An Overview of the Deco System: Data Model and Query Language; Query Processing and Optimization

Hyunjung Park[†], Richard Pang[‡], Aditya Parameswaran[†],
Hector Garcia-Molina[†], Neoklis Polyzotis[*], Jennifer Widom[†]
[†]Stanford University [‡]Google, Inc. [*]UC Santa Cruz

ABSTRACT

Deco is a comprehensive system for answering declarative queries posed over stored relational data together with data obtained on-demand from the crowd. In this overview paper, we describe Deco's data model, query language, and system prototype, summarizing material from earlier papers. Deco's data model was designed to be general, flexible, and principled. Deco's query language extends SQL with simple constructs necessary for crowdsourcing, and has a precise semantics for arbitrary queries. Deco's query execution engine and cost-based query optimizer incorporate many novel techniques to address the limitations of traditional query processing techniques in the crowdsourcing setting. Query processing is guided by the objective of minimizing monetary cost and reducing latency.

1. INTRODUCTION

Crowdsourcing [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 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 figures out how best to obtain crowdsourced data, and how to integrate it with existing data.

Deco (for "declarative crowdsourcing") [12, 13, 14, 15] is a system that answers declarative queries posed over stored relational data together with data gathered on-demand from the crowd. Since humans are involved in generating answers, query results from Deco will not be instantaneous, but may be more comprehensive and useful than those from a traditional system.

In [12], we defined a data model for Deco that is general, flexible, and principled. We also defined a query language for Deco as a simple extension to SQL with constructs necessary for crowdsourcing. In [14], we described Deco's query plans, and how Deco executes a given query plan to minimize monetary cost while reducing latency. In [15], we described how Deco chooses the best query plan to answer a query, in terms of estimated monetary cost.

This paper provides an overview of Deco's data model and query language (Section 2), system prototype (Section 3), query execution (Section 4), and query optimization (Section 5), summarizing material from [12, 13, 14, 15]. Related work is covered in Section 6, and we conclude with future directions in Section 7.

2. DATA MODEL AND QUERY LANG.

We begin by illustrating each of the Deco data model components using a running example, then we describe Deco's query language and semantics.

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

As a simple running example, suppose our users are interested in querying a conceptual relation with information about countries:

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

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

Raw Schema: Deco is designed to use a conventional RDBMS as its back-end. The *raw schema*—for the data tables actually stored in the underlying RDBMS—is derived automatically from the conceptual schema, and is invisible to both the schema designer and end-users. Raw tables contain existing data obtained by past queries or otherwise present in the database, and are extended as new data is obtained from the crowd, enabling seamless integration of conventional data and crowdsourced data.

For each relation R in the conceptual schema, there is one *anchor table* containing the anchor attributes, and one *dependent table* for each dependent attribute-group; dependent tables also contain anchor attributes.

In our example, we have the raw schema:

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

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 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 attributes $A_1 \cup A_2$. The schema designer also specifies a fixed monetary cost for each fetch rule, to be paid to human workers once they complete the fetch rule.

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

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

There are many more possible fetch rules for our example [12].

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, where A' is a subset of the anchor attributes ($A' \subseteq A$). 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 for A'.

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

- $\varnothing \rightarrow$ country : *dupElim*
- country $\rightarrow$ language : *majority-of-3*
- country $\rightarrow$ capital : *majority-of-3*

Resolution function *dupElim* produces distinct country values. Resolution function *majority-of-3* produces the majority of three or more language (or capital) answers for a given country. We assume a "shortcutting" version that can produce an answer with only two values, if the values agree. (We will see in Section 4 how our query processor incorporates shortcutting to fetch the fewest required values.) Note any resolution functions are permitted, not just the types used here for illustration.

Data Model Semantics: The semantics of a Deco database is defined as a potentially infinite set of *valid instances* for the conceptual relations (capturing an open-world assumption). A valid instance is obtained by a *Fetch-Resolve-Join* sequence: (1) *Fetching* additional data for the raw tables using fetch rules; this step may be skipped. (2) *Resolving* inconsistencies 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 only those portions of the conceptual data needed to produce the query result are actually materialized.

Query Language and Semantics: A Deco query Q is simply a SQL query specified over the conceptual relations. Deco's query semantics dictate that the answer to Q must represent the result of evaluating Q over some (logical) valid instance of the database.

