Deco: A System for Declarative Crowdsourcing

Hyunjung Park
Stanford University
hyunjung@cs.stanford.edu

Richard Pang
Stanford University
rrpang@cs.stanford.edu

Aditya Parameswaran
Stanford University
adityagp@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

Deco is a system that enables declarative crowdsourcing: answering SQL queries posed over data gathered from the crowd as well as existing relational data. Deco implements a novel push-pull hybrid execution model in order to support a flexible data model and a precise query semantics, while coping with the combination of latency, monetary cost, and uncertainty of crowdsourcing. We demonstrate Deco using two crowdsourcing platforms: Amazon Mechanical Turk and an in-house platform, to show how Deco provides a convenient means of collecting and querying crowdsourced data.

1. INTRODUCTION

Crowdsourcing [3, 6] 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, allowing the end user to 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 how to obtain the best crowdsourced data, and how it must be integrated with other data.

We propose to demonstrate *Deco* (for "declarative crowdsourcing"), a system that answers declarative queries posed over stored relational data together with data gathered on-demand from the crowd. 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). Deco's query language is a simple extension to SQL, and expresses the constraints necessary for crowdsourcing. In a companion full paper [5], we describe in detail the Deco prototype, which uses a novel push-pull hybrid query execution model to overcome the limitations of the traditional iterator model in the crowdsourcing setting. In [5] we also illustrate how the flexibility of the Deco data model provides many alternative query execution plans with different performance characteristics.

There have been other recent systems designed to support declarative crowdsourcing [4]. Deco's generality and semantics enable a variety of query processing alternatives not available to these systems, and we propose to highlight these alternatives in our demonstration scenarios.

Proceedings of the VLDB Endowment, Vol. 5, No. 12

2. DATA MODEL AND QUERY LANGUAGE

We begin by illustrating each of the Deco data model components using a running example, then we describe our query language and semantics. For more details see [5].

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 are interested in querying a conceptual relation of countries:

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

Each dependent attribute-group (single attributes in this case) is enclosed within square brackets. In this example, the anchor attribute name identifies a country, while language and capital are independent properties of the country. We assume that each country can have one or more languages but exactly one capital.

Raw Schema: Deco is designed to use a conventional RDBMS as its back-end. The *raw schema*—for the 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. 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(name)
CountryD1(name, language)
CounrtyD2(name, capital)

Fetch Rules: *Fetch rules* allow the schema designer to specify how data can be obtained from humans. A fetch rule takes the form $A_1 \Rightarrow A_2 : P$, where A_1 and A_2 are sets of attributes from one conceptual relation (with $A_1 = \varnothing$ permitted), and P is a *fetch procedure* that implements access to human workers. (P might generate HITs (Human Intelligence Tasks) to Amazon 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.

- $\varnothing \Rightarrow$ name: Ask for a country name, inserting the obtained value into raw table CountryA.
- name $\Rightarrow$ capital: Ask for a capital given a country name, inserting the resulting pair into table CountryD2.
- language $\Rightarrow$ name: Ask for a country name given a language, inserting the resulting country name into table CountryA, and inserting the name-language pair into CountryD1.

There are many more possible fetch rules for our example. In the demonstration scenario (Section 4), we use slightly more sophisticated fetch rules such as name $\Rightarrow$ language,capital and language $\Rightarrow$ name,capital. For a full description of the allowable fetch rules, see [5].

Resolution Rules: Suppose we've obtained values for our raw tables, but we have inconsistencies in the collected data. We use *resolution rules* to cleanse the raw tables—to get values for conceptual relations that are free of inconsistencies. For each conceptual relation, the schema designer can specify a resolution rule $\varnothing \rightarrow A : f$ for the anchor attributes A treated as a group, and one resolution rule $A \rightarrow D : f$ for each dependent attribute-group D. In $\varnothing \rightarrow A : f$, 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 three resolution rules:

- $\varnothing \rightarrow$ name : *dupElim*(1,1)
- name $\rightarrow$ language : *dupElim*(4,2)
- name $\rightarrow$ capital : *majority*(3)

where the resolution functions are defined as:

- *dupElim*(n, m): Given a set S of values, if $|S| \geq n$, retain all distinct values appearing at least m times in S, otherwise return NULL.
- *majority*(n): Given a set S of values, if $|S| \geq n$ and a value v appears at least $\lceil \frac{n+1}{2} \rceil$ times, return v, otherwise return NULL.

Recall that each country can have one or more languages but exactly one capital. Resolution function *dupElim*(4,2) for a given country produces distinct language values that appear at least twice in four or more answers for the country. Resolution function *majority*(3) produces the majority of three or more capital answers for a given country.

