ABSTRACT

We are building Deco, a comprehensive system for answering declarative queries posed over stored relational data together with data gathered from the crowd. In this paper we present Deco’s query processor, building on Deco’s data model and query language presented earlier. In general, it has been observed that query processing over crowdsourced data must contend with issues and tradeoffs involving cost, latency, and uncertainty that don’t arise in traditional query processing. Deco’s overall objective in query execution is to maximize parallelism while fetching data from the crowd (to keep latency low), but only when the parallelism will not issue too many tasks (which would increase cost). Meeting this objective requires a number of changes from traditional query execution. First, Deco’s query processor uses a hybrid execution model, which respects Deco semantics while enabling our objective. Our objective also requires prioritizing accesses to crowdsourced data, which turns out to be an interesting NP-hard problem. Finally, because Deco incorporates resolution functions to handle the uncertainty in crowdsourced data, query execution bears as much similarity to incremental view maintenance as to a traditional iterator model. The paper includes initial experimental results, focusing primarily on how our query execution model and access prioritization scheme maximize parallelism without increasing cost.

1. INTRODUCTION

Crowdsourcing [4, 17] provides a programmatic way of incorporating human computation to accomplish a variety of tasks that are difficult for computer algorithms to perform, e.g., evaluating search results and labeling images. Even though crowdsourcing is considered a promising tool for many applications, programmers encounter some common challenges in taking advantage of crowdsourcing in practice: resolving inconsistencies in crowdsourced data, seamlessly integrating crowdsourced data with existing data, and reducing the monetary cost and latency of crowdsourcing.

To address these challenges, we proposed Deco (for “declarative crowdsourcing”) [15, 16], a system that answers declarative queries posed over stored relational data together with data gathered from the crowd. In [15], we defined a data model for Deco that is general (it can be instantiated to other proposed models), flexible (it allows methods for data cleansing and external access to be plugged in), and principled (it has a precisely-defined semantics). We also defined a query language for Deco as a simple extension to SQL with constructs necessary for crowdsourcing, and with a semantics that directly follows the data-model and SQL semantics.

In this paper, we describe Deco’s query processor. While at a high level Deco’s query processor resembles its counterpart in traditional database systems, we had many significant new challenges to address:

- As a data source, the crowd has unique characteristics that are not found in conventional sources (like disks); very high latency and abundant parallelism. Thus, it is important for a crowdsourcing infrastructure (like Deco) to hide the latency and exploit the parallelism by invoking “access methods” in parallel.
- Accessing crowdsourced data incurs monetary cost. However, reducing monetary cost and latency at the same time is not straightforward. It is critical to exploit the right degree of parallelism to optimize for both monetary cost and latency.
- The query execution engine should attempt to answer a given query using existing data before it collects more data through crowdsourcing. This requirement seems trivial, however it is not trivial to implement. All query operators must buffer existing data and combine it correctly with new data.
- Deco’s data model is general and flexible enough to invalidate some key assumptions behind traditional iterator-based query execution, discussed shortly.

In response to these challenges, we set the overall objective for Deco’s query execution as follows: exploit as much parallelism as possible when accessing the crowd (to reduce latency), but only when the parallelism will not waste work (to minimize monetary cost). We designed and implemented a query processor for Deco that respects Deco semantics while meeting this objective.

- Deco’s query processor executes a query plan using a novel hybrid execution model. In this model, each query operator can “asynchronously pull” more tuples from its child operator: The parent operator proceeds without waiting for new tuples, and the child operator can “asynchronously push” new tuples to its parent later. This model helps us meet our objective by precisely adjusting the degree of parallelism.
- To maximize parallelism but not incur extra cost, we use a technique based on prioritization of accesses to crowdsourced data. This prioritization problem is unique to the crowdsourcing setting, and it turns out to be critical to performance when Deco must combine existing data with newly crowdsourced data. The solution to this NP-hard problem should be effective and easily incorporated into the query execution engine. We show experimentally that our heuristic algorithm can achieve a near-optimal prioritization.
- Deco’s query execution bears as much similarity to incremental view maintenance as to a traditional iterator model: Query operators can remove or modify output tuples that were passed to their parent previously. This feature is key to supporting the way Deco handles uncertainty and inconsistencies in crowdsourced data. Moreover, not only do Deco’s “access methods” insert new tuples into a single table during query execution, but...
as a side effect they may provide new tuples spanning multiple tables. Changes based on these new tuples propagate up the plan similar to incremental view maintenance.

In this paper we show some alternative valid plans for a given query, and explain how different plans affect query execution. However, this paper does not include plan costing or enumeration, which is a large and complex challenge on its own.

In summary, our contributions are the following:

- We review Deco’s data model and query language (Section 2), summarizing materials from [15, 16].
- We establish the goals for Deco’s query execution, discuss several challenges, and describe our approach (Section 3).
- We describe how Deco executes queries using its hybrid execution model, for the special case when there is no existing data (Section 4).
- We extend query execution for the general case where existing data must be combined with new data (Section 5).
- We formalize the prioritization problem and describe how Deco prioritizes accesses to crowdsourced data to minimize monetary cost (Section 6).
- We describe how Deco implements prioritization for the special case of Mechanical Turk (Section 6.4).
- We experimentally evaluate Deco’s query execution model and access prioritization scheme in various settings (Section 7).

Related work is covered in Section 8, and we conclude with future directions in Section 9.

2. DATA MODEL AND QUERY LANGUAGE

We begin by reviewing each of the Deco data model components using a running example, then we describe Deco’s query language and semantics. For more details see [15].

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. We will see how they are used below.

As a simple 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. In the Country relation, the anchor attribute country is a country name, while language and capital are properties of the country. In the City relation, the pair of city and country names identifies a city, while population is a property of the city.

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. (Later we will see that the back-end RDBMS is also used for intermediate storage during query execution.) Specifically, 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.

In our example, we have the raw schema:

CountryA(country)
CountryD1(country, language)
CountryD2(country, capital)
CityA(city, country)
CityD1(city, country, population)

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 \( A_1 = \emptyset \) permitted), and \( P \) is a fetch procedure that implements access to human workers. (\( P \) might generate HITs (Human Intelligence Tasks) to Mechanical Turk [1], for example.) 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 \( A_1 \cup A_2 \).

Here are some example fetch rules and their interpretations for our running example.

- \([\text{Country}] \emptyset \Rightarrow \text{country} \): Ask for a country name, inserting the obtained value into raw table CountryA.
- \([\text{Country}] \text{country} \Rightarrow \text{capital} \): Ask for a capital given a country name, inserting the resulting value into table CountryD2.
- \([\text{Country}] \text{language} \Rightarrow \text{country} \): Ask for a country name given a language, inserting the resulting country name into CountryA, and inserting the country-language pair into CountryD1.

There are many more possible fetch rules for our example. For a full description of the allowable fetch rules in Deco, see [15].

The schema designer may add or remove fetch rules at any time during the lifetime of a database—they are more akin to “access methods” than to part of the permanent schema.

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 \( \emptyset \Rightarrow A : f \) for the anchor attributes \( A \) treated as a group, and one resolution rule \( A \Rightarrow D : f \) for each dependent attribute-group \( D \). In \( \emptyset \Rightarrow A : f \), the resolution function \( f \) “cleans” a set of anchor values. In \( A \Rightarrow D : f \), the resolution function \( f \) “cleans” the set of dependent values associated with specific anchor values.

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

- \([\text{Country}] \emptyset \Rightarrow \text{country} : \text{dupElim} \)
- \([\text{Country}] \text{country} \Rightarrow \text{language} : \text{majority-of-3} \)
- \([\text{Country}] \text{country} \Rightarrow \text{capital} : \text{majority-of-3} \)
- \([\text{City}] \emptyset \Rightarrow \text{city, country} : \text{dupElim} \)
- \([\text{City}] \text{city, country} \Rightarrow \text{population} : \text{average-of-2} \)

Resolution function \text{dupElim} produces distinct country values for Country and distinct city-country pairs for City. Resolution function \text{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. On the other hand, sometimes more than three values are input to the function, in which case a majority is needed to produce an output. Resolution function \text{average-of-2} produces the average of two or more population answers for a given city.

