Deco: Declarative Crowdsourcing

Aditya Parameswaran
Stanford University
adityagp@cs.stanford.edu

Hyunjung Park
Stanford University
hyunjung@cs.stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Neoklis Polyzotis
UC Santa Cruz
alkis@cs.ucsc.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

Crowdsourcing enables programmers to incorporate "human computation" as a building block in algorithms that cannot be fully automated, such as text analysis and image recognition. Similarly, humans can be used as a building block in data-intensive applications—providing, comparing, and verifying data used by applications. Building upon the decades-long success of declarative approaches to conventional data management, we use a similar approach for data-intensive applications that incorporate humans. Specifically, declarative queries are posed over stored relational data as well as data computed on-demand from the crowd, and the underlying system orchestrates the computation of query answers.

We present *Deco*, a database system for declarative crowdsourcing. We describe Deco's data model, query language, and our prototype. Deco's data model was designed to be *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). Syntactically, Deco's query language is a simple extension to SQL. Based on Deco's data model, we define a precise semantics for arbitrary queries involving both stored data and data obtained from the crowd. We then describe the Deco query processor which uses a novel push-pull hybrid execution model to respect the Deco semantics while coping with the unique combination of latency, monetary cost, and uncertainty introduced in the crowdsourcing environment. Finally, we describe our current prototype, and we experimentally explore the query processing alternatives provided by Deco.

1. INTRODUCTION

Crowdsourcing [12, 31] uses human workers to capture or generate data on demand and/or to classify, rank, label or enhance existing data. Often, the tasks performed by humans are hard for a computer to do, e.g., rating a new restaurant or identifying features of interest in a video. We can view the human-generated data as a *data source*, so naturally one would like to seamlessly integrate the crowd data source with other conventional sources, so the end user can interact with a single, unified database. And naturally one would like a *declarative* system, where the end user describes the needs, and the system dynamically figures out what and how to obtain crowd data, and how it must be integrated with other data. This overall vision, and the underlying issues and challenges, were outlined in our earlier paper [27].

In this paper, we realize our earlier vision to present Deco (short for "declarative crowdsourcing"), a database system that answers declarative queries posed over stored relational data, the collective knowledge of the crowd, as well as other external data sources. Our goal is to make Deco appear to the end user as similar as possible to a conventional database system (a relational one in our case), while hiding many of the complexities of dealing with humans as data sources (e.g., breaking down large tasks into smaller ones, posting tasks to a marketplace and pricing them, dealing with latency, and handling errors and inconsistencies in human-provided data).

We describe the Deco data model, the query language and semantics, and the query processor.

While the idea of declarative access to crowd data is appealing and natural, there are significant challenges to address:

- How do we resolve disagreeing human opinions? For instance, if we collect five ratings for a movie, we may want to give the end user the average, but if we collect five phone numbers, we may want to instead eliminate duplicates. How does the schema designer (or DBA) specify what to do, and when during query execution do we resolve the opinions?
- How does the database system interact with the human workers (the crowd)? For instance, to get restaurant information, we may want Deco to give the worker the name of a restaurant (e.g., "Bouchon"), and ask for its cuisine. But in other cases Deco may want to give the worker the cuisine (e.g., "French"), and ask for restaurants serving that cuisine. Or Deco may ask for cuisine and rating given the name of a restaurant. How does the schema designer define the available options, which we can view as different "access methods"? Are there restrictions on the access methods that can be defined? And how does Deco decide what access method to use for a given query? How do we enable Deco to use external sources in addition to the crowd? For instance, we may want Deco to be able to use an information extraction program that, given a restaurant, extracts a phone number from a webpage.
- What is the right data model and query language for a crowdsourced database? We already argued for a model and end user language that are as close as possible to conventional ones, but extensions are needed to deal with the uncertainties and ambiguities of crowd data. For instance, since there is an "endless supply" of crowd data, the end user (or the schema designer) needs to circumscribe what is needed. For instance, the user may state that five answer tuples are sufficient, or that five opinions are enough to report an average movie rating. How are such constraints/guidelines defined, and by whom?
- What data should the crowdsourced database system actually store? Is it cleansed data, or is it the uncleansed data? If it is cleansed data, then how do we update it as new opinions arrive? If it is uncleansed data, what does the user see, when is it computed, and how can it be stored compactly? Should there be a notion of crowdsourced data becoming stale?

- How does the query processor execute crowd queries? How does it deal with the complexity of having several access methods to the crowd? How does it deal with the latency of crowdsourcing services? How does it deal with the resolution of disagreement or uncertainty in answer tuples? How does it ensure that constraints (such as a desired number of answer tuples) are met?

Our Deco design addresses these questions, trying to strike a balance between too much generality, and achieving an elegant, implementable system. While it may not be immediately apparent to the reader, the design of Deco required significant effort as well as consideration of many possible alternative designs. In particular, we will argue in our related work section (Section 7) that Deco is more principled, general, and flexible than recent approaches for declarative crowdsourcing [14, 24, 25].

In summary, our contributions are the following:

- We present a data model for Deco (Section 2) that is practical, based on sound semantics, and derived from the familiar relational data model. We define the precise semantics of the data model (Section 2.6). We also describe important data model extensions (Section 2.7).
- We describe the Deco query language (Section 3) that minimally departs from SQL, and expresses the constraints necessary for crowdsourcing.
- We present our design for the query processor (Section 4) which uses a novel push-pull hybrid execution model.
- We experimentally analyze queries running on the Deco prototype (Section 6). We show that our Deco prototype supports a large variety of plans with varying performance, and we gain some valuable insights in the process.
- We compare Deco's design with other declarative crowdsourcing systems, and survey other related work (Section 7).

1.1 Running Example

Throughout the paper we use a running example that is by design simple and a bit contrived, yet serves to illustrate the major challenges of declarative crowdsourcing, and to motivate our solutions to these challenges. We (as the users) are interested in querying a (logical) database containing information about *restaurants*. Each restaurant may have one or more *addresses* (multiple addresses indicate a chain), and each restaurant may serve one or more *cuisines*. We also have *ratings*, which are associated with restaurant-address pairs since different outlets of a chain may be rated differently. We will see that Deco's *conceptual schema* encourages denormalized relations, so our restaurant information can be captured in a single logical relation:

Restaurant(name,address,rating,cuisine)

The catch is that our database may not contain a full set of information to begin with, in fact it may not contain any information at all! In response to queries, we obtain information from external sources—including humans—using a set of "fetch rules" (formalized as part of our data model). For example, given a restaurant and an address we might seek ratings, or given a rating and a cuisine we might seek restaurant-address pairs. We will shortly see many more examples of fetch rules, as well as other aspects of Deco relations, such as how we deal with the uncertainty resulting from inconsistencies in the information we obtain.

We add a second (logical) relation containing address information. We assume a restaurant's address may be encoded in some fashion—perhaps as a string or a geolocation—but we may want to pose queries involving, say, cities or zipcodes. Thus we have the relation:

AddrInfo(address,city,zip)

Here too, we may obtain some or all of the information from external sources in response to queries. We might use human workers for this task, or we might invoke information extraction tools, perhaps with results verified by humans. In addition, we may need to coordinate values between our two relations, since queries are likely to join them on their address fields.

We will see how these logical relations form the conceptual schema that is declared by the administrator setting up a Deco database, and they are the relations queried by end users and applications. Underneath, Deco uses a different *raw schema*. The heart of Deco query processing is the significant machinery and algorithms for combining stored data with crowdsourced (or other externally-obtained) data to answer declarative queries over the conceptual schema.

2. DATA MODEL

There are several components to Deco's data model:

- The *conceptual schema*. In our running example introduced in Section 1.1, relations Restaurant and AddrInfo are part of the conceptual schema. These are the relations specified by the schema designer, and they are queried by end users and applications. The conceptual schema also includes:
  - Partitioning of the attributes in each conceptual relation into *anchor attributes* and *dependent attribute-groups*. Roughly speaking, anchor attributes typically identify "entities" while dependent attribute-groups specify properties of the entities, although schema designers could use attributes differently.
  - *Fetch rules*, specifying how data in the conceptual relations can be obtained from external sources (including humans).
  - *Resolution rules*, used to reconcile inconsistent or uncertain values obtained from external sources.
- The *raw schema*. Deco is designed to use a conventional relational DBMS as its back-end. The raw schema is the one stored in the DBMS. It is derived automatically from the conceptual schema, and is invisible to both the schema designer and end users.
- The *data model semantics*. We specify the semantics of a Deco schema in terms of conventional relations. Loosely, the "valid instances" of a Deco database are those conventional databases that can be constructed by executing fetch rules to expand the contents of the raw tables, then applying resolution rules and joining the raw tables to produce the conceptual relations.

