CrowdDB: Query Processing with the VLDB Crowd

Amber Feng
AMPLab, UC Berkeley
amber.feng@berkeley.edu

Michael Franklin
AMPLab, UC Berkeley
franklin@cs.berkeley.edu

Donald Kossmann
Systems Group, ETH Zurich
donaldk@inf.ethz.ch

Tim Kraska
AMPLab, UC Berkeley
kraska@cs.berkeley.edu

Samuel Madden
CSAIL, MIT
madden@csail.mit.edu

Sukriti Ramesh
Systems Group, ETH Zurich
ramess@student.ethz.ch

Andrew Wang
AMPLab, UC Berkeley
awang@cs.berkeley.edu

Reynold Xin
AMPLab, UC Berkeley
rxin@cs.berkeley.edu

ABSTRACT

Databases often give incorrect answers when data are missing or semantic understanding of the data is required. Processing such queries requires human input for providing the missing information, for performing computationally difficult functions, and for matching, ranking, or aggregating results based on fuzzy criteria. In this demo we present CrowdDB, a hybrid database system that automatically uses crowdsourcing to integrate human input for processing queries that a normal database system cannot answer.

CrowdDB uses SQL both as a language to ask complex queries and as a way to model data stored electronically and provided by human input. Furthermore, queries are automatically compiled and optimized. Special operators provide user interfaces in order to integrate and cleanse human input. Currently CrowdDB supports two crowdsourcing platforms: Amazon Mechanical Turk and our own mobile phone platform. During the demo, the mobile platform will allow the VLDB crowd to participate as workers and help answer otherwise impossible queries.

1. INTRODUCTION

Relational database systems are the de-facto standard to store and query data. Still, many queries cannot be answered correctly and require human interactions. One obvious situation where existing systems produce wrong answers is when information that is required for answering the question is missing. For example, the query:

```
SELECT abstract FROM paper
WHERE title = "CrowdDB";
```

will return an empty answer if the *paper* table at that time does not contain a record for "CrowdDB". The database system will also give a wrong answer if the record was entered incorrectly, say, as "CrowDB", or if the same real-world entity has multiple representations. This latter "entity resolution" problem can even arise, if no error was made during data entry. Finally, the database is incapable of answering queries that require semantic understanding of existing data. For example, the following query, which tries to find the 10 papers with the most novel ideas:

```
SELECT title FROM paper
ORDER BY novel_idea LIMIT 10;
```

will fail, unless the novelty of the ideas has been previously obtained and stored.

For all these queries, relational database systems require the user to look at the data and manually collect or correct them. Even worse, the relational database provides little or no support during this process. Typically, the whole burden of the task lands on the shoulders of a single user, forcing him/her to look or enter possibly thousands of records manually.

CrowdDB[3] aims at answering those types of queries by automatically leveraging the knowledge of people through crowdsourcing. Microtask crowdsourcing platforms such as Amazon's Mechanical Turk (AMT)[1] enable people to access the crowd in an on-demand fashion. The platforms provide the infrastructure, connectivity, and payment mechanisms that allow hundreds of thousands of people to perform paid work on the Internet. CrowdDB uses these platforms by extending a traditional query engine with a small number of operators that solicit human input. Considering the previous sample queries, correctly answering these queries relies on two capabilities of people:

Finding new data - Relational database systems are based on the "Closed-World Assumption": information that is not in the database is considered to be false or non-existent. The crowd, on the other hand, aided by tools such as search engines and reference sources, are usually capable of finding information that they do not have readily at hand.

Comparing data - People are skilled at making comparisons that are difficult or impossible to encode in computer algorithms. For example, if given the right context, it is easy for a person to tell whether "CrowDB" and "CrowdDB" refer to the same entity. Likewise, people can easily compare items using the derived attributes, for example comparing the level of novelty of ideas presented in various papers (assuming that they understand the field).

For CrowdDB, we develop crowd-based query operators for finding and comparing data, while relying on the traditional relational query operators to do the heavy lifting for bulk data manipulation and processing. We also introduce some minimal extensions to the SQL data definition and query languages, called CrowdSQL, to enable the generation of queries that involve human computation.

Our approach has four main benefits. First, by using SQL we create a declarative interface to the crowd. We strive to maintain SQL semantics so that developers are presented with a known computational model. Second, CrowdDB provides Physical Data Indepen-

Proceedings of the VLDB Endowment, Vol. 4, No. 12

dence for the crowd. Application developers can write SQL queries without having to worry which operations will be performed by the computers and which by the crowd. Existing SQL queries can be run on CrowdDB, and in many cases will return more complete and correct answers than if run on a traditional DBMS. Third, as we discuss in subsequent sections, user interface design is a key factor in enabling questions to be answered by people. Our approach leverages schema information to support the automatic generation of effective user interfaces for crowdsourced tasks. Finally, by using the crowd, we enable hundreds of workers to work in parallel on obtaining new or comparing data. This allows us to free the user/developer from a very time-consuming task.