One valid instance of the database can be obtained by resolving and joining the current contents of the raw tables, without invoking any further fetch rules. Thus, it appears Q could always be answered correctly without consulting the crowd at all. The problem is that this "correct" result may be very small, or even empty. To retain our straightforward semantics over valid instances, while still forcing answers to contain some amount of data, we add to our query language a "MinTuples n" constraint: The result of Q must be over some valid instance for which the answer has at least n tuples without NULL attributes. (As future work we will address other constraints such as "MaxCost c" and "MaxTime t" [14].)

3. SYSTEM OVERVIEW

We implemented our Deco prototype in Python with a PostgreSQL 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. Deco's overall architecture is shown in Figure 1.

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 user defined fetch procedures and resolution functions. Us-

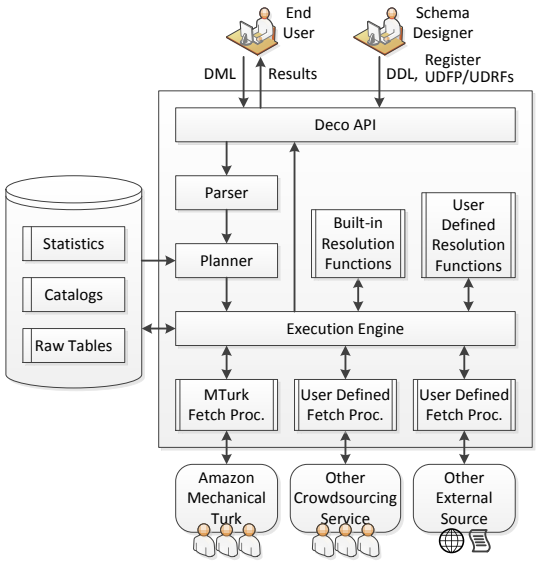


Figure 1: Deco Architecture

ing the Deco API, we built a command-line interface as well as a web-based graphical user interface. The GUI executes queries (Figure 2), visualizes query plans, and shows log messages in real-time.

When the Deco API receives a query, at a very high level the overall process of parsing the query, choosing the best query plan, and executing the chosen plan is similar to a traditional database system. However, there are many significant differences in the details. For example, the query planner translates declarative queries posed over the conceptual schema to execution plans over the raw schema, and the query execution engine is not aware of the conceptual schema at all. Moreover, to obtain data from the crowd, the query execution engine invokes fetch procedures, and the raw data is cleaned by invoking resolution functions. We describe how the system executes a selected plan in Section 4, and how the system constructs and selects a plan in Section 5.

4. QUERY EXECUTION

Given a Deco query plan with "MinTuples n" constraint, our primary goal in query execution is to produce at least n result tuples while minimizing monetary cost. In addition, our secondary goal is to reduce latency by exploiting parallelism when accessing the crowd. Achieving both goals translates to the following overall objective for query execution: *Maximize parallelism while fetching data from the crowd (to reduce latency), but only when the parallelism will not waste work (to minimize monetary cost).*

4.1 Challenges and Approach

To meet the objective while respecting Deco's semantics, we incorporated several novel techniques into Deco's query execution engine.