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 the *Fetch-Resolve-Join* sequence: (1) *Fetching* additional data for the raw tables using fetch rules; this step may be skipped. (2) *Resolving* inconsistencies using resolution rules for each of the raw tables. (3) *Outerjoining* the resolved raw tables to produce the conceptual relations.

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 for the tuples in the query result.

Query Language and Semantics: A Deco query Q is a SQL query over the conceptual relations. The answer to Q must be the result of evaluating Q over some (logical) valid instance of the database.

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 at all. The problem is that often times this "correct" query result will also be 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 without NULL attributes.

For further details and examples of Deco's data model and query semantics, see [5].

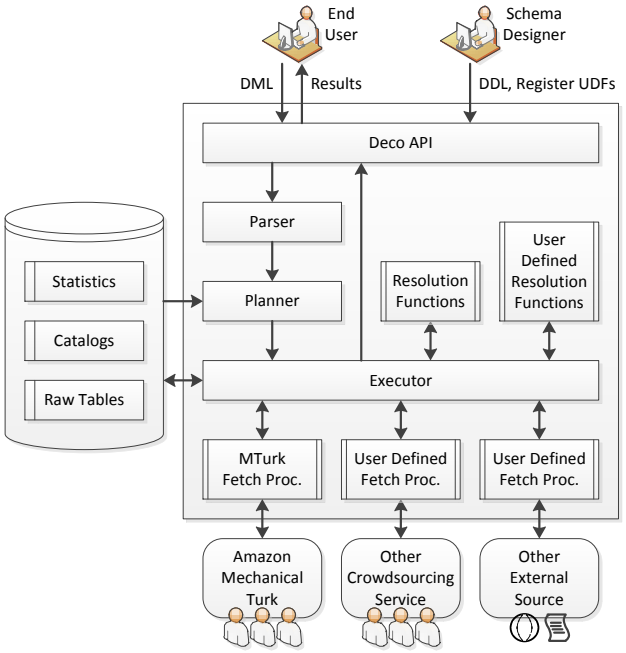


Figure 1: Deco Architecture

3. SYSTEM OVERVIEW

We implemented our Deco prototype in Python with a SQLite back-end. Currently, the system supports DDL commands to 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 1) and query processing model.

3.1 Architecture

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.

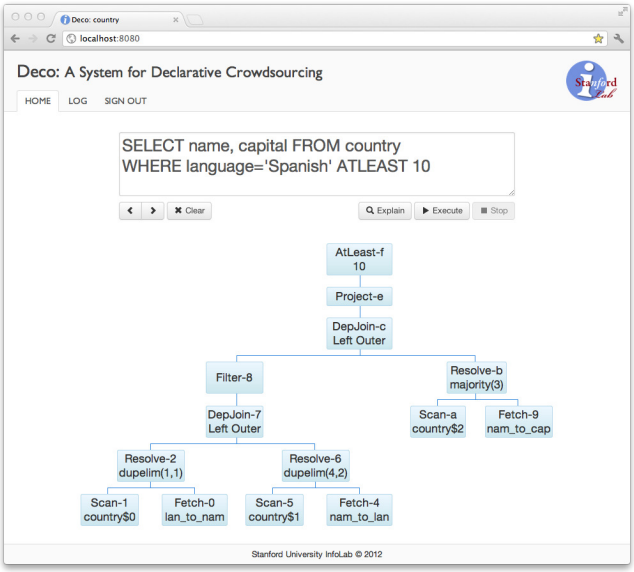
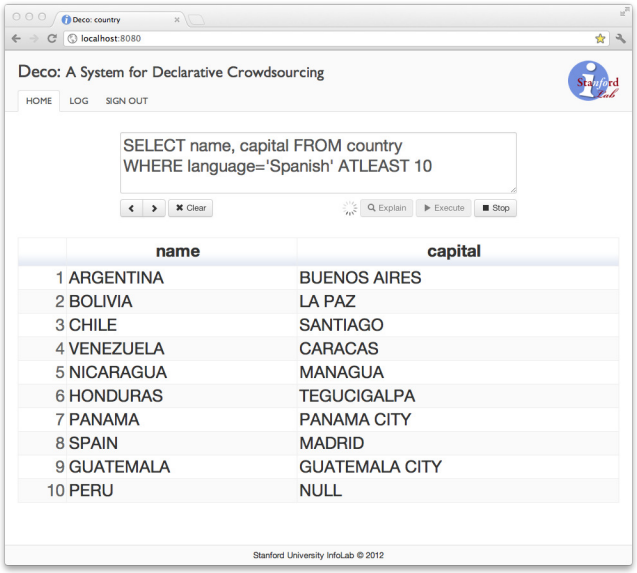