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

- \([\text{Country}] \emptyset \Rightarrow \text{country} : \text{dupElim} \)
- \([\text{Country}] \text{language} \Rightarrow \text{country} : \text{majority-of-3} \)
- \([\text{Country}] \text{capital} \Rightarrow \text{country} : \text{majority-of-3} \)
- \([\text{City}] \emptyset \Rightarrow \text{city, country} : \text{dupElim} \)
- \([\text{City}] \text{city, country} \Rightarrow \text{population} : \text{average-of-2} \)

Resolution function \text{dupElim} produces distinct country values for Country and distinct city-country pairs for City. Resolution function \text{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. On the other hand, sometimes more than three values are input to the function, in which case a majority is needed to produce an output. Resolution function \text{average-of-2} produces the average of two or more population answers for a given city.

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 obtained 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 us-
ing resolution rules for each of the raw tables. (3) Outerjoining the resolved raw tables to produce the conceptual relations.

It is critical to understand that the Fetch-Resolve-Join sequence is a logical concept only. When Deco queries are executed, not only may these steps be interleaved, but only those portions of the conceptual data needed to produce the query result are actually materialized.

Query Language and Semantics: A Deco query $Q$ is simply a SQL query specified over the conceptual relations. Deco’s query semantics dictate that the answer to $Q$ must represent the result of evaluating $Q$ over some (logical) valid instance of the database. Because our Fetch-Resolve-Join database semantics is based on outerjoins, conceptual relations may logically include numerous NULL values. Thus, we restrict query results to include only those conceptual tuples whose attribute values are non-NULL, although this restriction could be lifted if desired.

One valid instance of the database can be obtained by resolving and joining the current contents of the raw tables, without invoking any further fetch rules. Thus, it appears a query $Q$ could always be answered correctly without consulting the crowd at all. The problem is that this “correct” query result may be very small, or even empty. To retain our straightforward semantics over valid instances, while still forcing answers to contain some amount of data, we add one of the following constraints to a Deco query:
- “MaxCost $c$” specifies to spend up to $c$ dollars (or other cost unit) to answer the query, while attempting to maximize the number of tuples in the result.
- “MaxTime $t$” specifies to spend up to $t$ seconds (or other time unit) to answer the query, while attempting to maximize the number of tuples in the result.
- “MinTuples $n$” specifies that at least $n$ tuples should be in the result, while attempting to minimize cost (or time).

In this paper we focus on the third type of constraint: producing a minimum number of (non-NULL) result tuples while minimizing cost. This problem alone is quite challenging; as future work we will address the other constraint types.

### 3. QUERY EXECUTION OVERVIEW

Having defined the Deco data model and query language, we now consider how to build a query execution engine that implements the defined semantics while executing queries efficiently. The query execution engine must support constraints such as MinTuples while dealing with the inherent challenges in crowdsourcing, such as monetary cost and latency.

In this section, we first establish the goals and objectives for query execution, then we discuss several challenges in query execution as well as our approach to tackle them. Lastly we walk through execution of an example query plan at a high level.

#### 3.1 Goals

Given a Deco query plan with “MinTuples $n$” constraint, our primary goal in query execution is to produce at least $n$ result tuples while minimizing monetary cost. In addition, our secondary goal is to reduce latency by exploiting parallelism when accessing the crowd. Achieving both the primary and secondary goals translates to the following two fundamental objectives for Deco’s query execution engine:
- **Objective #1**: Never parallelize accesses to the crowd if it might increase the monetary cost.
- **Objective #2**: Always parallelize accesses to the crowd if it cannot increase the monetary cost.

We will see in the next subsections how Deco’s query execution strategy is driven by these objectives.

#### 3.2 Challenges and Approach

Deco’s data model and query semantics along with the objectives defined above give us several unique challenges to address. We describe those challenges and how Deco’s execution model addresses them. Like a traditional query plan, a Deco query plan is a rooted DAG of query operators. Each operator produces “output tuples” based on “input tuples” that are the outputs of its children.

**Executing Queries in Two Phases:** To minimize monetary cost, Deco’s query execution engine has to ensure that fetches are issued only if the raw tables do not have sufficient data to produce $n$ result tuples. To do so, 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 constraint is met.

**Enabling Parallelism Using Asynchronous Pull:** To reduce latency, it is important for Deco’s query execution engine to exploit parallelism when accessing the crowd, as discussed in Section 1. However, the traditional iterator model cannot enable this kind of parallelism because getNext calls do not return until data is provided. (The solution used in parallel database systems does not apply in our setting; see Section 8.) In Deco’s hybrid execution model, query operators do not expect an immediate response to a “pull” request; the child operator will respond whenever a new output tuple becomes available. This built-in asynchrony in query plan execution ultimately allows Deco to ask multiple questions to the crowd in parallel, without having to wait for individual crowd answers. The concept of asynchronous iteration introduced in [7] also aims to enable parallelism when accessing outside sources during query execution. However, our specific setting led us to a somewhat different solution; see Section 8.

**Choosing Right Degree of Parallelism:** To reduce latency while still minimizing monetary cost, it is critical to choose the right degree of parallelism: too much parallelism might waste work (and therefore increase monetary cost), while too little increases latency. As described in Section 3.1, our approach is to maximize the number of fetches being executed in parallel, as long as all of them may be required to produce the query result. For example, to produce $n$ output tuples, a Filter operator starts by “pulling” $n$ input tuples in parallel from its child, then initiates another pull whenever it receives an input tuple not satisfying the predicate. Similarly, a Resolve operator first pulls the minimum number of input tuples required for the resolution function to produce an output value, then pulls additional input tuples as long as the resolution function indicates that more are needed for an output value.

**Initiating Good Fetches:** In certain cases where existing data must be combined with new data to produce the result, minimizing monetary cost is especially difficult because existing data can make some fetches more profitable than others. Thus, Deco’s query execution engine has to choose not only the right degree of parallelism, but also the specific “good” fetches to invoke in parallel. Individual query operators do not always have enough information to choose the good fetches, so our approach is to invoke more fetches than needed, but prioritize them so the better fetches are more likely to complete first (thus minimizing monetary cost). The query engine cancels any outstanding fetches once the MinTuples constraint is
met, which for our model we assume incurs no extra cost. (Note this assumption largely holds, e.g., Mechanical Turk [1].)

**Changing Result Incrementally**: Due to the flexibility of Deco’s fetch rules and resolution rules, query operators sometimes have to remove or modify output tuples that were passed to their parents previously. For example, fetch rules such as country ⇒ language,capital can provide tuples to multiple raw tables. Even if this fetch rule is invoked based on the need for a language value, data may as a side effect be inserted into the raw table capital. This insertion can update tuples already produced because resolution functions are not necessarily monotonic. In our example, if we’ve already computed majority-of-3 for the capital of a country and passed the result to the parent, we may need to propagate an update to modify the capital if the majority changes. The process of propagating updates up the plan becomes similar to incremental view maintenance [2].

### 3.3 Example Query Plan

We now describe our execution model in the context of a basic query plan, depicted in Figure 1a. The plan is for the following query on the example database from Section 2:

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

We use three fetch rules: ∅ ⇒ country (operator 8), country ⇒ language (operator 11), and country ⇒ capital (operator 14). Abbreviations in the query plan should be self-explanatory.

At a high level, the plan performs an outerjoin (operator 5) of a resolved version of CountryA (operator 6) and a resolved version of CountryD1 (operator 9), followed by a filter on language (operator 4). This result is outerjoined (operator 3) with a resolved version of CountryD2 (operator 12). Lastly, country and capital attributes are projected (operator 2). The Scan-Fetch operator pairs at the leaves of the plan will be explained shortly.