In the remainder of this section we begin by illustrating each of the data model components informally, then we step through each component formally.

We use the term *designer* to refer to both the schema designer as well as the database administrator. We use the term *user* to refer to the end user or the application developer.

2.1 Example of Data Model Components

Consider the restaurant relation introduced in Section 1.1: Restaurant (name, address, rating, cuisine). The designer designates name and address as *anchor attributes*, and rating and cuisine as *dependent attributes*. Informally, we can see that the pair of name and address attributes together identify the "entities" in our relation while the other two attributes are properties of entities; we will see shortly the specific roles of anchor versus dependent attributes.

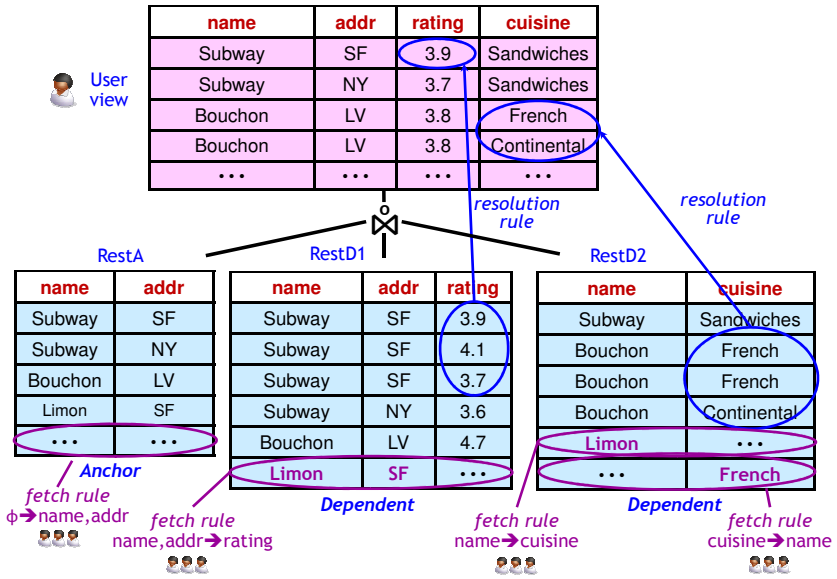


Figure 1: Components of the Deco Data Model

The raw schema corresponding to this specification of Restaurant is shown in the lower half of Figure 1. These relations are the ones actually stored as tables in the back-end RDBMS. There is one *anchor table* (RestA) containing the anchor attributes, and one *dependent table* for each dependent attribute (RestD1 and RestD2); dependent tables also contain some anchor attributes. (In general, both anchor and dependent attributes can be a group of attributes.) Recall that we associate cuisines with the restaurant name, and rating with a name-address pair, since different branches of a restaurant (such as NY and SF in Figure 1—to save space, we use abbreviated addresses) can have different ratings, but all branches serve the same kind of food. We will see in Section 2.5 how the raw schema are generated.

The top of the figure shows the original conceptual relation, which is the outerjoin of the raw tables with certain attribute values "resolved" (explained shortly).

Now let us consider how our database might be populated. Perhaps we already have some restaurant name-address pairs, with or without ratings and/or cuisines. If so, Deco might ask human workers to specify ratings and/or cuisines given a restaurant name and/or address. Alternatively, Deco might ask human workers to specify restaurant names and addresses given a cuisine and/or rating, or to provide restaurant names without regard to ratings or cuisines. Referring to Figure 1, the designer can specify *fetch rules* that:

- Ask for one or more restaurant name-address pairs, inserting the obtained values into raw table RestA.
- Ask for a rating given a restaurant name and an address (e.g., (Limon,SF) in the figure), inserting the resulting pair into table RestD1; similarly ask for a cuisine given a restaurant name (e.g., Limon in the figure), inserting the resulting pair into RestD2.
- Ask for a restaurant name given a cuisine, inserting the resulting restaurant into table RestA, and inserting the restaurant-cuisine pair into RestD2 (e.g., French in the figure).

These fetch rules are depicted at the bottom of the raw tables in Figure 1. There are many more fetch rules that may be used to populate this database, we return to this point later on.

Now suppose we've obtained values for our raw tables, but we have inconsistencies or uncertainty in the collected data. One decision we made in Deco is to provide a conceptual schema that does not have uncertainty as a first-class component, however *metadata* in both the raw and conceptual schemas (described later in Section 2.7) can be used to encode information about confidence, worker quality, or other aspects of collected data that may be useful to applications. To obtain conceptual relations that are "clean" from raw tables that may contain inconsistencies, we use *resolution rules*, specified by the designer. In Figure 1 we illustrate two resolution rules:

- A resolution rule for attribute rating specifying that the conceptual schema contains one rating for each restaurant name-address pair, namely the average of the ratings stored in the raw schema.
- A resolution rule for attribute cuisine specifying that the conceptual schema contains all of the cuisines for each restaurant name from the raw schema, but with duplicates eliminated.

The semantics of a Deco database is defined based on a *Fetch-Resolve-Join* sequence. Every Deco database has a (typically infinite) set of *valid instances*. A valid instance is obtained by logically: (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. Note that the "intermediate" relations between steps (2) and (3) are not depicted in Figure 1; in the figure we resolve and join in one step. Also 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 typically no conceptual data is materialized except the query result.

Note that valid instances could contain wildly varying amounts of data, from no tuples at all to several million tuples, and they are all valid. So, when a user poses a query to the database, the valid instance used to answer his query may be the one with no tuples at all. We therefore need a mechanism to allow the user to request that at least a certain number of tuples are returned, discussed further in Section 3.

2.2 Conceptual Relations, Anchor and Dependent Attributes

Now let us begin formalizing the concepts illustrated in Section 2.1. The designer declares two conceptual relations, and designates their anchor and dependent attributes. In our example:

Restaurant(name,address,[rating],[cuisine])
AddrInfo(address,[city,zip])

Each attribute is either an *anchor attribute* or a member of exactly one *dependent attribute-group*. Thus, to partition a relation's attributes, it suffices to enclose dependent attribute-groups within square brackets. In our example, the pair of anchor attributes name and address identify individual restaurants, while rating and cuisine are dependent attributes that are independent of each other. (We assume ratings are not associated with specific cuisines of a restaurant.) In relation AddrInfo, address is the anchor attribute; attributes city and zip are not independent of each other with respect to an address, so they form a single dependent attribute-group. The next subsection clarifies the purpose of these designations within the Deco data model.

2.3 Resolution Rules

For each conceptual relation, the schema designer must specify one resolution rule for each dependent attribute-group, and one resolution rule for the anchor attributes treated as a group. Thus, a resolution rule takes one of the following two forms, where A, A', and D are sets of attributes and f is a function.

1. $A' \rightarrow D : f$
   where A' is a subset of the anchor attributes ($A' \subseteq A$) and D is a dependent attribute-group
2. $\varnothing \rightarrow A : f$
   where A is the set of anchor attributes

In the first form, *resolution function* f takes as input a tuple of values for the anchor (A') attributes, and a set of tuples with values for the dependent (D) attributes. It produces as output a new (possibly empty, possibly unchanged) set of dependent tuples. The idea is that function f "cleans" the set of dependent values associated with specific anchor values, when the dependent values may be inconsistent or uncertain. In the second form, function f "cleans" a set of anchor (A) values. In either case, if no cleaning is necessary then f can simply be the identity function.

Let us look at a set of resolution rules for our running example. (Rules RR_1 and RR_2 are displayed in Figure 1.)

[RR_1, relation Restaurant] name,address → rating : *avg*()
[RR_2, relation Restaurant] name → cuisine : *dupElim*()
[RR_3, relation Restaurant] ∅ → name,address : *canonicalize*()
[RR_4, relation AddrInfo] address → city,zip : *majority*()
[RR_5, relation AddrInfo] ∅ → address : *identity*()

(Note the function specifications here are abstracted; in practice the Deco system requires resolution functions to adhere to a specific API.) The interpretation of these rules is:

- **RR_1:** The rating for a specific restaurant (name-address pair) in the conceptual database is the average of the ratings in the raw data.
- **RR_2:** The cuisines for a restaurant are associated with the restaurant name but not its address (under the assumption all outlets in a chain serve the same cuisine). The conceptual database contains each cuisine for each restaurant in the raw data, with duplicates removed.
- **RR_3:** The name-address pairs in the conceptual database are a "canonicalized" version of the raw name-address pairs. Canonicalization can put the values in a particular form, and can perform "entity resolution" to merge differing name- address pairs that are judged to refer to the same restaurant.
- **RR_4:** If the raw data contains more than one city-zip pair for a given address, the pair occurring most frequently is the one present in the conceptual database. We can assume ties are broken arbitrarily.
- **RR_5:** Assuming address values in the AddrInfo relation are never uncertain or inconsistent, the simple identity resolution function is used in this case.