Figure 2: Query Plan Visualization



	name	capital
1	ARGENTINA	BUENOS AIRES
2	BOLIVIA	LA PAZ
3	CHILE	SANTIAGO
4	VENEZUELA	CARACAS
5	NICARAGUA	MANAGUA
6	HONDURAS	TEGUCIGALPA
7	PANAMA	PANAMA CITY
8	SPAIN	MADRID
9	GUATEMALA	GUATEMALA CITY
10	PERU	NULL

Figure 3: Query Execution

3.2 Query Processing

The Deco prototype uses a novel *Push-Pull Hybrid Execution Model* to implement the Deco semantics. This new model has the following significant differences with respect to the traditional iterator model:

- Not only may each query operator pass its output tuples to its parent operator, but it can also remove or modify output tuples that were passed to the parent operator previously, analogous to incremental view maintenance [2]. For example, the output of the resolution function *majority* can change when new tuples are added to the raw table.
- The *getNext* calls are asynchronous. A parent operator does not expect an immediate response to its *getNext* request; the child operator will respond later when a new tuple becomes available. This built-in asynchrony allows Deco to ask multiple questions to the crowd in parallel without having to wait for individual crowd answers.
- Operators can pass a new tuple to their parent operators without receiving any *getNext* requests. For example, if a fetch rule name ⇒ language,capital is invoked based on the need for a language value, data may as a side effect be inserted in the raw table for capital.

Deco queries are executed in two phases, referred to as *materialization* and *accretion*. In the materialization phase, the "current" result is materialized using the current contents of the raw tables without invoking additional fetches. If this result does not meet the AtLeast constraint, the accretion phase invokes fetch rules to obtain more results. This second phase maintains the result incrementally as fetch rules complete, and invokes more fetches as necessary until the AtLeast constraint is met.

4. DEMONSTRATION

4.1 Basic Scenario

In the introductory scenario, we set up our simple countries database schema using a series of DDL commands, and we execute a query to demonstrate Deco's functionality and usability.

Create Resolution Functions: To create the conceptual relation of countries in Deco, we first need to create the resolution functions. For example, the following command creates the resolution function *majority*(n) in Python:

```
CREATE FUNCTION majority AS
'def majority(tuples, n):
    for t in tuples:
      if tuples.count(t) > 0.5 * max(n, len(tuples)): return [t]
    if len(tuples) >= n: return []'
```

The first argument tuples contains the set S of values to be resolved.

Create Table: Once the resolution functions are created, we can create the conceptual relation Country:

```
CREATE TABLE country (name varchar USING dupelim(1,1),
  [language varchar USING dupelim(4,2)],
  [capital varchar USING majority(3)])
```

Create Fetch Rules: We create three fetch rules: (a) language ⇒ name, (b) name ⇒ language, and (c) name ⇒ capital. As an example, the fetch rule name ⇒ capital is created with the following command:

```
CREATE FETCHRULE nam_to_cap ON country
(name TO capital) USING mturk WITH '{''reward'': 0.05,
''question'': ''What is the capital city of ${name}?''}'
```

When this fetch rule is invoked, the question template "What is the capital city of ${name}?" is instantiated by replacing ${name} with an actual country name.

Execute Query: Now, we ask for ten Spanish-speaking countries along with their capital cities using the following query:

```
SELECT name, capital FROM country
WHERE language='Spanish' ATLEAST 10
```

Before executing the query, we can visualize the query plan in the Deco web interface by clicking the Explain button (Figure 2). Figure 4 shows the same query plan with a few additional details. We will refer back to this plan, calling it a "reverse" configuration because the fetch rule language ⇒ name obtains value for the anchor attribute given values for the dependent attribute.