To keep things simple, in this example we illustrate the case where there are no existing tuples in any of the raw tables. First, the root operator sends eight “pull” requests to its child operator in parallel. Starting with eight requests based on MinTuples meets our two objectives in Section 3.1. These requests propagate down the left (outer) side of the joins, and eventually invoke fetch rule ∅ ⇒ 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 “pushed” into the plan by the Scan operator. Through the DLOJoin (for Dependent Left Outerjoin [5, 7, 10]), new countries trigger invocations of fetch rule country ⇒ 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 to produce an output value. 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 to exploit as much parallelism as possible while minimizing monetary cost. For example, if the two instances of fetch rule country ⇒ 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, or fetch rule ∅ ⇒ country gives a duplicated country value, the plan invokes a new instance of fetch rule ∅ ⇒ 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.

In Section 4, we provide further details of query execution for the case where there are no existing tuples in any of the raw tables. Then, in Section 5, we extend query execution for the general case where there may be existing tuples in raw tables.

### 4. QUERY EXEC. WITH NO EXISTING DATA

In this section, we describe in detail how to execute a Deco query plan starting with empty raw tables. Under this assumption, the “current” query result computed in the materialization phase is always empty and thus does not satisfy the “MinTuples n” constraint. Therefore, we focus on the accretion phase, which fetches new data to obtain n result tuples. Section 5 addresses the general case of existing raw data.

We first describe the basics of Deco query operators. Then we explain how those query operators work together during the accretion phase in the context of the basic query plan from Figure 1a as well as two alternative plans using more sophisticated fetch rules.

#### 4.1 Query Operators

We start by describing how Deco query operators communicate with each other and store intermediate tuples, then we describe the behavior of individual operators.

**Operator Overview**: Each query operator has a queue that stores messages received from other operators, and runs on its own thread that dequeues and processes one message at a time. Processing a

---

**Figure 1: Query plans**

(a) Basic
(b) Reverse
(c) Combined
message typically involves changing the operator’s local state as well as sending messages to its parent or child operators.

**Messages:** A message consists of a method to be run at the target operator and an optional argument. In the accretion phase covered in this section, bind messages pull more tuples from child operators, and add and remove messages push incremental changes to parent operators.

**Buffers:** Some operators have a local buffer that stores all input tuples to the operator. (Note DLOJoin and DepJoin have two local buffers, one for input tuples from the outer child and another for input tuples from the inner child.) These buffers are needed to determine the incremental changes to be propagated up the plan. The buffers are initially populated and indexed in the materialization phase, incrementally maintained in the accretion phase, and cleaned up when query execution terminates. All operators store their buffers in the same back-end RDBMS used for the raw tables, so large buffers can be handled efficiently.

**Partial Result:** We define the buffer contents at the root MinTuples operator as the *partial result* of the query: It contains the answer to the query computed so far, but may include tuples with NULL values. The goal is for the buffer to contain at least $n$ tuples with no NULL values, to satisfy the “MinTuples $n$” constraint.

Since our operators must deal with asynchrony and handle both push and pull requests, their details differ significantly from their iterator model counterparts. Table 1 summarizes the behavior of our operators for the case of initially empty raw tables. The entries in the tables will be explained in the next subsections.

### 4.2 Basic Query Plan

We now provide additional details of query execution using the example query plan in Figure 1a.

**Initial Fetches:** Initially, the MinTuples operator sends $n$ bind(messages to its child operator, where $n$ is the MinTuples constraint, and $\emptyset$ denotes an empty tuple. When an operator receives a bind message, its job is to produce at least one output tuple for its parent. Under the assumption that query execution starts with empty raw tables, all operators except Fetch simply pass the responsibility (i.e., the bind message) on to a child (the outer child for DLOJoin/DepJoin and Fetch child for Resolve), and Fetch operators invoke their fetch rules to obtain tuples from the crowd.

In our example plan, MinTuples operator 1 initiates eight bind messages, and these bind messages propagate down to Fetch operator 8, which invokes eight instances of its fetch rule $\emptyset \Rightarrow \text{country}$ in parallel.

**Populating Result:** When an outstanding fetch completes, the Fetch operator sends an add message with the new tuple to its corresponding Scan operator. The Scan operator inserts the tuple into its raw table, then the change is propagated up the plan via add and remove messages, similar to incremental view maintenance. (In Section 4.3 we will see that a single fetch can actually result in multiple changes.)

In addition to bind messages from the MinTuples root, some operators may initiate additional bind messages based on the tuples propagating up, in the following two cases:

- **DLOJoin and DepJoin** are dependent join operators [5, 7, 10]: they send bind messages to their inner children when they receive outer tuples that need to be matched.

- **Filter and Resolve** operators may initiate an additional bind message if an input tuple does not produce a new output tuple to propagate up. To minimize monetary cost, these operators have to ensure that the additional bind message is in fact necessary to satisfy the MinTuples constraint.

As a result, the set of outstanding fetches at any given time maintains the right degree of parallelism according to the objectives in Section 3.1.

Continuing our example, suppose an outstanding fetch $\emptyset \Rightarrow \text{country}$ completes with answer Peru. First, Fetch operator 8 sends an add message with tuple (Peru) to Scan operator 7. The Scan inserts the tuple into its raw table CountryA and forwards the add message to its parent. Then, Resolve operator 6 (duplicate elimination) forwards the add to its parent because its buffer contains no other (Peru) tuples. Upon receiving this add from the outer child, DLOJoin operator 5 sends add with NULL-padded tuple (Peru,NULL) to its parent because there are no matching tuples in its inner buffer. (Even though pushing padded tuples may seem unnecessary for this plan, in general all query operators must push all output tuples to their parents because they have no global view of the plan they belong to.) Filter operator 4 does not forward the padded tuple further because the tuple does not satisfy predicate language='Spanish'.

In addition to pushing new data up the plan, DLOJoin operator 5 sends a bind message with Peru to its inner child to obtain a language value for Peru. The job of Resolve operator 9 is to ultimately pass up an add message with tuple (Peru,X), where $X$ is a resolved language value for Peru. It does so by sending two bind messages to Fetch operator 11, which invokes two instances of fetch rule Peru $\Rightarrow$ language in parallel.

Now suppose both outstanding fetches Peru $\Rightarrow$ language complete with answer Spanish. For each completed fetch, Fetch operator 11 sends an add message with (Peru,Spanish) to Scan operator 10, which extends its raw table CountryD1 and forwards the add to its parent. Once Resolve operator 9 receives both adds with (Peru,Spanish), it sends add with (Peru,Spanish) to its parent.

Next, DLOJoin operator 5 modifies its output tuple (Peru,NULL) to (Peru,Spanish) using a remove and an add message. The new tuple passes through Filter operator 4 and reaches DLOJoin operator 3, which pushes a padded tuple (Peru,Spanish,NULL) up the plan. Also, DLOJoin operator 3 sends a bind message with Peru to its inner child. Once receiving an add message with (Peru,Y) from Resolve operator 12, the DLOJoin incorporates $Y$ into the partial result by modifying its output tuple (Peru,Spanish,NULL) to (Peru,Spanish,Y).

As a final example, suppose an outstanding fetch $\emptyset \Rightarrow \text{country}$ completes with answer Korea. Then, similarly to how Peru is processed, two instances of Korea $\Rightarrow$ language are invoked in parallel. When both instances complete with the same answer Korean, DLOJoin operator 5 sends its parent a remove message with (Korea,NULL) and an add message with (Korea,Korean). At this point, Filter operator 4 finds out that the new input tuple does not pass its predicate, so it initiates a bind message to obtain another country-language pair. This bind message propagates down the plan and invokes another instance of fetch rule $\emptyset \Rightarrow \text{country}$. (Note that Filter operator 4 does not initiate bind messages for input tuples with NULL language values: an additional bind message is initiated only if the predicate is evaluated as “false” in a three-valued logic.)