As an example, the three tuples corresponding to (Subway,SF,*) in raw table RestD1 in Figure 1 are resolved using resolution function RR_1 into a single tuple (Subway,SF,3.9), which then participates in the join with RestA and RestD2.

Readers inclined towards dependency theory may already have noticed that that resolution rules suggest multivalued dependencies on the conceptual relations. In relation Restaurant we have the multivalued dependencies name,address→↠rating and name→ →cuisine. Furthermore, when a resolution rule for a dependent attribute-group is guaranteed to produce exactly one value, we have a functional dependency. For instance, name and address functionally determine rating, since the resolution rule for rating produces exactly one value.

2.4 Fetch Rules

The schema designer may specify any number of fetch rules. Unlike resolution rules, fetch rules may be added or removed at any time during the lifetime of a database—they are more akin to "access methods" than to part of the permanent schema. A fetch rule takes the following form:

$$A_1 \Rightarrow A_2 : P$$

where A_1 and A_2 are sets of attributes from one relation (with $A_1 = \varnothing$ or $A_2 = \varnothing$ permitted, but not both), and P is a *fetch procedure* that implements access to human workers or other external sources. (P might generate HITs (Human Intelligence Tasks) to Amazon's Mechanical Turk [3], for example, but nothing in our model or system is tied to AMT specifically.) The only restriction on fetch rules is that if A_1 or A_2 includes a dependent attribute D, then $A_1 \cup A_2$ must include all anchor attributes from the left-hand side of the resolution rule containing D. In other words, if A_D is the left hand side of a resolution rule containing dependent attribute D, then, if $D \subseteq A_1 \cup A_2$, then $A_D \subseteq A_1 \cup A_2$. We will see the reason for this restriction in Section 2.6.

Let us look at a set of possible fetch rules for our running example. We use R and A to abbreviate Restaurant and AddrInfo respectively. (Rules FR_1, FR_2, FR_4, and FR_5 are displayed in Figure 1.)

[FR_1] R.name, R.address ⇒ R.rating : P_1
[FR_2] R.name ⇒ R.cuisine : P_2
[FR_3] A.address ⇒ A.city, A.zip : P_3
[FR_4] R.cuisine ⇒ R.name : P_4
[FR_5] ∅ ⇒ R.name, R.address : P_5
[FR_6] R.name ⇒ R.address : P_6
[FR_7] ∅ ⇒ R.name, R.cuisine : P_7
[FR_8] R.name, R.address ⇒ ∅ : P_8
[FR_9] R.address ⇒ R.name, R.rating, R.cuisine : P_9

The interpretation of these rules is:

- **FR_1–FR_3:** It is very common—though not required—to have a set of fetch rules that parallel the resolution rules. For example, fetch rule FR_1 says that procedure P_1 takes a specific

restaurant (name-address pair) as input, and it accesses humans or other external sources to obtain zero or more rating values for that restaurant.

- **FR_4:** Another common form is the reverse of a resolution rule: gather anchor values for given dependent values. For example, FR_4 says that procedure P_4 takes a cuisine as input, and accesses humans or other external sources to obtain one or more restaurant name values for that cuisine.
- **FR_5–FR_6:** Fetch rules can be used to gather values for anchor attributes, either from scratch or from other anchor attributes. FR_5 says restaurant name-address pairs can be obtained without any input, while FR_6 says that address values can be obtained given name values.
- **FR_7:** Fetch rules can gather anchor and dependent values simultaneously. Instead of asking for a cuisine given a name (as in FR_2) or a name given a cuisine (as in FR_4), procedure P_7 takes no input and asks for name-cuisine pairs.
- **FR_8:** Fetch rule FR_8 is the reverse of FR_7 and has a quite different function. A fetch rule with $\varnothing$ on the right-hand side performs verification: Procedure P_8 takes a name-address pair and accesses humans or other external sources to return a yes/no verification of the input values. A similar rule might be used to verify address-city-zip triples for relation AddrInfo, for example.
- **FR_9:** Fetch rule FR_9 demonstrates a fetch rule that can't be pigeon-holed into a particular structure. It says that procedure P_9 will take an address as input, and will obtain zero or more name-rating-cuisine triples for that address. Note that this rule would not be allowable if name were omitted: The presence of dependent attributes rating and cuisine requires that the anchor attributes from their resolution rules (name,address and name respectively) are also present.

Invocations of the Fetch Rule FR_1 (with (Limon,SF)), FR_2 (with Limon), FR_4 (with French), FR_5 (with no input) are depicted in Figure 1. Note that these fetch rules affect at most one dependent raw table; other fetch rules, such as FR_9, affect multiple dependent raw tables.

There are many, many more possible fetch rules for our running example. Which fetch rules are used in practice may depend on the capabilities (and perhaps costs) of the human workers and other external data sources, and perhaps the programming burden of implementing a large number of fetch procedures. Also remember that fetch rules are not set in stone—they can be added, removed, and modified as capabilities change or tuning considerations dictate.

2.5 Raw Schema

The raw schema—for the tables actually stored in the underlying RDBMS—depends only on the definitions of the conceptual relations, anchor and dependent attributes, and resolution rules. Specifically, for each relation R in the conceptual schema, the raw schema contains:

- One *anchor table* whose attributes are the anchor attributes of R
- One *dependent table* for each dependent attribute-group D in R, containing the attributes in the resolution rule for D

From the conceptual schema in Section 2.2, and the resolution rules in Section 2.3, it is straightforward to derive the following raw schema:

RestA(name,address)
RestD1(name,address,rating)
RestD2(name,cuisine)
AddrA(address)
AddrD1(address,city,zip)

For readers inclined towards dependency theory: Continuing from the discussion at the end of Section 2.3, we can see that the raw schema is in fact a Fourth Normal Form (4NF) decomposition of the conceptual schema based on the multivalued dependencies implied by the resolution rules.

Applications or end users may wish to insert, modify, and/or delete data in the conceptual relations, in a standard database fashion. The mapping from conceptual relations to raw tables is simple enough that all such modifications can be translated directly to corresponding modifications on the raw tables. More interestingly, if an application has chosen to use Deco, then we expect some of the data stored in the raw tables to be obtained from human workers or other external sources over time as part of query processing. How that process works, while properly reflecting our data model and query language semantics, is the topic of much of the remainder of this paper. Next we define the data model semantics, which centers around the mapping from raw tables to conceptual relations.

2.6 Data Model Semantics

The semantics of a Deco database at a given point in time is defined as a set of *valid instances* for the conceptual relations. The valid instances are determined by the current contents of the raw tables, the potential of the fetch rules, and the invocation of resolution rules before outerjoining the raw data to produce the conceptual relations. Let us consider this *Fetch-Resolve-Join* sequence in detail. Note that these three steps may not actually be performed, but query answers must reflect a valid instance that could have been derived by these three steps, as we will discuss in Section 3.

1. Fetch

The current contents of the raw tables may be extended by invoking any number of fetch rules any number of times, inserting the obtained values into the raw database. Consider a fetch rule $A_1 \Rightarrow A_2 : P$ with $A_2 \neq \varnothing$. By definition, procedure P takes as input a tuple of values for the attributes in A_1, and produces as output zero or more tuples of values for the attributes in A_2. We can equivalently think of an invocation of the fetch rule as returning a set $\mathcal{T}$ of tuples for $A_1 \cup A_2$. Note the input values for A_1 may come from the database or from a query (see Section 4), but for defining semantics we can assume any input values.

The returned tuple set $\mathcal{T}$ with schema $A_1 \cup A_2$ may include any number of attributes from any number of raw tables. Consider any of the raw tables $T(A)$, and let $B = A \cap (A_1 \cup A_2)$ be the attributes of T present in $\mathcal{T}$. If B is nonempty, we insert $\Pi_B(\mathcal{T})$ into table T, assigning NULL values to the attributes of T that are not present in $\mathcal{T}$ (i.e., the attributes in $B - A$). Informally, we can think of the process as vertically partitioning $\mathcal{T}$ and inserting the partitions into the corresponding raw tables, with anchor attributes typically replicated across multiple raw tables. The one guarantee we have, based on our one fetch rule restriction (Section 2.4), is that no NULL values are inserted into anchor attributes of dependent tables. This property simplifies the resolution process, discussed shortly. As an example, fetch rule FR_9 above would insert a tuple in all three restaurant raw tables. Note that inserting tuples in this manner may end up proliferating duplicate tuples, especially in the anchor raw table. As an example, if we have a name-address pair in RestA, then every time we use the name-address pair in the fetch rule name,address $\Rightarrow$ rating, we end up adding an extra copy of the same name-address pair to RestA. If duplicates may pose a problem, then the origin of the tuples (i.e., the fetch rule invocation that added them) can be recorded in the metadata (Section 2.7) and used by the resolution function.

