexibility, thus requiring the novel query execution techniques described in [20]. (A detailed comparison between the two systems can be found in [18].) Qurk [16] is a workflow engine that uses crowdsourcing primarily as part of its operators, and reference [16] studied how to reduce the monetary cost of its crowd-powered sort and join operators by improving worker interfaces. Since Deco makes no assumption about worker interfaces, their work is complimentary to Deco and can be incorporated into Deco to improve fetch procedures. To the best of our knowledge, CrowdDB and the other systems in this category do not yet have a cost-based query optimizer.

There has been a large body of previous work addressing query optimization in traditional systems [3]. Sections 3 and 5 elaborated on several key differences between traditional query optimizers and Deco's query optimizer in plan costing and enumeration.

Also related is prior work on query optimization over diverse data sources in the context of heterogeneous or federated database systems [6, 9, 14]. In some sense, Deco's overall architecture [19] is analogous to federated database systems: Deco's query processor, fetch procedures, and the crowd correspond to a mediator, wrappers, and data sources, respectively. However, as far as the query optimization problem is concerned, all of the fundamental differences between Deco and traditional database systems that we have described earlier also apply when comparing Deco and federated database systems.

8. CONCLUSIONS AND FUTURE WORK

We presented Deco's query optimizer that finds the best plan to answer a query in terms of estimated monetary cost. We incorporated several novel techniques into the query optimizer to reflect Deco's query semantics and plan execution strategies. Coupled with Deco's query execution engine that executes the chosen plan with least monetary cost, Deco's query processor as a whole provides a complete solution for answering a Deco query while minimizing monetary cost.

For future work, we would like to incorporate alternatives to MinTuples, such as MaxCost and MaxTime, enabling end-users to specify a monetary or time budget to answer a query, while maximizing the number of result tuples. We are also interested in incorporating adaptive query processing techniques into Deco.

9. REFERENCES

[1] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *WWW*, pages 1009–1018, 2012.

[2] M. J. Carey and D. Kossmann. On saying ”enough already!” in sql. In *SIGMOD*, pages 219–230, 1997.

[3] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[4] S. Chaudhuri, H. Lee, and V. R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, pages 531–542, 2010.

[5] S. Chaudhuri and V. R. Narasayya. Automating statistics management for query optimizers. In *ICDE*, pages 339–348, 2000.

[6] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, pages 716–727, 2002.

[7] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Declarative platform for data sourcing games. In *WWW*, pages 779–788, 2012.

[8] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4):86–96, 2011.

[9] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in a heterogeneous dbms. In *VLDB*, pages 277–291, 1992.

[10] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[11] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *SIGMOD*, pages 285–296, 2000.

[12] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[13] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[14] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, pages 276–285, 1997.

[15] P. G. Ipeirotis. Analyzing the amazon mechanical turk marketplace. *ACM Crossroads*, 17(2):16–21, 2010.

[16] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

[17] A. Morishima, N. Shinagawa, T. Mitsuishi, H. Aoki, and S. Fukusumi. CyLog/Crowd4U: A declarative platform for complex data-centric crowdsourcing. *PVLDB*, 5(12):1918–1921, 2012.

[18] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: Declarative crowdsourcing. In *CIKM*, pages 1203–1212, 2012.

[19] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.

[20] H. Park, A. Parameswaran, and J. Widom. Query processing over crowdsourced data, http://ilpubs.stanford.edu:8090/1052/. Technical report, Stanford InfoLab, 2012.

[21] H. Park and J. Widom. Query optimization over crowdsourced data, http://ilpubs.stanford.edu:8090/1063/. Technical report, Stanford InfoLab, 2012.

[22] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD*, pages 291–299, 1990.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.