**Signaling Termination:** Once the partial result at the root operator satisfies the MinTuples constraint, all query operators are terminated, and the operator buffers in the back-end RDBMS are cleaned up. We will see later that in the general case, outstanding fetches may need to be canceled, but in the special case of empty raw tables, there cannot be outstanding fetches. Finally, all non-NULL result tuples in the buffer at the MinTuples operator buffer are returned to the client, and query execution terminates.
4.3 Alternative Plans

In addition to standard plan transformations based on relational algebra [15], our Fetch-Resolve-Join semantics enables additional plan alternatives based on the selection of fetch rules. In this section, we describe two alternative examples for our query.

**Reverse Fetches:** Suppose the predicate language='Spanish' is very selective. If we use the query plan in Figure 1a, even obtaining a single answer could be expensive in terms of monetary cost and latency, because we are likely to end up fetching many countries and languages that do not satisfy the predicate.

Instead, we can use the “reverse” fetch rule language ⇒ country underneath the Resolve operator to obtain countries with a certain language, rather than random countries. Figure 1b shows a query plan that uses this approach. Note the only change from Figure 1a is operator 8a. Whenever Fetch operator 8a receives a bind message from its parent, it invokes the fetch rule language ⇒ country with Spanish as the language value. (The association of predicate language='Spanish' with the Fetch operator is made when the query plan in Figure 1b is built.) When completed with a new country name \( Z \), this fetch rule invocation adds tuples to both CountryA and CountryD1 via Scan operators 7 and 10, and the two added tuples propagate up the plan separately. Assuming the country name \( Z \) is not a duplicate, Resolve operator 6 pushes \( Z \) up to DLOJoin operator 5, which sends a bind message to its inner child to complete the language of \( Z \). Meanwhile, Scan operator 10 pushes \((Z,\text{Spanish})\) to Resolve operator 9. Since the Resolve already has one raw language value for \( Z \), it ends up asking for one or two more language values using the fetch rule country ⇒ language (Fetch operator 11) to compute a resolved language value for \( Z \).

**Combined Fetches:** It may be less expensive to use a fetch rule that gathers multiple dependent attributes at the same time, rather than fetching attributes separately. For the example query, we can use the “combined” fetch rule country ⇒ language,capital instead of the two fetch rules country ⇒ language and country ⇒ capital. Figure 1c shows a query plan that uses this approach. Note that Figure 1c differs from Figure 1a only in operator 11a and the absence of operator 14. Both Resolve operators 9 and 12 send bind messages to Fetch operator 11a when they need more tuples. (Fetch operators keep track of outstanding fetches to ensure that redundant fetches are not invoked.) Once an outstanding fetch completes, Fetch operator 11a sends add messages to both Scan operators 10 and 13.

To illustrate a more interesting scenario, let us consider the query plan in Figure 1c but with majority-of-5 as the resolution function for capital in Resolve operator 12 (with shortcutting, so three matching values are needed). Suppose an outstanding fetch \( \emptyset \Rightarrow \) country completes with answer Bolivia. DLOJoin operator 5 initiates a bind message, and Fetch operator 11a invokes two instances of Bolivia ⇒ language,capital in parallel. Suppose both fetches complete with the same answer (Spanish,La Paz). Then, a new
output tuple (Bolivia,Spanish) reaches \textit{DLOJoin} operator 3, and a partial result tuple (Bolivia,NULL) is stored at \textit{MinTuples} operator 1. At the same time, another instance of Bolivia \Rightarrow language,capital is invoked to obtain another capital value. Upon receiving (Quechua,Sucré) as the third answer, the same fetch rule is invoked again. Suppose the fourth answer is (Aymara,La Paz). While \textit{Resolve} operator 12 finally produces La Paz as the resolved capital value, \textit{Resolve} operator 9 no longer has a majority for the language of Bolivia. Thus it sends a remove message with (Bolivia,Spanish) to its parent, which propagates up and eventually invalidates the partial result tuple (Bolivia,NULL). More bind messages are initiated until resolved values for country and capital are sufficient to produce the query result.

### 4.4 Join of Conceptual Relations

Although our example query from Section 3.3 contains only one conceptual relation Country in its \textit{From} clause, Deco's query execution engine also supports joining conceptual relations using the \textit{DepJoin} operator. While \textit{DLOJoin} handles outerjoins between anchor and dependent tables according to Deco's data model semantics, \textit{DepJoin} handles join predicates explicitly specified in \textit{Where} clauses. Both join operators are variants of dependent join, so they send bind messages with join values extracted from outer tuples to their inner children to receive matching inner tuples. However, \textit{DepJoin} is somewhat more complex than \textit{DLOJoin}. By definition, \textit{DLOJoin} is always an equijoin over anchor attributes, whose values are fixed by the time a bind message is issued to the inner child, i.e., no additional fetches or resolution functions are applied for these anchor values. On the other hand, \textit{DepJoin} predicates can be over any attributes (although restricted to equijoins; non-equijoins are handled as filters). When a \textit{DepJoin} sends a bind message with join attribute values to the inner child, due to the behavior of resolution functions, it cannot be assured that the inner tuple returned will have matching join values. If the values do not match, additional bind messages must be sent to produce the needed join result tuple.

### 5. QUERY EXEC. WITH EXISTING DATA

Now we describe how to execute a Deco query in the general case where there may be existing data in raw tables. As discussed in Section 3.2, this general case is significantly more difficult to handle than the special case of empty raw tables, in terms of minimizing monetary cost and reducing latency.

In this section we first describe how an initial query result is computed in the materialization phase and how the accretion phase differs from the special case described in Section 4.

#### 5.1 Materialization Phase

The materialization phase computes an initial query result based on the existing contents of the raw tables, and as a side effect populates operator buffers. Starting from \textit{Scan} operators at the bottom, each operator in the plan pushes its initial output tuples to its parent using populate messages, then sends a shift message to the parent indicating the end of the initial output tuples at the operator (thus "shifting" to the accretion phase, if needed). Table 2 summarizes the behavior of operators for materialization. When the root \textit{MinTuples} operator receives a shift message, its buffer contains the initial partial result. If the partial result satisfies the \textit{MinTuples} constraint, execution terminates. Otherwise, execution continues to the accretion phase.

Although our current approach for computing the initial result is fairly naive, we can easily incorporate many conventional techniques into the materialization phase to improve performance. For example, we can cache the resolved versions of the raw tables across queries so that \textit{Scan} operators do not have to send all raw tuples again for each query. In this case, the materialization phase starts from \textit{Resolve} operators, and \textit{Scan} operators do not participate. We could also incorporate a more traditional iterator model for the materialization phase, as long as it is properly extended to populate the operator buffers. Note that unless we have sufficient raw data, the bulk of query execution time is spent waiting for human answers, so latency improvement from speeding up the materialization phase is marginal.

As an example, suppose we execute the query plan in Figure 1a with the following initial contents for the raw tables:

<table>
<thead>
<tr>
<th>CountryD1</th>
<th>country</th>
<th>country</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chile</td>
<td>Chile</td>
<td>Spanish</td>
<td></td>
</tr>
<tr>
<td>Italy</td>
<td>Italy</td>
<td>English</td>
<td></td>
</tr>
<tr>
<td>Korea</td>
<td>Italy</td>
<td>Italian</td>
<td></td>
</tr>
<tr>
<td>Peru</td>
<td>Italy</td>
<td>Italian</td>
<td></td>
</tr>
<tr>
<td>Spain</td>
<td>Peru</td>
<td>Spanish</td>
<td></td>
</tr>
<tr>
<td>Spain</td>
<td>Spain</td>
<td>Spanish</td>
<td></td>
</tr>
</tbody>
</table>

Recall the resolution function for both language and capital is majority-of-3, with shortcircuiting. During the materialization phase, \textit{DLOJoin} operator 5 accumulates the following input tuples in its buffers:

<table>
<thead>
<tr>
<th>Outer buffer</th>
<th>Inner buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td>country</td>
</tr>
<tr>
<td>Chile</td>
<td>Chile</td>
</tr>
<tr>
<td>Italy</td>
<td>Italy</td>
</tr>
<tr>
<td>Korea</td>
<td>Korea</td>
</tr>
<tr>
<td>Peru</td>
<td>Peru</td>
</tr>
<tr>
<td>Spain</td>
<td>Spain</td>
</tr>
</tbody>
</table>

Among the five output tuples of the \textit{DLOJoin}, three of them do not pass \textit{Filter} operator 4: (Italy,Italian), (Korea,NULL), and (Peru,NULL). Thus, \textit{DLOJoin} operator 3 accumulates the following input tuples in its buffers:

<table>
<thead>
<tr>
<th>Outer buffer</th>
<th>Inner buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td>country</td>
</tr>
<tr>
<td>Chile</td>
<td>Chili</td>
</tr>
<tr>
<td>Spain</td>
<td>Spain</td>
</tr>
</tbody>
</table>

The initial partial result at \textit{MinTuples} operator 1 contains (Chile,Spanish) and (Spain,Madrid).