Lastly, consider a fetch rule $A_1 \Rightarrow \varnothing : P$. In this special case where a fetch rule is being used for verification (recall Section 2.4), we proceed with the insertions as described above, but only when P returns a yes value. If the no values are important for resolution, a more general scheme for verification may be used, and is described in Section 2.7 below.

2. Resolve

After gathering additional data via fetch rules, each anchor and dependent table in the raw schema is logically replaced by a "resolved" table. Consider a dependent table $T(A', D)$ with corresponding resolution rule $A' \rightarrow D : f$. We logically: (a) group the tuples in T based A' values; (b) call function f for each group; (c) replace each group with the output of f for that group. (Recall from Section 2.3 that f takes as input a tuple of values for the A' attributes and a set of tuples with values for the D attributes. It produces as output a new, possibly empty or unchanged, set of tuples for D.) Resolving an anchor table $T(A)$ simply involves invoking the function f in the resolution rule $\varnothing \rightarrow A : f$ with all tuples in T.

Note that NULL values may appear in anchor tables, and for dependent attributes in dependent tables. We assume if NULL values are possible, then the corresponding resolution functions are equipped to deal with them. We need not worry about NULLs when grouping anchor attribute values (step (a) above), since our fetch-rule restriction (Section 2.4) ensures that all anchor values in dependent tables are non-NULL.

In Figure 1, resolution on the three RestD1 tuples corresponding to Subway, SF returns an average rating of 3.9. Since cuisine has a duplicate-elimination resolution function, resolution on the three tuples corresponding to Bouchon in RestD2 returns two tuples (one for French, one for Continental).

3. Join

The final step is simple: For each conceptual relation R, a left outerjoin of the resolved anchor and dependent tables for R is performed to obtain the final tuples in R. Note that an outerjoin (as opposed to a regular join) is necessary to retain all resolved values, since values for some attributes may not be present. In Figure 1, resolved tuples (Subway,SF) from RestA, (Subway,SF,3.9) from RestD1 and (Subway,Burgers) from RestD2 join to give the tuple: (Subway,SF,3.9,Burgers).

The left outerjoin must have the anchor table as its first operand, but otherwise the resolved dependent tables may be left outerjoined in any order; Appendix A proves that all such orders are equivalent.

To recap, a *valid instance* of a Deco database is obtained by starting with the current contents of the raw tables and logically performing the three steps above, resulting in a set of data for the conceptual relations. Since in step 1 any number of fetch rules may be invoked any number of times (including none), we typically have an infinite number of valid instances. This unboundedness is not a problem; as we will see shortly, our goal is to deliver the result of a query over *some* valid instance, not over all valid instances.

We emphasize again that these three steps are logical—they are used only to define the semantics of the data model. Of course some amount of fetching, resolving, and joining does need to occur in order to answer queries, but not necessarily as "completely" as defined above, and not necessarily in *Fetch-Resolve-Join* order. Section 4 describes our query processing strategies that respect the semantics while meeting cost, performance, and quality objectives.

2.7 Metadata

The Deco data model as described so far has precise formal underpinnings and relies heavily on the relational model with all of its benefits. However in reality there are several messy aspects of using crowdsourced (or other external) data that also must be accommodated—pieces of the crowdsourcing puzzle that we have chosen not to make first-class in our data model, yet are crucial for some applications. Examples include:

- *Data expiration:* If we store fetched data in the raw tables for future queries, we may wish to purge it automatically at some later point in time.
- *Worker quality:* When data is obtained from humans, we may have some knowledge of the expected quality, e.g., based on the crowdsourcing service, or on worker history.
- *Voting:* In Section 2.4 we described how fetch rules can be used to ask humans or other sources to verify uncertain values. If the response of such a fetch rule is a yes/no answer, we could record the tally of yes/no votes. This tally can then be used for resolution.
- *Confidence scores:* While worker quality and voting are mechanisms associated with confidence in data values, ultimately we may wish to associate explicit confidence scores with our data.
- *Fetch Rule:* Since some fetch rules may be more accurate or informative than others, we may wish to record which fetch rule generated which data.

All of these examples can be handled by allowing additional *metadata* columns in the raw tables. Metadata columns can be used for timeout values, to store information about worker quality, to tally votes, and for confidence values. Metadata attributes can be included in the input and/or output of fetch procedures and resolution functions. In our examples, *data expiration* and *worker quality* values might be returned by a fetch function; a resolution function might take *worker quality* and *fetch rules* as part of its input and return *confidence scores* as part of its output. Metadata columns can be exposed in the conceptual relations too, if their contents may be of direct use to applications.

3. QUERY LANGUAGE

The Deco query language and semantics is straightforward:

> *A Deco query Q is a relational query over the conceptual relations. The answer to Q is the result of evaluating Q over some valid instance of the database as defined in Section 2.6.*

We assume either relational algebra or SQL for posing queries. Since the Deco system supports SQL, we use SQL for examples in this paper.

Referring back to Section 2.6, recall that one valid instance of the database can be obtained by resolving and joining the current contents of the raw tables, without invoking any fetch rules. Thus, it appears a query Q can be always answered correctly without consulting human workers or other outside sources. The problem is that often times this "correct" query result will also be empty. For example, if our query is:

```
Select name,address From Restaurant
Where cuisine='Thai' and rating > 4
```

and there are currently no highly-rated Thai restaurants in our database, then the query result is empty.

To retain our straightforward semantics over valid instances while avoiding the empty-answer syndrome, we simply add to our query language an "AtLeast n" clause. This clause says that not only must the user receive the answer to Q over some valid instance of the database, but it must be a valid instance for which the answer

has at least n tuples with non-NULL attributes. Adding "AtLeast 5" to the query above, for example, forces the query processor to collect additional data using fetch rules until it obtains the name and address values for at least five highly-rated Thai restaurants.

Our basic problem formulation for query processing, discussed in detail in the next section, thus requires a minimum number of tuples in query results, while attempting to optimize other dimensions such as number of fetches, monetary cost, response time, and/or result quality. These considerations suggest other possibilities at the language level, for example:

- Specify MaxTime instead of AtLeast, in which case the query processor must deliver the result within a certain amount of time, and attempts to return as many tuples as possible.
- Specify MaxBudget or MaxFetches instead of MaxTime or AtLeast, in which case the query processor is constrained in how much external data it can obtain (based on monetary cost or number of fetch procedures allowed), and attempts to return as many tuples as possible.

There are many interesting variants, but it is important to note that all of them rely on the same query language semantics: return the relational answer to Q over some valid instance of the conceptual relations while satisfying any performance-related constraints.

4. QUERY PROCESSING

Having defined the Deco data model and query language, we now consider how to build a query processor that implements the defined semantics. The query processor must support constraints such as AtLeast while dealing with the inherent challenges in crowdsourcing, such as latency, cost, and uncertainty.

We first describe Deco's execution model, then we present a space of alternate plans that demonstrate Deco's flexibility. Plan costing and selection is a topic of ongoing work.

4.1 Execution Model

The Deco data model and query semantics gives us some unique challenges to address at the execution-model level.

- First, the result tuples may change based on the added data in the raw tables. For example, an additional rating value for a given restaurant may change the output of the resolution function. A traditional *iterator* model does not allow us to modify values in tuples once they are passed up the plan.
- Second, Deco fetch rules are general enough to provide tuples spanning multiple raw tables. For example, a fetch rule invocation name,address $\Rightarrow$ rating,cuisine provides a rating and a cuisine at the same time. Thus, data can be inserted in the raw table for cuisine even if the intent was to obtain more ratings. The iterator model has no mechanism for operators to signal arrival of new data to parent operators.
- Finally, since crowdsourcing services have high latency and provide natural parallelism, our execution model needs to invoke multiple fetch rules in parallel. While *asynchronous iteration* [15] can solve this problem, it does not solve the other problems above.