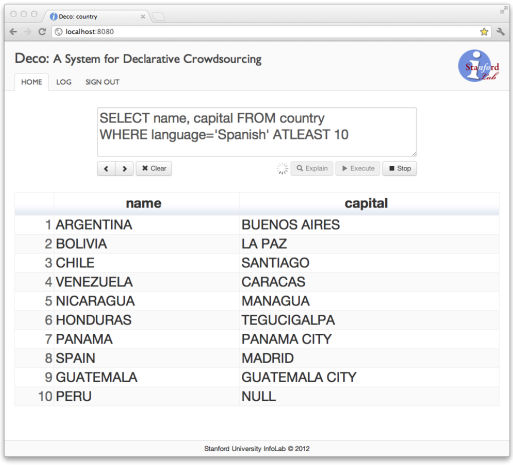
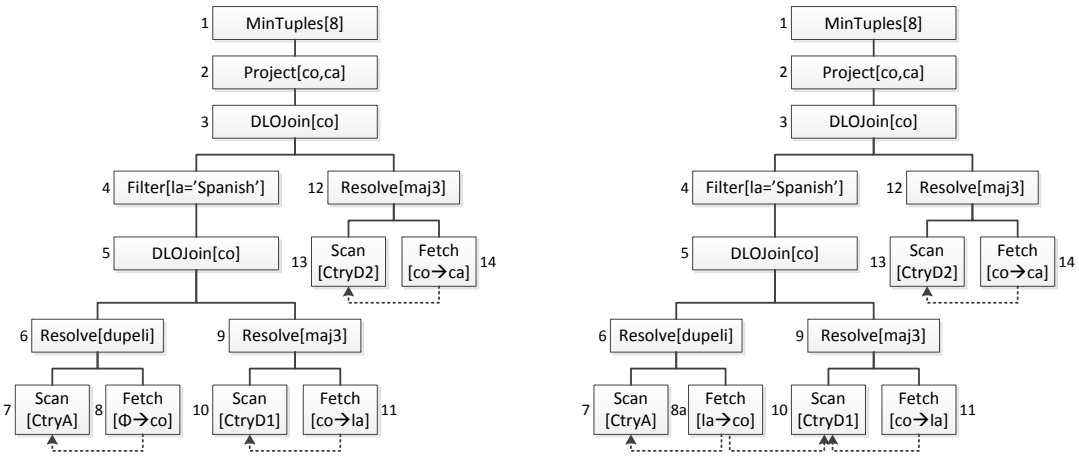


Figure 2: Deco User Interface Screenshot

Hybrid Execution Model: To reduce latency, it is important for Deco's query execution engine to exploit parallelism when accessing the crowd. However, the traditional *iterator* model cannot enable this kind of parallelism because *getNext* calls do not return until data is provided. In Deco, query operators do not expect an immediate response to a *getNext* (pull) request; the child operator will respond whenever a new output tuple becomes available (push). This built-in asynchrony in query plan execution ultimately allows Deco to ask multiple questions to the crowd in parallel, without having to wait for individual crowd answers. Also, this model enables us to precisely choose the right degree of parallelism: too much parallelism might waste work (and therefore increase monetary cost), while too little increases latency.

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

Two Phase Execution: To ensure that fetches are issued only if the raw tables do not have sufficient data, Deco executes queries in two phases. In the *materialization* phase, the "current" result is materialized using



(a) Basic **(b)** Reverse

Figure 3: (Physical) Query Plans

the existing contents of the raw tables without invoking fetches. If this result does not meet the MinTuples constraint, the *accretion* phase invokes fetch rules to obtain more results. This second phase extends the result incrementally as fetches complete, and invokes more fetches as necessary until the MinTuples constraint is met.

Dynamic Fetch Prioritization: In certain cases, minimizing monetary cost is especially difficult because existing data can make some fetches more profitable than others. Deco incorporates an algorithm that identifies specific "good" fetches, i.e., those that help satisfying the MinTuples constraint with less cost. Individual query operators do not always have enough information to choose the good fetches, so our approach is to invoke more fetches than needed in parallel, but prioritize them so the better fetches are more likely to complete first (thus minimizing cost). When enough data has been obtained, outstanding fetches are canceled.

4.2 Basic Query Plan

Figure 3a shows one possible query plan for the following query on the example database from Section 2:

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

In this example, we use three fetch rules: $\varnothing \Rightarrow$ country (operator 8), country $\Rightarrow$ language (operator 11), and country $\Rightarrow$ capital (operator 14). Abbreviations in the plan should be self-explanatory. Deco-specific query operators used in this plan are as follows:

- The *Fetch* operator corresponds to a fetch rule $A_1 \Rightarrow A_2 : P$. It receives values for A_1 from its parent and invokes procedure P. It does not wait for answers, so many procedures may be invoked in parallel. When values for $A_1 \cup A_2$ are returned by P, they are sent to one or more *Scan* operators that work with the *Fetch* operator. *Scan* operators insert the new tuples into raw tables and also pass them up to their parent.
- The *Resolve* operator corresponds to a resolution rule $A_1 \rightarrow A_2 : f$. It receives from its child tuples containing attribute values for $A_1 \cup A_2$. It applies function f based on groups of tuples with the same A_1 value, and passes up resolved values for $A_1 \cup A_2$.
- The *DLOJoin* (for Dependent Left Outerjoin [8]) operator is similar to a relational index nested-loop join. It receives attribute values from its outer child, which it passes to its inner child to obtain additional attributes that constitute join result tuples. In our plans, *DLOJoin* always obtains anchor attributes from its outer and dependent attributes from its inner.
- The *MinTuples* operator determines when the answer is complete.