#### 5.2 Accretion Phase

Similarly to how the special case of empty raw tables proceeds (Section 4), the accretion phase first invokes some initial fetches to get started. Then, it extends result based on newly fetched data, possibly invoques more fetches as necessary, and finally signals termination once the \textit{MinTuples} constraint is met. We will later discuss how our approach meets the two objectives from Section 3.1.

**Initial Fetches:** As specified in Table 3, initial bind messages are generated for two distinct purposes:

- **Gathering Completely New Tuples:** The \textit{MinTuples} operator initiates \(k\) bind messages, where \(k\) is the minimum number of additional partial result tuples needed to satisfy the \textit{MinTuples} constraint (based on the partial result after materialization).
- **Joining Existing Tuples with New Tuples:** The other method of increasing the size of the result is to attempt to join existing outer tuples with new inner tuples. (For \textit{DLOJoin}, this means replacing NULL values with actual values.) The bind
At the start of accretion phase

Add to buffer. If the “MinTuples n” constraint is met, terminate execution and return all non-NULL tuples in the buffer to the client. Otherwise, proceed to the accretion phase.

Forward populate[ΠA(t)] to parent. Forward shift to parent.

Add t to buffer. Group all tuples in buffer by the left-hand side attributes of the resolution rule, apply the resolution function for each group, and send the resolved tuples to parent using populate messages. Then send shift to parent.

Add t to outer or inner buffer. Send the left outerjoin of the two buffers to parent using populate messages. Then send shift to parent.

Add t to outer or inner buffer. Send the join of the two buffers to parent using populate messages. Then send shift to parent.

Add t to buffer. Group all tuples in buffer by the left-hand side attributes of the resolution rule, apply the resolution function for each group, and send the resolved tuples to parent using populate messages. Then send shift to parent.

At the start of the materialization phase, for each t in the raw table, send populate[Πt] to parent. Then send shift to parent.

<table>
<thead>
<tr>
<th>Operator</th>
<th>On receiving populate[tf] (from child)</th>
<th>On receiving shift (from child)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinTuples</td>
<td>Add t to buffer.</td>
<td>If the “MinTuples n” constraint is met, terminate execution and return all non-NULL tuples in the buffer to the client. Otherwise, proceed to the accretion phase.</td>
</tr>
<tr>
<td>Project</td>
<td>Forward populate[ΠA(t)] to parent.</td>
<td>Forward shift to parent.</td>
</tr>
<tr>
<td>Filter</td>
<td>If p(t) is true (where p denotes the filter’s predicate), forward populate[tf] to parent.</td>
<td>Forward shift to parent.</td>
</tr>
<tr>
<td>DLOJoin</td>
<td>Add t to outer or inner buffer.</td>
<td>Send the left outerjoin of the two buffers to parent using populate messages. Then send shift to parent.</td>
</tr>
<tr>
<td>DepJoin</td>
<td>If all join attribute values are non-NULL, add t to outer or inner buffer.</td>
<td>Send the join of the two buffers to parent using populate messages. Then send shift to parent.</td>
</tr>
<tr>
<td>Resolve</td>
<td>Add t to buffer.</td>
<td>Group all tuples in buffer by the left-hand side attributes of the resolution rule, apply the resolution function for each group, and send the resolved tuples to parent using populate messages. Then send shift to parent.</td>
</tr>
<tr>
<td>Scan</td>
<td>At the start of the materialization phase</td>
<td></td>
</tr>
</tbody>
</table>
6. FETCH PRIORITIZATION

We now explain how we prioritize fetching data from the crowd, in order to minimize cost. Recall that all fetches are the result of bind messages passed down the plan until they reach Fetch operators. After a fetch rule has been invoked and before it returns, we refer to that instance as an “outstanding fetch.” As discussed earlier, at any given time there may be a large number of outstanding fetches executing in parallel; our goal is to profitably influence the order in which these outstanding fetches are handled. To separate our optimization goal from the details in crowdsourcing platforms, we assign scores to each outstanding fetch, reflecting our prioritization. How the scores result in actual fetch prioritization is platform-specific.

Based on our discussion in the previous section, all outstanding fetches have one of two purposes:

1. To create new partial result tuples. These fetches may be a result of bind messages originating from MinTuples, DepJoin, or DLOJoin operators.
2. To fill in NULL values in the partial result. These fetches are always a result of binds originating from DLOJoin operators.

For a “MinTuples $n$” constraint, our overall prioritization strategy is to first create at least $n$ partial result tuples (purpose 1), then fill in NULL values (purpose 2) until we have $n$ result tuples. Although this strategy may not be optimal in all cases, intuitively we must have $n$ partial result tuples regardless in order to produce a query result. Thus, we focus our optimization on filling in NULL values in a fashion that minimizes cost. As future work, we plan to consider the more difficult problem of globally optimizing purposes 1 and 2 together, which will undoubtedly depend in complex ways on the query plan itself.

To find the optimal set of fetches that can fill in enough NULL values to generate the query result, there are several important factors to consider:

- Our goal is to generate tuples with no NULL values, so we may prefer to fill in NULL values for tuples that are “nearly” complete.
- Sometimes a single fetch can fill in multiple NULL values.
- Since resolution rules often require multiple input values in order to produce one output value, multiple fetches may be needed to fill in one NULL value.

We first formalize our fetch prioritization problem and show that it is NP-hard. We then propose two heuristic scoring functions and describe their implementation in the context of Deco’s query execution engine. Lastly we describe how fetch prioritization is implemented in one crowdsourcing platform, Mechanical Turk.

6.1 Formal Problem Definition

Suppose we have $m$ partial result tuples for a query plan with “MinTuples $n$”. As described above, we are interested in the case where $m \geq n$, but fewer than $n$ tuples are non-NULL. The query processor creates a set of outstanding fetches, each one associated with one or more NULLs in the partial result. (For formalizing the problem, we assume resolution functions require a fixed number of input tuples, however we will see that our heuristics allow us to drop this assumption.) Our goal is to select a subset of the outstanding fetches that completes the result at minimum cost.

Finding the optimal solution for our fetch prioritization problem turns out to be NP-hard in the data size and the number of desired result tuples, using a reduction from the three-dimensional matching problem. A theorem and proof is given in Appendix A. Thus, we use a heuristic approach, described in Section 6.2. Here we formulate the fetch prioritization problem as a polynomial zero-one program [9] to motivate our heuristics. (In fact, the fetch prioritization problem can be posed as an integer linear program, making it NP-complete; however, we use the polynomial zero-one program for ease of exposition.)

Let $\text{NULL}_{i1}, \ldots, \text{NULL}_{ip}$, denote the $p_i$ NULL values of the $i$-th partial result tuple ($p_i \geq 0$). To fill in $\text{NULL}_{ij}$, $q(i,j)$ identical outstanding fetches $f_{q(i,j)1}, \ldots, f_{q(i,j)p(i,j)}$ must complete. (Note that an outstanding fetch $f_{hk}$ contributes to all $\text{NULL}_{ij}$’s with $g(i,j) = h$ and $q(i,j) = k$.) Finally, let $x_{hk}$ be an indicator variable specifying if fetch $f_{hk}$ is chosen. Then, the fetch prioritization problem is directly translated to the following:

\[
\text{minimize } \sum_{h,k} x_{hk} \text{ subject to } \sum_{i=1}^{m} \prod_{j=1}^{p} q(i,j) x_{g(i,j)k} \geq n \\
\quad x_{hk} = 0, 1 \ \forall h, k
\]

The objective function is the number of chosen fetches. The inequality constraint says that at least $n$ tuples have to be completed.