To address these challenges, Deco uses a novel *Push-Pull Hybrid* Execution Model, drawing on a combination of ideas from incremental view maintenance [7] and asynchronous iteration (developed in WSQ-DSQ [15]). It has the following features:

- **Incremental Push**: We borrow ideas from incremental view maintenance to make sure that the output reflects the current state of the raw tables. The result of a fetch rule is handled as an update to one or more base tables (raw tables in our case), and then propagated to the view (the conceptual table in our case).
- **Asynchronous Pull**: We borrow ideas from asynchronous iteration to cause messages to flow down the plan to initiate multiple new fetches in parallel and feed more tuples back to the plan as soon as any fetches complete. The messages that flow down the plan are similar to *getNext* requests in an iterator model, except that the parent operator does not wait for a response (due to the inherent latency in crowdsourcing, it would take a very long time to get a response). Without these messages, the query processor would only be able to passively apply given changes to the raw tables rather than actively drive them.
- **Two Phase**: The query processor needs to make sure that fetches are issued only if sufficient data is not present in the raw tables to answer the query. Deco achieves this constraint by using a two phase execution model, where the first phase, termed *materialization*, tries to answer the query using the current content of raw tables, and the second phase, termed *accretion*, issues fetch rules to obtain more result tuples.

Significant additional details of the Push-Pull Hybrid Execution Model are given in [30].

4.2 Alternate Plans

To show the flexibility of our execution model, we present four different query plans for the following query on the Restaurant relation from Section 1.1:

Q: Select name,address,rating,cuisine
From Restaurant Where rating $>$ 4 AtLeast 5

As resolution functions, we use duplicate elimination for name-address, average of 3 (or more) tuples for rating, and majority of 3 (or more) for cuisine. We will see in Section 6 how these different query plans perform in terms of execution time and monetary cost.

Basic Plan: Figure 2(a) shows a basic query plan that uses the fetch rules $\varnothing \Rightarrow$ name,addr (operator 7), name,addr $\Rightarrow$ rating (operator 10), and name $\Rightarrow$ cuisine (operator 13). At a high level, the plan performs an outerjoin (operator 4) of a resolved version of RestA (operator 5) and a resolved version of RestD1 (operator 8), followed by a filter on rating (operator 3). Then the result is outerjoined (operator 2) with a resolved version of RestD2 (operator 11). The root operator (operator 1) stops the execution once five output tuples are generated.

Predicate Placement: Alternatively, we can place the *Filter* operator above both of the *DLOJoin* or Dependent Left Outerjoin operators 16 and 17 (Figure 2(b)).

Due to the nature of our execution model, this simple transformation has more significant implication. The plan in Figure 2(a) will fetch cuisine only for the restaurants that satisfy rating$>$4, so it may have lower monetary cost. On the other hand, the plan in Figure 2(b) increases the degree of parallelism by fetching rating and cuisine at the same time, while having a higher monetary cost.

Reverse Fetches: Another interesting alternative plan can use reverse fetch rules. For our query Q, suppose the predicate rating $>$ 4 is very selective. If we use the query plans in Figure 2, even obtaining a single answer could be expensive in terms of latency and monetary cost, because we are likely to end up fetching a number of restaurants that do not satisfy the predicate.

Instead, we can use the reverse fetch rule rating $\Rightarrow$ name,address underneath the *Resolve* operator to start with a restaurant with a certain rating (according to at least one worker) instead of a random restaurant. Figure 3(a) shows a query plan that uses this reverse

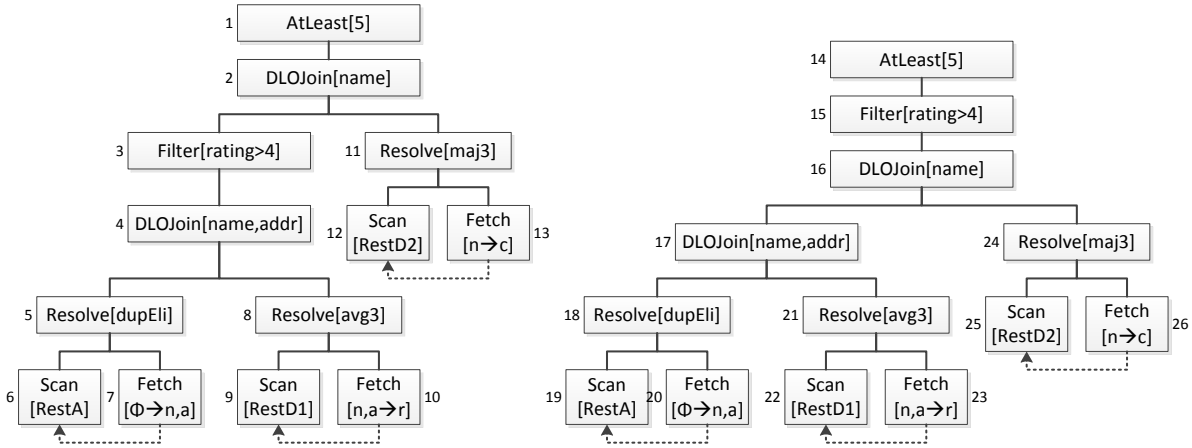


Figure 2: (a) Basic Plan (b) Filter Later

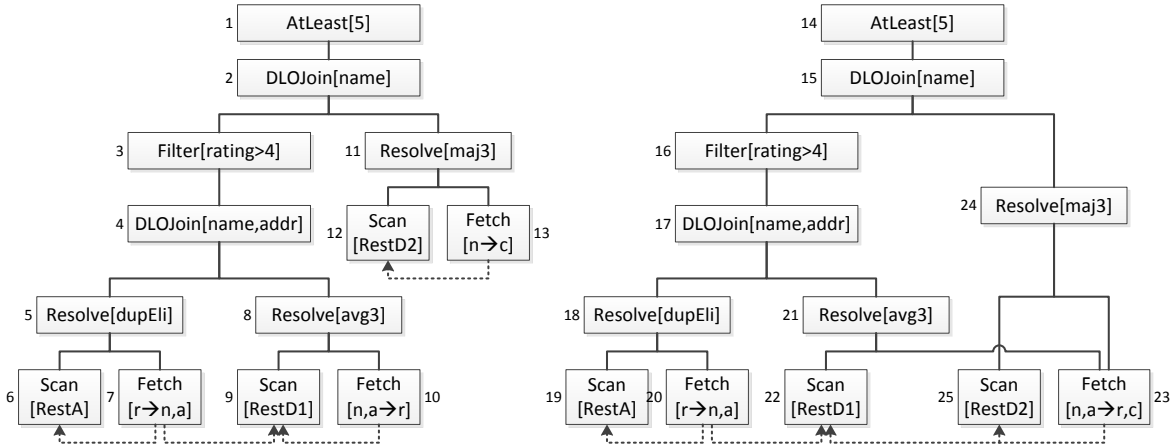


Figure 3: (a) Reverse Fetch (b) Combined Fetch

fetch rule. Notice that the *Fetch* operator 7 "pushes" tuples to both RestA and RestD1 via the *Scan* operators 6 and 9 (dotted arrows).

Combined Fetches: It may be less expensive to use a fetch rule that gathers multiple dependent attributes at the same time, rather than fetching one attribute at a time. For the example query Q, we can use a combined fetch rule name,address ⇒ rating,cuisine instead of two fetch rules name,address ⇒ rating and name,address ⇒ cuisine. Figure 3(b) shows a query plan that uses this combined fetch rule. Notice that both *Resolve* operators 21 and 24 "pull" more tuples from the *Fetch* operator 23.

5. SYSTEM

We implemented our Deco prototype in Python with a SQLite back-end. Currently, the system supports some DDL commands that create tables, resolution functions, and fetch rules as well as a DML command that executes queries. In this section, we describe Deco's overall architecture (Figure 4).

Client applications interact with the Deco system using the Deco API, which implements the standard Python Database API v2.0: connecting to a database, executing a query, and fetching results. The Deco API also provides an interface for registering and configuring fetch procedures and resolution functions. Using the Deco API, we built a command line interface, as well as a web-based graphical user interface that executes queries, visualizes query plans, and shows log messages in real-time.

When the Deco API receives a query, the overall process of parsing the query, choosing the best query plan, and executing the chosen plan is similar to a traditional DBMS. However, the query planner translates declarative queries posed over the conceptual schema to execution plans over the raw schema, and the query executor is not aware of the conceptual schema at all. To obtain data from humans, the query executor invokes fetch procedures, and the raw data is cleaned by invoking resolution functions.

6. EXPERIMENTAL EVALUATION

To illustrate the types of performance results that can be obtained from Deco, we present two experiments. One studies how different fetch rule configurations affect query performance, while the second experiment evaluates different query plans given a particular fetch rule configuration. Given our space limitations, it is impossible to present here more comprehensive results.