We describe the case where there are no existing tuples in any of the raw tables. First, the root operator sends eight *getNext* requests to its child operator (based on "MinTuples 8"). These requests propagate down the left (outer) side of the joins, and eventually invoke fetch rule $\varnothing \Rightarrow$ country eight times, without waiting for answers. At this point, there are eight outstanding fetches in parallel.

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

Until the MinTuples constraint is met, the query plan invokes additional fetches as needed to exploit as much

parallelism as possible while minimizing monetary cost. For example, if the two instances of fetch rule country ⇒ language for a given country return two different language values, the plan invokes another instance of the same fetch rule to obtain the third language value. Likewise, as soon as a resolved language value for a certain country turns out to not be Spanish, the plan invokes a new instance of fetch rule ∅ ⇒ country. For countries whose resolved language value is Spanish, the plan obtains capital values for the country, in parallel with other fetches similarly to how language values were obtained. Once the MinTuples constraint is met, the result tuples are returned to the client.

4.3 Alternative Plan

Here we describe another plan for our query. In Section 5.2 we will describe the entire plan search space.

Suppose predicate language='Spanish' is very selective. If we use the query plan in Figure 3a, even obtaining a single answer could be expensive in terms of monetary cost and latency, because we are likely to end up fetching many countries and languages that do not satisfy the predicate. Instead, we can use the "reverse" fetch rule language ⇒ country underneath *Resolve* operator 6 to obtain countries with a certain language, rather than random countries. Figure 3b shows a query plan that uses this approach. Note the only change from Figure 3a is operator 8a. Whenever *Fetch* operator 8a receives a *getNext* request from its parent, it invokes the fetch rule language ⇒ country with Spanish as the language value. When completed with a country value, *Fetch* operator 8a adds tuples to both CountryA and CountryD1 via *Scan* operators 7 and 10, and the two added tuples propagate up the plan separately.

5. QUERY OPTIMIZATION

The goal of Deco's cost-based query optimizer is to find the best query plan to answer a query, where "best" means the least estimated monetary cost across all possible query plans.

5.1 Challenges and Approach

Deco's query semantics, execution model, and optimization objective introduce several new challenges in plan costing and enumeration.

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

Statistical Information: Deco's cost model requires statistical information about both existing data and new data. For existing data, we use the statistical information maintained by the back-end RDBMS. For data obtained from the crowd, we primarily rely on information provided by the schema designer and/or end-user. We require a *selectivity factor* to be provided by the schema designer for each resolution function, specifying how many output tuples are produced on average for each input tuple. For filters, we allow the end-user to provide a selectivity factor; if none is provided we resort to heuristic default constants.

Estimating Cardinality and Database State: As Deco executes a query, it also changes the state of the database because it stores newly crowdsourced data. The estimated cardinality and cost of a subplan depend in part on the expected end-state of the database, which is a property of the entire plan, not just the subplan. Thus, Deco's cardinality estimation algorithm estimates cardinality and end-state simultaneously, using a top-down recursive process. As a result, Deco's cardinality estimation is holistic, making traditional plan enumeration techniques break down.

5.2 Search Space

To find the best plan for a given query, Deco's query optimizer explores a space of alternative query plans. Under the Fetch-Resolve-Join semantics, Deco's plan alternatives are defined by selecting a join tree and a set of fetch rules: A join tree corresponds to an order of join evaluation, while one fetch rule must be assigned to obtain additional tuples for each raw table in a plan. We construct a logical query plan based on a selected join tree, and expand the logical plan into a set of physical plans by selecting fetch rules.

Suppose we are interested in finding the best query plan to answer the example query from Section 4.2. We assume there are four available fetch rules: ∅ ⇒ country, language ⇒ country, country ⇒ language, and country ⇒ capital. In this example, there are two possible join orders:

(CountryA ⋈ CountryD1) ⋈ CountryD2
(CountryA ⋈ CountryD2) ⋈ CountryD1

Also, for each join order, there are two possible selections of fetch rules since we can use either ∅ ⇒ country or language ⇒ country for raw table CountryA. Thus, our plan search space has four query plans including the two plans in Figure 3.