Example: Consider the following raw tables for the Country and City relations. Recall that the resolution functions for the language, capital, and population attributes are majority-of-3, majority-of-3, and average-of-2, respectively.

<table>
<thead>
<tr>
<th>CityA</th>
<th>CityD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>city</td>
<td>city</td>
</tr>
<tr>
<td>country</td>
<td>country</td>
</tr>
<tr>
<td>Istanbul</td>
<td>Venice</td>
</tr>
<tr>
<td>Italy</td>
<td>Italy</td>
</tr>
<tr>
<td>Venice</td>
<td>Venice</td>
</tr>
<tr>
<td>Venice</td>
<td>Italy</td>
</tr>
<tr>
<td>Trento</td>
<td>Venice</td>
</tr>
<tr>
<td>Italy</td>
<td>Italy</td>
</tr>
</tbody>
</table>

Suppose that we have the following five fetch rules:

- [Country] $\Rightarrow$ country
- [Country] country $\Rightarrow$ language
- [Country] country $\Rightarrow$ capital
- [City] $\Rightarrow$ city, country
- [City] city, country $\Rightarrow$ population

Further suppose our query asks for any two cities along with their population and language:

\[
\text{SELECT city, country, population, language FROM City, Country WHERE City.country = Country.country}\]

Evaluating the query over the conceptual relations implied by the current contents of the raw tables, we get:

<table>
<thead>
<tr>
<th>city</th>
<th>country</th>
<th>population</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Istanbul</td>
<td>Turkey</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Venice</td>
<td>Italy</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Trento</td>
<td>Italy</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Formulating this case as a polynomial zero-one program, we get:

\[
\begin{align*}
\text{minimize} & \quad x_{11} + x_{12} + x_{21} + x_{31} + x_{32} \\
+ x_{41} + x_{42} + x_{43} + x_{51} + x_{52} \\
\text{subject to} & \quad x_{11}x_{12}x_{21} + x_{31}x_{32}x_{41} + x_{21}x_{31}x_{52} \\
+ x_{31}x_{32}x_{51}x_{52} & \geq 2 \\
x_{hk} & = 0, 1 \ \forall h, k
\end{align*}
\]

Recall the objective is to minimize the number of chosen fetches out of all \( f_{hk} \)'s. Each term in the constraint indicates if its corresponding partial result tuple is completed; for example, \( x_{11}x_{12}x_{21} \) corresponds to the second partial result tuple for \((\text{Venice,Italy})\). The optimal solution is \( x_{11}=x_{12}=x_{31}=x_{32}=1 \), which translates to actual outstanding fetches \((\text{Venice,Italy}) \Rightarrow \text{population}, (\text{Trento, Italy}) \Rightarrow \text{population,} \text{ and Italy} \Rightarrow \text{language}. \)

Note that the number of outstanding fetches \( q(i,j) \) required to fill in a NULL value may not be fixed as in our formulation, depending on the resolution function. Even in our simple running example, majority-of-3 can stop after obtaining two raw values if they agree with each other. We will see that our heuristic approach accommodates variable values for \( q(i,j) \).

### 6.2 Heuristic Algorithm

Our practical solution to the NP-hard prioritization problem is based on assigning scores to outstanding fetches. The scores can then be used by the crowdsourcing platform to influence which fetches are most likely to complete (Section 6.4). Initial scores are assigned to outstanding fetches at the start of accretion phase based on a heuristic solution to the problem formalized in Section 6.1. As fetches complete and the partial result changes, scores for the remaining outstanding fetches may be adjusted. We will see in Section 7 that performance with our online heuristic approach can be close to optimal. We present two scoring functions. The first one assumes that only one fetch is needed to complete each NULL value, i.e., \( q(i,j) = 1 \). The second scoring function takes the resolution function into account and therefore multiple fetches may be needed, i.e., \( q(i,j) \geq 1 \).

Our first scoring function, \( \text{score}_1 \), is expressed in terms of \( p_i \), the number of remaining NULL values in each partial result tuple. Specifically, we regard the contribution of \( f_{hk} \) to the \( i \)-th partial result tuple as \( 1/p_i \) (if \( f_{hk} \) contributes to the tuple at all):

\[
\text{score}_1(f_{hk}) = \sum_{i=1}^{m} (\lfloor j : q(i,j) = h \rfloor \times \frac{1}{p_i})
\]

In our example in Section 6.1, we have \( p_1=2 \) for \( i=1,2,3 \), so \( \text{score}_1(f_{hk}) \) depends on the number of partial result tuples to which \( f_{hk} \) contributes. Thus, we have \( \text{score}_1(f_{hk})=1/2 \) for \( h=2,3,4 \) and \( \text{score}_1(f_{hk})=1 \).

Our second scoring function, \( \text{score}_2 \), is expressed in terms of the number of remaining fetches to complete each partial result tuple, \( \sum_{j=1}^{p_i} q(i,j) \). The contribution of \( f_{hk} \) to the \( i \)-th partial result tuple is \( 1/\sum_{j=1}^{p_i} q(i,j) \):

\[
\text{score}_2(f_{hk}) = \sum_{i=1}^{m} (\lfloor j : q(i,j) = h \rfloor \times \frac{1}{\sum_{j=1}^{p_i} q(i,j)})
\]

In our example in Section 6.1, we have \( \sum_{j=1}^{p_i} q(i,j) = 5, 3, 4 \) for \( i=1,2,3 \), respectively. Since each of \( f_{k1}, \ldots, f_{k4} \) contributes to a single partial result tuple, we get \( \text{score}_2(f_{k1})=\text{score}_2(f_{k2})=1/5 \), \( \text{score}_2(f_{k3})=1/3 \), and \( \text{score}_2(f_{k4})=1/4 \). Since \( f_{k5} \) contributes to two partial result tuples, we get \( \text{score}_2(f_{k5})=1/3+1/4 \). Notice the top-5 initial \( \text{score}_2 \) values correspond to the optimal solution.

### 6.3 Query Execution Engine Extension

We now explain how we have extended Deco’s query execution engine to compute scoring function \( \text{score}_1 \) or \( \text{score}_2 \) for each outstanding fetch. Actual prioritization of fetches based on the scores depends on fetch procedures and crowd interfaces. For Mechanical Turk, our fetch procedure always presents the outstanding fetch with the highest score whenever a worker arrives (see Section 6.4).

Since \( \text{score}_1 \) relies on the entire set of current partial result tuples, the MinTuple operator is responsible for computing it. However, computing \( \text{score}_1 \) incurs little overhead due to the nature of the execution model: the MinTuple operator can incrementally maintain the scores as partial result tuples are updated by add and remove messages from its child operator. Specifically, when a partial result tuple \( t \) is added or removed, the MinTuple operator can adjust \( \text{score}_1(f) \) for each outstanding fetch \( f \) that can fill in NULL values in the tuple \( t \).

Our approach requires us to track all outstanding fetches that can fill in a NULL value in a given partial result tuple \( t \). To do so, we assign a unique identifier for each outstanding fetch \( f \) and embed the identifier in the NULL values that \( f \) can fill in. DLOJoin and Fetch operators cooperate to perform this embedding.

For \( \text{score}_2 \), we can use the same overall approach as for \( \text{score}_1 \) except we need to manage \( q \), the number of remaining fetches to complete each NULL value. Resolve operators calculate \( q \) based on the numbers of required input tuples for the resolution function (set by the schema designer), and the tuples currently in its buffer. Resolve operators must propagate calculated \( q \) values up to the MinTuple operator so that \( \text{score}_2 \) can reflect up-to-date \( q \) values. In this case, simply embedding \( q \) in its corresponding NULL value is not enough: Whenever \( q \) changes, Resolve operators must modify the \( q \) part of the NULL value using a remove plus an add message. When these messages reach the MinTuple operator, the \( \text{score}_2 \) values can be updated without affecting the partial result. Due to the additional computation and messages, computing \( \text{score}_2 \) is somewhat more expensive than computing \( \text{score}_1 \); however, we will see in Section 7 that \( \text{score}_2 \) is a better heuristic.