Experimental Setup: We used the following conceptual relation:

Country(name,[language],[capital])

The anchor attribute name identifies a country, while the two dependent attributes language and capital are independent properties of the country. For resolution functions, we used *dupElim* (duplicate elimination) for name, and *majority-of-3* for language and capital. (We assume one language per country.) The resolution function *majority-of-3* finds the majority of three or more tuples.

In addition, we created six fetch rules that use the Mechanical Turk fetch procedure to fetch data. Our experiments use one of the following fetch rule configurations:

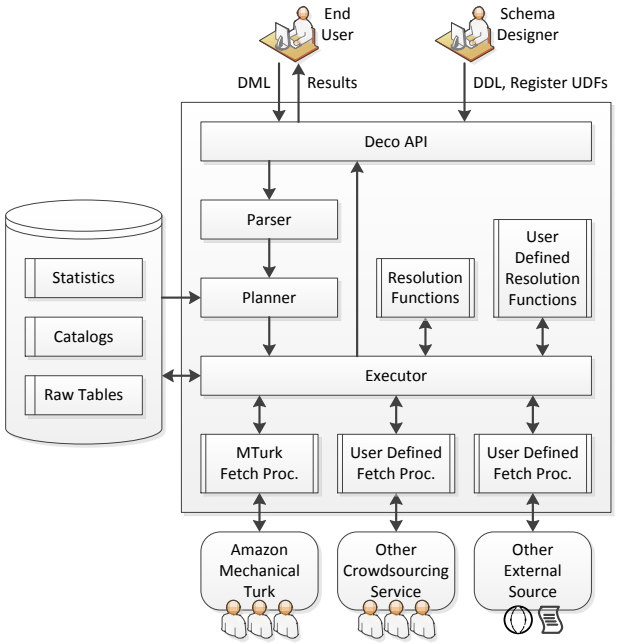


Figure 4: Deco Architecture

- Basic configuration (3 fetch rules): $\varnothing \Rightarrow$ name, name $\Rightarrow$ language, and name $\Rightarrow$ capital
- Reverse configuration (3 fetch rules): language $\Rightarrow$ name, name $\Rightarrow$ language, and name $\Rightarrow$ capital
- Hybrid configuration (2 fetch rules): language $\Rightarrow$ name,capital and name $\Rightarrow$ language,capital

For attributes name and capital, workers were allowed to enter free text, but our question server validated this input to ensure that the text had no leading or trailing whitespaces and that all letters were uppercase. For language, workers were allowed to select one language from a drop-down list of about 90 languages.

On Mechanical Turk, we paid a fixed amount of five cents per HIT (i.e., per fetch rule invocation) as compensation for completed work. All experiments were conducted on weekends in Feb 2012.

Our benchmark query requested eight Spanish-speaking countries along with their capital cities.

```
Select name,capital From Country
Where language='Spanish' AtLeast 8
```

For each experiment, we begin with empty raw tables. According to Wikipedia, there are 20 Spanish-speaking countries in total.

Experiment 1: Performance for Different Fetch Configurations: In our first experiment, we evaluated the query performance of different fetch rule configurations to study the usefulness of the flexibility of Deco fetch rules. Our fetch configurations are by default translated into query plans similar to Figure 2(a) (Basic), Figure 3(a) (Reverse), Figure 3(b) (Hybrid), where the raw table containing names is left outerjoined with the raw table containing names and languages, followed by a filter on language, and then the result is left outerjoined with the raw table containing names and capitals. However, the fetch operators in each of these plans use different fetch rules. For instance, for the Hybrid configuration, the fetch operator for the name raw table uses language $\Rightarrow$ name,capital, while the (shared) fetch operator for language and capital raw tables uses name $\Rightarrow$ language,capital as the fetch rule.

We ran the same benchmark query with the three sets of fetch rules (therefore different query plans) on our Deco prototype [29]. For each set, we ran the query twice and noted similar results. Both runs are included in our graphs.

Figure 5(a) depicts the number of non-NULL result tuples obtained over time. Since our query specified AtLeast 8, reaching eight output tuples signifies the completion of query. In addition, Figure 5(b) shows the number of HITs submitted by workers over time. Note that the monetary cost of a query is proportional to the total number of HITs submitted.

Using the hybrid configuration, the query took 10.5 minutes and cost $1.35 for 27 worker answers on average (across two runs). Using the reverse configuration, the query took 15 minutes and cost $2.30 for 46 worker answers on average. In comparison, the basic configuration performed very poorly: the query took two hours overall and cost around $12.05. (We ended up collecting 64 countries and their languages.)

Thus, we find that it is important to consider a variety of query plans, and the decisions of the query optimizer can significantly impact query performance in terms of latency and total monetary cost. Choosing fetch rules is a significant decision under the control of the optimizer. In particular, for this benchmark query, those fetch rules that get multiple pieces of information at once (such as name$\Rightarrow$language,capital), and those that operate in the reverse direction (such as language$\Rightarrow$name) offer significant benefits over basic fetch rules. In general, it is important for a declarative crowdsourcing system (like Deco) to handle fetch rules of different types, and thus increase the opportunities for finding a good execution plan. (Of course, we depend on the schema designer to take advantage of the flexibility and provide multiple fetch rules.)

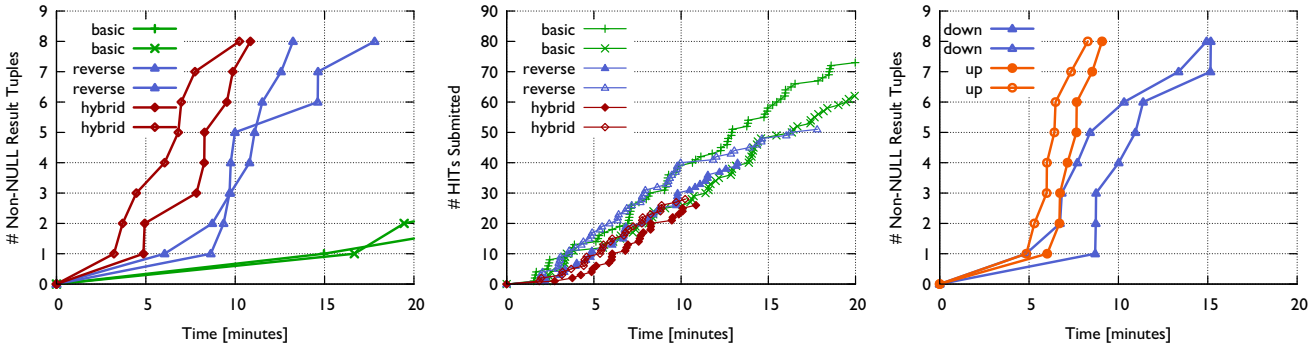
Overall, our findings validate the role of fetch rules as "access methods" for the crowd, and reinforce the importance of including them in a principled query optimization framework.

We repeated our experiment with the constraint AtLeast 12 instead of AtLeast 8 (graphs not shown). While the relative performance of our configurations was similar, it was interesting to observe that AtLeast 12 queries did not take significantly longer than AtLeast 8 despite producing more answers, but they did incur higher cost. This behavior is because our execution model issues more fetches (i.e., HITs) in parallel (via bind messages) for AtLeast 12 than for AtLeast 8, and a larger number of HITs in the same HIT group attract more workers. We plan to investigate trading off cost for time by varying the number of bind messages in future work.

In terms of the quality, the results were clean and correct except one typo ("Ddominican Republic"). Not suprisingly, individual answers had several errors that are not exposed in the result. Common errors for capital cities were for Spain and Bolivia.

Experiment 2: Performance for Different Query Plans: In this experiment, we evaluated the query performance of different query plans under the reverse fetch rule configuration. Specifically, we studied the performance implication of pulling up the (language = 'Spanish') predicate. Our Deco prototype pushes all predicates down as much as possible by default and produces a query plan similar to Figure 3(a) in the reverse configuration. Under this plan (termed "down"), Deco does not invoke the fetch rule name $\Rightarrow$ capital until language is resolved to Spanish. On the other hand, under the plan (termed "up") that is produced by applying the predicate pull-up transformation (similar to Figure 2(b)), Deco invokes fetch rules name $\Rightarrow$ language and name $\Rightarrow$ capital simultaneously as soon as a new country name is fetched.

We ran the same benchmark query with these two query plans. Figure 5(c) depicts the number of non-NULL result tuples obtained over time for two runs each. The completion times for the "up" and "down" plans were 9 minutes and 15 minutes, respectively.