5.3 Cost Estimation

Deco's cost estimation algorithm takes as input a physical plan and statistics about data, and produces as output the estimated cost in dollars. It turns out estimating the monetary cost amounts to estimating cardinality for

each *Fetch* operator, because no Deco query operators except *Fetch* cost money. Note that in Deco we estimate cardinality of a subplan or operator as the total number of output tuples expected in order to obtain a query result with a sufficient number of tuples. We describe cardinality estimation using examples below.

Continuing our example in Section 5.2, let us assume the selectivity factors of predicate language='Spanish' and resolution function *majority-of-3* are 0.1 and 0.4, respectively. Also, we assume each fetch costs \$0.05, for all fetch rules. Finally we assume the raw tables are initially empty.

Cardinality estimation is a top-down recursive process, where each operator calls its subplan(s) with a set of predicates, and the number of tuples it needs from its subplan that satisfy the predicates. For the basic plan in Figure 3a, the process begins with the root operator calling its child with no predicates and a requirement of eight tuples. Eventually *Fetch* operator 8 receives predicate language='Spanish' and a requirement of eight tuples. Since the selectivity of the predicate is 0.1, *Fetch* operator 8 returns an estimated cardinality of 80. As the recursion unwinds, *Resolve* operator 6 and *Filter* operator 4 return estimated cardinality of 80 and 8, respectively. Because *Resolve* operators 9 and 12 have selectivity of 0.4, *Fetch* operators 11 and 14 are expected to produce $80/0.4 = 200$ and $8/0.4 = 20$ tuples, respectively. Thus, the estimated cost is $\$0.05 \times (80 + 200 + 20) = \$15.00$.

For the reverse plan in Figure 3b, cardinality estimation begins and proceeds exactly the same as above. However, fetch rule language $\Rightarrow$ country at *Fetch* operator 8a is instantiated with left-hand side value "Spanish", so the operator expects the predicate to be satisfied by all fetched data, and returns an estimated cardinality of 8. *Filter* operator 4 does not apply the selectivity 0.1 again and returns the same estimated cardinality of 8. Thus, both *Fetch* operators 11 and 14 are expected to produce 20 tuples, and the estimated cost is $\$0.05 \times (8 + 20 + 20) = \$2.40$.

In our example, the reverse plan in Figure 3b is much cheaper than the basic plan in Figure 3a, and is actually the best plan in the search space. In [15], we compare our estimated costs against the actual costs using Mechanical Turk, and found out that our estimation is reasonably accurate with a mean percentage error of 14%.

6. RELATED WORK

Several recent systems have proposed a declarative approach to leverage crowdsourced data [2, 4, 5, 7, 9, 10, 11]. CrowdDB [7] is perhaps the most similar to Deco in terms of the data model and query language; however, Deco opts for more generality and flexibility, thus requiring the novel query processing techniques described in this paper. (A detailed comparison between the two systems can be found in [12].) Qurk [10, 11] supports crowd-powered operations on relational data, and reference [10] studied how to optimize its crowd-powered sort and join operations by using worker interfaces tailored for the specific operations.

7. FUTURE WORK

One important avenue of future work is to incorporate adaptive query processing techniques into the Deco prototype. Due to the simplified statistical information about crowdsourced data and the long-running nature of Deco queries, the "optimize-then-execute" paradigm may not always yield the best possible execution strategy in our setting.

Among many other avenues of future work, we are interested in extending the Deco prototype to support more general SQL queries beyond Select-Project-Join; for example, order-by and group-by are certainly useful SQL constructs in any environment. Furthermore, we plan to incorporate alternatives to MinTuples, such as MaxCost and MaxTime. With these alternatives, end-users can specify a monetary or time budget to answer a query, while maximizing the number of result tuples.

8. REFERENCES

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

[2] S. Ahmad, A. Battle, Z. Malkani, and S. D. Kamvar. The jabberwocky programming environment for structured social computing. In *UIST*, 2011.

[3] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[4] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *WWW*, 2012.

[5] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Declarative platform for data sourcing games. In *WWW*, 2012.

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

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

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

[9] S. R. Jeffery, L. Sun, M. DeLand, N. Pendar, R. Barber, and A. Galdi. Arnold: Declarative crowd-machine data integration. In *CIDR*, 2013.

[10] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. In *VLDB*, 2012.

[11] A. Marcus, E. Wu, S. Madden, and R. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.

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

[13] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. In *VLDB*, 2012. Demonstration.

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

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