### 6.4 Mechanical Turk Support

We now describe how Deco’s generic Mechanical Turk (MTurk) fetch procedure supports fetch prioritization based on scoring as described in the previous subsection. The MTurk fetch procedure translates fetch rule invocations from the query execution engine into HITs (Human Intelligence Tasks, unit tasks on MTurk) so that workers can answer them. For prioritization, the goal of the MTurk fetch procedure is to present the outstanding fetch with the highest score whenever a worker “accepts” one of the HITs. To do so, the MTurk fetch procedure works together with a dedicated Deco form server, which hosts HTML forms that the workers fill in, and keeps track of scores for outstanding fetches.

More specifically, when the MTurk fetch procedure is invoked by a Fetch operator, the procedure instantiates a form template for the fetch rule, sends the HTML form and its score to the form server using a web service API, and creates a HIT that points to the form server. The MTurk fetch procedure also forwards any changes in scores for outstanding fetches from the query execution engine to the form server. When a worker accepts any of the HITs, the worker’s web browser loads the address of the form server, which then chooses the HTML form with the highest score and serves it to the worker. Typically MTurk workers accept a series of HITs within their chosen HIT type, presented by MTurk in random order. Overall, our approach replaces random scheduling with score-based prioritization using our form server to delay the binding of HTML forms to HITs.
7. EXPERIMENTAL EVALUATION

In this paper we focus our experimental evaluation of the Deco execution model on the interactions among parallelism, cost, and latency. (In [15] we presented preliminary experiments showing Deco’s capability of executing query plans with different fetch rules, using Mechanical Turk.) Also, we evaluate the effectiveness of our fetch prioritization scheme, compared against the optimal prioritization and random prioritization.

To enable a large number of experiments without significant latency and dollar cost, we built a crowd simulator that responds to fetch requests by selecting values from a predetermined set. We can either set our simulator to always give “correct” answers (e.g., it always returns Lima as the capital of Peru), or specify a fixed probability for each fetch rule that “incorrect” answers are given.

7.1 Parallelism, Cost, and Latency

Experiment 1: Varying Degrees of Parallelism

Our first experiment determines whether Deco’s query execution engine accomplishes the two objectives in Section 3.1, by comparing monetary cost and latency of executing the same query plan with varying degrees of parallelism. We use the example query from Section 3.3:

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

For this experiment we start with empty raw tables. Thus, Deco begins by invoking eight anchor-value fetches in parallel, so the crowd can contribute to all eight (potential) result tuples at the same time. Additional fetches are invoked as needed when some of the original eight complete, but never more fetches than would contribute to eight result tuples. In our graphs, the x-axis corresponds to the degree of parallelism \(d\), defined as the maximum number of result tuples being produced in parallel. Thus, Deco’s behavior for our example query is at \(d=8\).

Figure 2a shows the monetary cost and latency of executing the reverse plan in Figure 1b for \(d=1...15\), with our crowd simulator configured to always give correct answers. We assume each fetch takes 5 seconds and costs $0.05, for all fetch rules. Since each result tuple needs three rounds of answers (e.g., Spanish \(\Rightarrow\) Peru, Peru \(\Rightarrow\) Spanish, and two instances of Peru \(\Rightarrow\) Lima in parallel), the minimum cost and latency are \(8 \times 4 \times 0.05 = $1.6\) and \(3 \times 5s = 15s\), respectively. In fact, Deco’s behavior \((d=8)\) minimizes both monetary cost and latency. As discussed in Section 3.2, too much parallelism \((d>8)\) increased monetary cost, while too little \((d<8)\) increased latency. Compared to no parallelism \((d=1)\), equivalent to the traditional iterator model), Deco reduced latency dramatically without increasing the cost.

Figure 2b shows the cost and latency of executing the reverse plan, with our simulator configured to give incorrect answers with 25% chance. Each data point reported is the average of 50 trials. Due to incorrect answers, Deco often ends up invoking fetch rule language \(\Rightarrow\) country more than eight times, so the cost remains at the minimum until \(d=9\), which is the best degree of parallelism. Although Deco did not quite achieve the minimum, latency is still very close to the minimum across \(d=1...9\).

Figure 2c shows the cost and latency of executing the basic plan in Figure 2a, with our simulator set to always give correct answers. Since cost and latency of executing this plan depend on the order in which countries are obtained, we picked a random order and used the same order for all data points. As in Figure 2a, Deco found the best degree of parallelism \((d=8)\), reducing latency as much as possible while minimizing cost, thereby meeting the objectives laid out in Section 3.1. (As a side note, the non-monotonicity of cost increase for \(d>8\) is related to the positions Spanish-speaking countries appear in the random order of countries obtained.)

7.2 Effectiveness of Fetch Prioritization

Now we present two additional experiments evaluating the effectiveness of fetch prioritization. Experiment 2 evaluates our two scoring functions for varying amounts of existing data in the raw tables, while Experiment 3 evaluates the scoring functions for different data distributions. In both experiments, we compare our scoring functions against the optimal prioritization computed using a brute-force enumeration, and against random prioritization (i.e., equal scores).

For both experiments, we use the following fetch rules:

- [Country] country \(\Rightarrow\) language, and country \(\Rightarrow\) capital
- [City] city.country \(\Rightarrow\) population

Recall the resolution functions for the language, capital, and population attributes are majority-of-3, majority-of-3, and average-of-2. Also, we set our simulator to always give correct answers, so that comparison against the optimal case makes sense. Each data point in the figures is the average of ten trials, which had little variance.

Experiment 2: Varying Amount of Existing Data

We use the following simple query:

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

We seed anchor table CountryA with 100 different country names. In this setting, each outstanding fetch contributes to only one NULL value (either language or capital for a particular country), so the partial result tuples are independent from each other. We can achieve the optimal prioritization by completing first those partial result tuples requiring fewer overall fetches to fill in. In fact, score<sub>2</sub> does exactly this in this experiment: For a partial result tuple requiring \(k\) fetches to fill in, each of those \(k\) fetches has a score<sub>2</sub> of \(1/k\). On the other hand, score<sub>1</sub> first completes partial result tuples with fewer NULLs, ignoring the effect of resolution functions on number of overall fetches.
Figure 3a shows the number of fetches completed to obtain \( X \) result tuples using \( \text{score}_1 \) and \( \text{score}_2 \), starting with empty dependent tables. Because our simulator gives correct answers only, we always need 400 fetches (two fetches for each NULL value) to obtain 100 result tuples. We observe that prioritization based on \( \text{score}_1 \) and \( \text{score}_2 \) needed 22% and 34% fewer fetches on average than random prioritization. As expected, \( \text{score}_2 \) achieves the optimal prioritization for this particular experiment.

In Figures 3b and 3c, we start the query with 100 and 200 tuples, respectively, in the two dependent tables combined. These tuples are randomly chosen from 400 correct tuples that would fill in all NULL values. In Figure 3b, the overall trends are similar to Figure 3a: prioritization based on \( \text{score}_1 \) and \( \text{score}_2 \) needed 28% and 41% fewer fetches on average than random prioritization. In Figure 3c, prioritization based on \( \text{score}_1 \) and \( \text{score}_2 \) needed 32% and 39% fewer fetches on average than random prioritization. In Figure 3c the difference between \( \text{score}_1 \) and \( \text{score}_2 \) is much smaller because the simplifying assumption behind \( \text{score}_1 \) holds more often: As we seed more tuples in the dependent tables, more NULL values can be filled in by completing one outstanding fetch.