(a) Completion rate for different fetch rules **(b)** Worker answer rate for different fetch rules **(c)** Completion rate for different query plans

Figure 5: Experimental Results

In both cases, the cost of query was about $2.20. In general, the predicate pull-up transformation makes Deco query plans invoke fetch rules more "speculatively," which results in more parallelism. More parallelism brings down latency, as we see in Figure 5(c). On the downside of pull-up, speculative fetch rule invocations can lead to higher overall cost. In this particular scenario, we did not observe a significant cost increase because the worker answers for the fetch rule language $\Rightarrow$ name were quite accurate, and the speculative invocations of name $\Rightarrow$ capital were not wasted.

The results reveal that, much like conventional query optimization, the placement of operators is crucial for plan performance, and in addition there are interesting interactions between fetch rules and other operators. Of course, the ability to explore alternative plans hinges on a principled framework for query optimization.

7. RELATED WORK

The prior work related to Deco fits into one of the following categories. We describe each category in turn.

Crowdsourcing and Databases: There has been recent interest in the database community in using crowdsourcing as part of database query processing [10, 14, 24, 27]. The work on CrowdDB [14] is perhaps the most similar to the one presented in this paper. Overall, Deco opts for more flexibility and generality, while CrowdDB made some fixed choices at the outset to enable a simpler design. A detailed comparison between the two systems can be found in Appendix B. We describe the key differences between Deco and CrowdDB here:

- **Interfaces:** While CrowdDB provides a limited set of methods to get data from the crowd, Deco gives the schema designer the flexibility to use very expressive and general fetch rules. As we saw in Figure 5(a)–(b), the reverse and combined fetch rules offer significant performance improvements over basic fetch rules.
- **Cleansing:** Deco stores raw data rather than cleansed data, in order to enable general resolution functions that vary their output based on newly added data. CrowdDB opts for a simpler design wherein a fixed number of humans are consulted for each unknown value, and then the cleansed data is stored.
- **Design:** Deco makes fewer assumptions about the schema: for instance, CrowdDB enforces the presence of primary keys while Deco does not. (As an example, in our restaurant schema, there is no primary key since we allow restaurants to serve multiple cuisines.)
- **Principled:** As far as we know, Deco is the first system to have a precisely defined semantics based on valid instances.

Qurk [24] is a workflow system that implements declarative crowdsourcing, unlike Deco which is a database. (Like Deco, however, Qurk may retain outcomes of prior tasks for reuse or fitting classifiers with the aim of reducing cost, using an underlying storage engine.) Since Qurk uses crowdsourcing primarily as part of its operators (filtering, joins), it is not as general in the kind of data it can obtain from the crowd.

Our prior work [27] outlines a wide variety of challenges in integrating crowdsourcing within a database. In this paper, we build on this work to design a principled and flexible data model and query processor to meet these challenges. Reference [10], while also dealing with crowdsourcing focuses mainly on the challenges underlying user feedback for information integration rather than a system for general declarative crowdsourcing.

Crowdsourcing Interfaces: There are multiple companies that provide marketplaces where users can post tasks and workers can find and attempt them. Mechanical Turk [3], oDesk [4], and Samasource [5] are three such companies. Yet other companies, such as CrowdFlower [2] and ClickWorker [1] act as intermediaries allowing large businesses and corporations to not have to worry about framing and posting tasks directly to crowdsourcing marketplaces.

Crowdsourcing Programming Libraries: Turkit [22] and HProc [18] are programming libraries designed to allow programmers to interface with MTurk. These libraries support a procedural approach where the programmer needs to specify all the tasks, combine their answers, orchestrate their execution and so on, whereas we advocate a declarative approach where the DBMS is responsible for these goals and also performs transparent optimizations.

Prior work in Databases: There has been prior work in the database area on expensive predicate or user-defined function (UDF) evaluation as part of query processing [11, 17]. While calls to crowdsourcing services can be viewed as UDFs, UDF evaluations are much simpler than crowdsourcing calls, disallowing direct application of prior work in this area: (a) UDF evaluations return the same "correct" answer every time they are run. (b) Apart from the computational cost of a UDF, there is no monetary cost. (c) There is only one way to evaluate a UDF, and two UDFs cannot be combined. (d) There is no advantage to evaluating multiple UDFs on different items at the same time. The work on WSQ-DSQ [15] is more relevant than the others in this area since it allows UDF calls to be issued "in parallel", and we do leverage the asynchronous query processing elements from WSQ-DSQ in our system.

There has been a lot of prior work over the last decade on uncertain databases with systems such as Trio [33], MystiQ [8], MayBMS [6] and PrDb [32]. In Deco, we have made a conscious decision not to expose uncertainty as a first-class construct to the end user. Instead, we provide explicit control to the schema designer on how uncertainty should be resolved for each attribute group. This approach is not only simpler (and similar to the forms of uncertainty resolution being used in crowdsourcing today), but it is more flex-

ible, and also eliminates the computationally intractable aspects of dealing with uncertainty correlations between different attributes. Intuitively, our intention is less to manage and query uncertain data as it is to fetch and clean data.

Our data model is also related to the field of Global-As-View (GAV) data integration [19]. In particular, our fetch rules can be viewed as binding patterns or capabilities of sources [13, 20, 21], and the conceptual schema can be viewed as a mediated database. However, the properties and behavior of humans and sources are different, necessitating different query processing techniques: First, humans, unlike external data sources, need to be compensated monetarily. Second, subsequent invocations of the same fetch rule improves the quality of results since multiple humans would be consulted, unlike external sources where repeating the same query returns the same results. Third, different invocations of fetch rules can be done in parallel (i.e., many humans may address tasks in parallel), while external data sources have a single point of access.

Crowdsourcing Algorithms: Recently, there has been an effort to design fundamental algorithms for crowdsourcing. In other words, the algorithms, such as sorting, filtering and searching use human workers to perform basic operations, such as comparisons, evaluating a predicate on an item, or rating or ranking items. This work is orthogonal to ours and can be used as part of the query processor of our Deco system. So far, there have been papers on filtering items [26], finding max [16], searching in a graph [28], and sorting and joins [23].

8. CONCLUSIONS

We presented Deco, a system for declarative crowdsourcing. Deco offers a practical and principled approach for accessing crowd data and integrating it with conventional data. We believe that our notions of fetch and resolution rules provide simple but powerful mechanisms for describing crowd access methods. We also believe that our split conceptual/raw data model (top and bottom parts of Figure 1), together with our "Fetch-Resolve-Join" semantics, yield an elegant way to manage the data before and after cleansing. Our query processor utilizes a novel push-pull execution model to allow worker requests to proceed in parallel, a critical aspect when dealing with humans and other high-latency external data sources. While our current Deco prototype does not *yet* perform sophisticated query optimization, our design and system do provide a solid foundation to study the cost, latency and accuracy tradeoffs that make crowdsourced data such an interesting challenge.

Acknowledgements: We thank Richard Pang for helping us run the experiments in this paper.

9. REFERENCES

[1] ClickWorker. *http://clickworker.com*.
[2] CrowdFlower. *http://crowdflower.com*.
[3] Mechanical Turk. *http://mturk.com*.
[4] ODesk. *http://odesk.com*.
[5] Samasource. *http://samasource.com*.
[6] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions. In *ICDE*, 2007.
[7] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.
[8] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. Mystiq: a system for finding more answers by using probabilities. In *SIGMOD*, 2005.
[9] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.
[10] X. Chai, B. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD*, 2009.
[11] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.
[12] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing systems on the world-wide web. *CACM*, 2011.
[13] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimi- zation in the presence of limited access patterns. In *SIGMOD*, 1999.
[14] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, 2011.
[15] R. Goldman and J. Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.
[16] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won? dynamic max discovery with the crowd. In *SIGMOD*, 2012.
[17] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, 1993.
[18] P. Heymann and H. Garcia-Molina. Turkalytics: analytics for human computation. In *WWW*, 2011.
[19] M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, 2002.
[20] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.
[21] C. Li and E. Chang. Query planning with limited source capabilities. In *ICDE*, 2000.
[22] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP*, 2009.
[23] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. In *VLDB*, 2012.
[24] A. Marcus, E. Wu, S. Madden, and R. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
[25] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, 2008.
[26] A. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *SIGMOD*, 2012.
[27] A. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR*, 2011.
[28] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: it's okay to ask questions. In *VLDB*, 2011.
[29] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. In *VLDB*, 2012.
[30] H. Park, A. Parameswaran, and J. Widom. Query processing over crowdsourced data. Technical report, Stanford Infolab, 2012.
[31] A. Quinn and B. Bederson. Human computation: a survey and taxonomy of a growing field. In *CHI*, 2011.
[32] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *ICDE*, 2007.
[33] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR*, 2005.