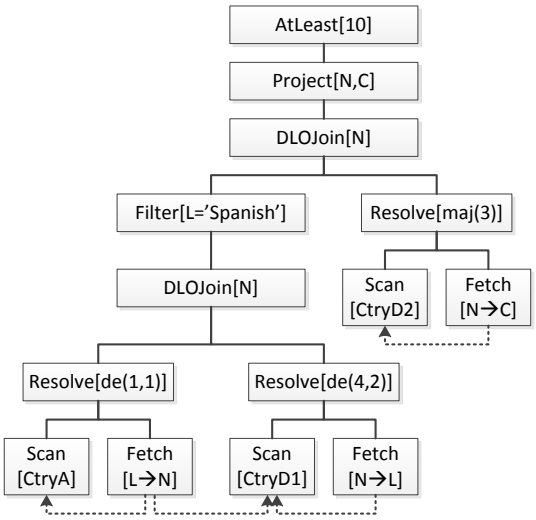


Figure 4: Reverse Configuration

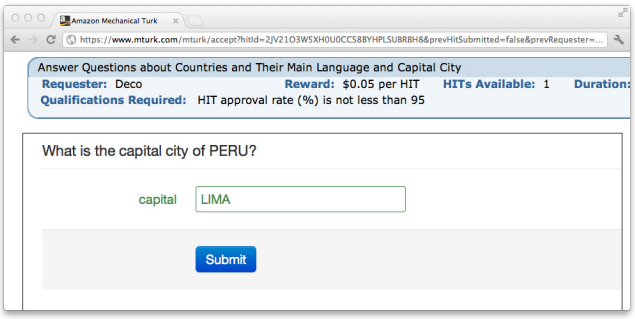


Figure 5: Mechanical Turk Worker Interface

Finally, we start executing the query by clicking the Execute button. Since the raw tables are empty, Deco invokes all three fetch rules to produce the result. As the query is running, the Deco web interface displays the up-to-date result (Figure 3). In this figure, our query is almost done except for one last NULL capital value for Peru. At this moment, Figure 5 shows a screenshot of a Mechanical Turk worker submitting an answer for the capital of Peru. To give an engaging demonstration, the audience will be invited to submit answers while the query is running.

4.2 Improving Execution Time and Cost

The second scenario primarily focuses on the flexibility of Deco fetch rules and its performance implications, by running the same query under different fetch configurations.

Create Additional Fetch Rules: Deco's data model supports a wide variety of fetch rules. Next we set up a "hybrid" configuration by creating two new fetch rules (a) language ⇒ name,capital and (b) name ⇒ language,capital. Notice that these fetch rules get two pieces of information at once.

Execute Query: Figure 6 shows Deco's chosen query plan when we use the hybrid configuration. Unlike Figure 4, the fetch rule name ⇒ language,capital is shared by two different parent operators. As a result, Deco invokes name ⇒ language,capital whenever it needs either a language or capital value for a given country.

We observe that Deco asks about 40% fewer questions of humans in the hybrid configuration compared to the original "reverse" configuration. In many cases, the execution time also improves as the number of questions decreases. Moreover, assuming we offer the same monetary compensation for each fetch rule, the total monetary cost is lower as well. In general, this scenario depicts that 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.

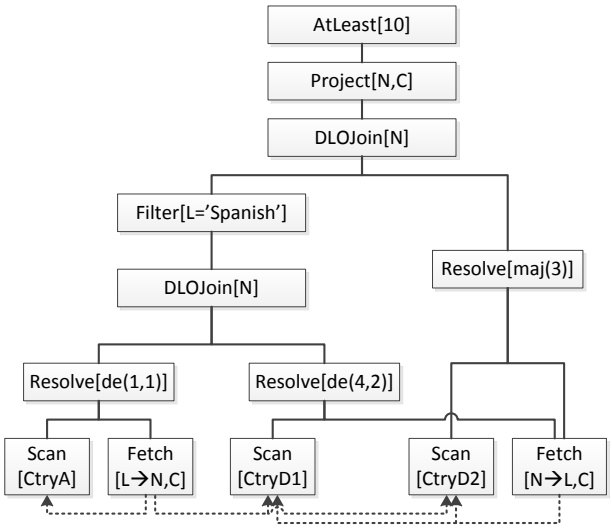


Figure 6: Hybrid Configuration

4.3 Improving Data Quality

The third and final scenario demonstrates how Deco's resolution rules are useful for controlling data quality.

Change Resolution Rules: Suppose our query result indicates that the capital of Spain is not Madrid but Barcelona, and we realize that asking just three different workers for the capital of a country is not enough. We change the resolution function for capital:

```
ALTER TABLE country
ALTER COLUMN capital USING majority(5)
```

Execute Query: If we execute our same query again but without "ATLEAST 10", we get ten Spanish-speaking countries with NULL capital values, since the current contents of the raw table CountryD2 populated by previous queries are insufficient to resolve capital values. When we add "ATLEAST 10", Deco collects more capital values for each country to produce ten result tuples with better quality data.

5. REFERENCES

[1] Mechanical Turk. *http://mturk.com*.

[2] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

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

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

[5] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: Declarative crowdsourcing, http://ilpubs.stanford.edu:8090/1015/. Technical report, Stanford Infolab, 2012.

[6] A. Quinn and B. Bederson. Human computation: a survey and taxonomy of a growing field. In *CHI*, pages 1403–1412, 2011.