**Experiment 3: Different Data Distributions**  We use the following join query:

```sql
SELECT city, country, population, language
FROM Country, City
WHERE City.country = Country.country
MIN TUPLES X
```

Anchor tables CountryA and CityA are already populated with a varying number and distribution of values. In this query, some outstanding fetches for language might complete multiple NULL values in the partial result, depending on the distribution of city-country pairs. Because the previous experiment showed that existing data has little impact on the overall trends, we start with empty dependent tables.

Figures 4a and 4b show the number of fetches to obtain \( X \) result tuples for two real datasets: the 100 largest European cities and the 200 largest cities in the world. Each country in the European and world city dataset has 4.3 and 2.9 cities on average, respectively. The European city dataset contains eight countries (out of 23) with only one city, while the world city dataset contains a long tail of 39 countries (out of 69) with only one city. In Figure 4a, \( \text{score}_2 \)-based and optimal prioritization needed 34% and 40% fewer fetches on average than random prioritization. In Figure 4b, \( \text{score}_2 \)-based and optimal prioritization needed 37% and 42% fewer fetches on average than random prioritization. Overall, prioritization based on \( \text{score}_2 \) is quite close to the optimal across the entire range of \( X \).

For Figure 4c, we deliberately generated a synthetic set of 100 city-country pairs to make our \( \text{score}_2 \)-based prioritization work as poorly as possible: all language values must be completed before any population values. Considering the resolution functions majority-of-3 and average-of-2, \( \text{score}_2 \) for country \( \Rightarrow \) language can be as low as \( k/5 \) where \( k \) is the number of cities for the country value. Since \( \text{score}_2 \) for city,country \( \Rightarrow \) population can be as high as \( 1/2 \) in this scenario, \( k \) should be at least 3 for all countries. Thus, the worst-case dataset contains 32 countries with three cities and one country with four cities. In Figure 4c, \( \text{score}_2 \)-based prioritization and the optimal schedule needed 19% and 35% fewer fetches on average than random prioritization.

8. **RELATED WORK**

Among prior work in the general area of query processing, WSQ-DSQ [7] and incremental view maintenance [2] are most relevant to Deco’s query execution. WSQ-DSQ enables an iterator execution model to concurrently access high-latency data sources, in the context of integrating web search results into a relational database. However, the WSQ-DSQ “placeholder” solution does not apply to Deco because of Deco’s flexible fetch and resolution rules. Parallel database systems [3] also access data sources in parallel. However, they typically rely on static data partitioning and operator replication, so their approach is not applicable to Deco, where arbitrary and dynamically adjustable degrees of parallelism must be sup-
ported. View maintenance algorithms propagate base table updates to a materialized view. Deco’s query execution engine similarly applies given base (raw) table changes to a partial query result, but in Deco base-data changes also influence further query execution.

Recently, several data-oriented systems leveraging crowdsourcing have been proposed. CrowdDB [6] closely resembles Deco, but Deco opts for more generality and flexibility and thus requires the novel query processing techniques described in this paper. (A detailed comparison between the two systems can be found in [15].) Qurk [12, 13] is a workflow system that supports crowd-powered operations on existing relational data. CDAS [11] improves accuracy of data analytics jobs using crowdsourcing techniques.

Finally, there have been several papers describing crowd-powered algorithms for basic database operations such as filtering items [14], finding max [8], sorting and joins [12], and collecting data [18]. Some of these algorithms may be incorporated into our Deco prototype to improve individual query operators.

9. CONCLUSIONS AND FUTURE WORK

We presented Deco’s query processor, focusing on how to minimize monetary cost and reduce latency when executing a query plan. We incorporated several novel techniques to answer Deco queries correctly and efficiently, including a hybrid execution model, dynamic fetch prioritization, two-phase query execution, and incremental view maintenance.

We are currently developing a complete query optimizer for Deco, requiring a cost estimator and plan enumeration strategy. We are also interested in extending the Deco prototype to support more general SQL queries beyond Select-Project-Join; for example, order-by and group-by are certainly useful SQL constructs in any environment. Finally, we plan to incorporate alternatives to MinTuples, such as MaxCost and MaxTime, described in Section 2.

10. REFERENCES


APPENDIX

A. PROOF OF HARDNESS OF THE FETCH PRIORITIZATION PROBLEM

Theorem 1. The Fetch Prioritization Problem is NP-hard.

Proof. We now reduce the NP-hard three-dimensional matching problem to the fetch prioritization problem, showing that the fetch prioritization problem is NP-hard.

In the three-dimensional matching problem, there are three sets of items A, B, and C, of equal size n. We are given a set of triples \( S = \{(a_i, b_i, c_i)\} \subseteq A \times B \times C \), |S| = n. The goal is to check if a perfect matching exists, i.e., a subset \( R \subseteq S \) of size n such that each \( a_i \), each \( b_i \), and each \( c_i \) appear exactly once.

We reduce the three-dimensional matching problem to the following instance of the fetch prioritization task: We create a conceptual relation \( G(A_1, A_2, A_3, [D_1], [D_2], [D_3], [D_4]) \), such that the following resolution rules hold (all correspond to majority-of-1 function):

\[
A_1 \rightarrow D_1 \quad A_2 \rightarrow D_2 \quad A_3 \rightarrow D_3 \quad A_1A_2A_3A_4 \rightarrow D_4
\]

and the single fetch rule:

\[
A_1A_2A_3A_4 \Rightarrow D_1D_2D_3D_4
\]

Based on the schema and the resolution rules, the data is stored in five tables (corresponding to the raw schema): one anchor table \( T_0 \) corresponding to \( A_1, A_2, A_3, A_4 \), and one dependent table \( T_i \) for each \( D_i \) (along with the left-hand-side of the corresponding resolution rule.)

The current data in these five tables is the following: \( T_0 \) consists of all tuples \((a_i, b_i, c_i, 0)\) and \((a_i, b_i, c_i, 1)\) such that \((a_i, b_i, c_i) \in S\) in the corresponding three-dimensional matching problem. Additionally, \( T_i \) consists of tuples \((a_i, b_i, c_i, 1, d)\) for all \((a_i, b_i, c_i) \in S\); none of \( T_1, T_2, T_3 \) contain any tuples.

Now, in the fetch prioritization problem, our goal is to select the smallest set of fetch invocations such that \( n \times \alpha \) output tuples are produced. Note first that the optimal set of fetch invocations must have \( A_4 = 0 \). To see this, if there are fetch invocations with \( A_4 = 1 \), then we can replace them with fetch invocations with \( A_4 = 0 \) to only possibly increase the number of output tuples, and then reduce the number of fetch invocations. (Recall that we already know \( D_4 \) values for all tuples in \( T_0 \) with \( A_4 = 1 \).)
Next, note that we need at least $n$ fetch invocations: We already have $m$ tuples in $T_0$ whose $D_4$ values are known (all with $A_4 = 1$). In order to complete $n$ more tuples, we must issue at least $n$ more fetch invocations, each of which will give us the $D_4$ value for precisely one tuple. Furthermore, as we saw earlier, all of these invocations must have $A_4 = 0$.

We now show that if the solution to the fetch prioritization problem requires only $n$ fetch invocations, then the fetch invocations correspond to the solution for the three-dimensional matching problem. Specifically, if $a_i, b_i, c_i, 0 \Rightarrow D_1, D_2, D_3, D_4$ is issued as a fetch, then $(a_i, b_i, c_i)$ is part of $R$, the solution to the three-dimensional matching problem. To see this, note that if only $n$ invocations are used, then each $a_i \in A_1, b_i \in A_2, c_i \in A_3$ appears precisely once in the fetch invocations. Only then will we complete all $m$ tuples in $T_0$ with $A_4 = 1$.

Furthermore, we always invoke fetches corresponding to complete tuples $\{(a_i, b_i, c_i, 0)\} \in T_0$ in order to fill in as many $D_4$ values as possible. Thus, our selection of fetch invocations corresponds to a perfect matching.

The converse is also true. That is, if there is a perfect matching, then the corresponding set of fetch invocations will be of size $n$ and will form a solution to the fetch prioritization problem. □