APPENDIX

A. JOIN ORDER

In this section, we show that all left outerjoin orders with the anchor table as the first operand followed by the dependent tables (in any order) produce identical results. Let $\rtimes$ denote the left outerjoin operator.

THEOREM A.1. *Let A be the anchor table, and let $D_1, D_2, \ldots, D_n$ be the dependent tables. Let π denote a permutation of $1 \ldots n$. For all permutations π, the following expression E_π evaluates to the same result:*

$$E_\pi = (\ldots((A \rtimes D_{\pi(1)}) \rtimes D_{\pi(2)}) \ldots \rtimes D_{\pi(n)})$$

PROOF. Let $t = (a_1, a_2, \ldots, a_m)$ be a tuple in A. Let $t(D_{\pi(i)}) = \Pi_{\delta_{\pi(i)}}(\{t\} \rtimes D_{\pi(i)})$, where $\delta_{\pi(i)}$ is the set of all dependent attributes in $D_{\pi(i)}$. Note that none of the $t(D_{\pi(i)})$ nor t share any attributes. Now, let us consider

$$E^t_\pi = (\ldots((\{t\} \rtimes D_{\pi(1)}) \rtimes D_{\pi(2)}) \ldots \rtimes D_{\pi(n)})$$

which is intuitively the contribution of t to E_π. Note that

$$E_\pi = \biguplus_{t \in A} E^t_\pi \tag{1}$$

Now, we have:

$$\begin{aligned}
E^t_\pi &= (\ldots((\{t\} \rtimes D_{\pi(1)}) \rtimes D_{\pi(2)}) \ldots \rtimes D_{\pi(n)}) \\
&= (\ldots((\{t\} \times t(D_{\pi(1)})) \rtimes D_{\pi(2)}) \ldots \rtimes D_{\pi(n)}) && \text{(2a)} \\
&= (\ldots(\{t\} \times t(D_{\pi(1)}) \times t(D_{\pi(2)})) \ldots \rtimes D_{\pi(n)}) && \text{(2b)} \\
&\vdots \\
&= \{t\} \times t(D_{\pi(1)}) \times t(D_{\pi(2)}) \times \ldots \times t(D_{\pi(n)}) && \text{(2c)}
\end{aligned}$$

Equality 2a holds by definition of left outerjoin. Equality 2b holds because the left outerjoin is unaffected by the cross-product with $t(D_{\pi(1)})$ since no attributes are shared between $t(D_{\pi(1)})$ and $D_{\pi(2)}$. Thus, $(\{t\} \times t(D_{\pi(1)})) \rtimes D_{\pi(2)} = (\{t\} \rtimes D_{\pi(2)}) \times t(D_{\pi(1)}) = \{t\} \times t(D_{\pi(2)}) \times t(D_{\pi(1)})$. Subsequent equalities hold for similar reasons.

Since $\times$ is associative and commutative, E^t_π (Expression 2c) is identical, independent of the permutation π. Since this argument applies for each t in A, using Equation 1, the argument holds for E_π itself. In other words, the final result is independent of the permutation π. □

B. COMPARISON WITH CROWDDB

Since the CrowdDB data model is the closest one to ours, in this section we perform a more thorough comparison between the CrowdDB and Deco data models. Specifically, we show that the Deco data model is more expressive than the CrowdDB data model as described in [14].

B.1 Data Model

We first provide some examples of data model constructs Deco supports that CrowdDB does not, and then show that the CrowdDB data model as presented in [14] is an instance of the Deco data model.

- Unlike CrowdDB, Deco does not require that every (conceptual) relation has a primary key. For instance, in our Restaurant example, there is no (nontrivial) key, since there could be multiple cuisines for each name.
- Deco allows various kinds of fetch rules, and fetch rules may be added or removed at any time. Thus, a Deco relation can have fetch rule that adds more tuples, along with another fetch rule that fills in missing values. A CrowdDB table on the other hand only allows either fetching an entire tuple (CrowdTables) or filling in missing values (CrowdColumns).
- Deco allows user-defined resolution functions, for instance, *average-of-3* for rating. Or, the schema designer could specify a more sophisticated resolution function, such as average of 3 to 10 values with standard deviation smaller than 1. Also, the resolution functions are allowed to output multiple values after resolution, for instance, *dupElim* for cuisine. On the other hand, CrowdDB only allows a majority vote for each attribute with a fixed maximum number of votes.
- Deco retains all crowdsourced data in raw tables, while CrowdDB discards crowdsourced data and populates the resolved values in its relations. If a new resolution function is to be applied or more crowdsourced data is provided, Deco can easily recompute the new resolved value on-demand, while CrowdDB will need to start afresh.

We now show that the CrowdDB data model is an instance of the Deco data model. CrowdDB supports regular relations, however, some relations may be designated as *CrowdTables*. For those relations that are not CrowdTables, some attributes may be designated as *CrowdColumns*.

CrowdColumns: The regular relations in CrowdDB that contain CrowdColumns may be expressed in Deco as conceptual relations with the same schema. We first designate the set of all attributes that are not CrowdColumns to be the anchor attribute group of that conceptual relation. (Note that in CrowdDB one of these attributes will be the primary key.) We may then express each CrowdColumn as a dependent attribute group. Since CrowdDB uses a majority vote to resolve multiple answers, we use the same majority vote as the resolution function for each dependent attribute. The anchor attribute group has duplicate elimination as the resolution function. The fetch rule used in CrowdDB to populate values in CrowdColumns is a single one from all the anchor attributes to all the dependent attributes.

CrowdTables: We can represent CrowdTables once again as a single conceptual relation in Deco. We designate the key attribute as the anchor attribute. Each of the other attributes is designated as a separate dependent attribute. As before, the resolution function for each of these dependent attributes is a majority vote. The resolution function is set to be duplicate elimination for the anchor attribute. In CrowdDB, entire tuples may be crowdsourced (which is equivalent to a fetch rule $\varnothing \Rightarrow$ all-attributes) or some attributes whose majority has already been reached are provided, and the other attributes are crowdsourced (which can be captured by a fetch rule $D \Rightarrow D'$, where $D \cup D' =$ all-attributes).

Thus, the Deco data model captures the CrowdDB data model as an instance. We now discuss other issues that differ in the two system designs.

B.2 Data Manipulation

Since CrowdDB does not store any of the unresolved raw data explicitly, all relations are visible to an end user. CrowdDB thus allows users to insert new tuples into these relations and to update and delete existing tuples.

In Deco, conceptual relations may be materialized on demand during query processing. As a result, we only allow updates and deletes to tuples in the raw tables. Inserts, on the other hand, are permitted on conceptual relations. They are handled just like fetch rules: the raw tables that have attributes mentioned in the insert statement are populated with new tuples.

Of course, a schema designer may wish to allow updates and deletes to conceptual relations. They would need to be translated to the raw tables, akin to update and delete rules in standard relational databases [9].

B.3 Query Language and Semantics

CrowdDB uses standard relational queries with an optional LIMIT clause to specify the maximum number of tuples required in the output. CrowdDB ensures that no unresolved values are output (i.e., all the CrowdColumns of each tuple must be filled in before the tuple is output).

While LIMIT is related to our AtLeast construct, it has somewhat different semantics. AtLeast ensures a lower bound on the number of output tuples given budget restrictions, while LIMIT ensures an upper bound on the number of output tuples. Let us consider a single-relation query. If there are no current tuples in a CrowdDB relation, then the query processor will return an empty result since the LIMIT clause does not place a lower bound on the number of output tuples. Additionally, if there are many tuples in a CrowdDB relation with all of their values already resolved, CrowdDB will still only return the number of tuples specified in the LIMIT clause (instead of all of them, even though returning all the tuples would not require any further crowdsourcing calls).

CrowdDB also supports two constructs CROWDEQUALS (to allow human-evaluated equality predicates) and CROWDORDER (to allow human-based sorting). All comparisons are cached in CrowdDB. In Deco, we can support these constructs using a similar approach. We create two conceptual relations: CrowdCompare contains all triples of the form (a, b, V) such that V corresponds to the boolean value of the expression $a < b$, according to a human worker. CrowdEquals contains all triples of the form (a, b, V) such that V corresponds to the boolean value of the expression $a = b$, according to a human worker. Resolution rules can be applied on these tables to improve the quality of the comparisons. Then, a query involving CROWDEQUALS can be translated into a query that joins with the CrowdEquals table. Similarly, a query involving CROWDORDER can be translated into one with an operator that implements the ORDER-BY semantics using CrowdCompare.