We currently have a working prototype of CrowdDB that we plan to demonstrate. CrowdDB implements all key features for an end-to-end study: it compiles CrowdSQL queries, optimizes the query with rule-based heuristics, creates tasks and collects answers from the crowd at run-time, caches the answers, and performs simple quality control before finally showing the results to the user.

In addition to CrowdDB, we have developed a local-area mobile crowdsourcing platform which allows tasks to be posted to users in a specific geographic area. CrowdDB can compile tasks to run on this platform as well as on Amazon Mechanical Turk (AMT). We plan to run the demo over the course of the conference. We will use CrowdDB to post a number of tasks to our mobile crowd service, allowing people at the conference to complete tasks using a mobile application. We will use CrowdDB to issue several conference-specific tasks, such as rating and commenting on talks, recommending restaurants, and noting interesting events. We will show how CrowdDB tasks are compiled onto the crowdsourcing platforms, and demonstrate other features of CrowdDB, such as our query language. This demo will differ from other crowd-sourced databases, such as Qurk [5] at SIGMOD 2011, by not only showing the different architecture and query language of CrowdDB, but also highlighting our mobile platform and the ability of CrowdDB to compile for it.

In the remainder of this demonstration proposal, we first sketch the syntax and semantics of CrowdSQL. Afterwards we present an architectural overview of CrowdDB and outline the query execution process. More details about the system such as the quality control mechanisms or the optimization heuristics, however, have to be omitted due to space constraints and can instead be found in [3]. Finally, in Section 4 we describe in more detail how we plan to demonstrate CrowdDB at VLDB and how we encourage the VLDB crowd to participate during the demo.

2. CrowdSQL

CrowdSQL is our small extension to SQL to support use cases that involve missing data and subjective comparisons. This approach allows application programmers to write (Crowd) SQL code in the same way as they do for traditional databases; i.e., in most cases, developers need not be aware that their code involves crowdsourcing. In the following we sketch the syntax and semantics of CrowdSQL with help from a series of running examples. Furthermore, the examples will also be part of the demo and hopefully result in some interesting facts about the VLDB conference itself.

2.1 Incomplete Data

Incomplete data can occur in two flavors: First, specific attributes of tuples could be crowdsourced. Second, entire tuples could be crowdsourced. We capture both cases by adding a special keyword, *CROWD*, to the SQL DDL, as shown in the following two examples.

EXAMPLE 1 (Crowdsourced Column) The following *Talk* table contains a list of technical presentations at VLDB 2011. The *abstract* and *nb_attendees* attributes are marked as crowdsourced. It is relatively easy to populate the table with the titles of the talks from the conference website. We cannot, however, determine the number of people in the audience but can use crowdsourcing to fill in this attribute. Similarly, the abstracts are often not provided on the conference website but likely available elsewhere.

```
CREATE TABLE Talk (
  title STRING PRIMARY KEY,
  abstract CROWD STRING,
  nb_attendees CROWD INTEGER );
```

EXAMPLE 2 (Crowdsourced Table) This example models a *NotableAttendee* table as a *crowdsourced table*. Here, *notable attendees* are recognized by other attendees as "well-known" researchers in the community. It is assumed that the database captures none or only a subset of the notable attendees of a talk. In other words, CrowdDB will expect that additional notable attendees may exist and possibly crowdsource more attendees if required for processing specific queries. We can query this table, for example, to sense new trending topics.

```
CREATE CROWD TABLE NotableAttendee (
  name STRING PRIMARY KEY,
  title STRING,
  FOREIGN KEY (title) REF Talk(title) );
```

In order to represent values in crowdsourced columns that have not yet been obtained, CrowdDB introduces a new value to each SQL type, referred to as *CNULL*. *CNULL* is the CROWD equivalent of the *NULL* value in standard SQL. *CNULL* indicates that a value should be crowdsourced when it is first used.

CrowdDB supports any kind of SQL query on CROWD tables and columns; for instance, joins between two CROWD tables are allowed. Furthermore, the results of these queries are as expected according to the (standard) SQL semantics. What makes CrowdSQL special is that it incorporates crowdsourced data as part of processing SQL queries. Specifically, CrowdDB asks the crowd to instantiate *CNULL* values if they are required to evaluate predicates of a query or if they are part of a query result. Furthermore, CrowdDB asks the crowd for new tuples of CROWD tables if such tuples are required to produce a query result. More details on the semantics can be found in [3].

2.2 Subjective Comparisons

Recall that beyond finding missing data, the other main use of crowdsourcing in CrowdDB is subjective comparisons. In order to support this functionality, CrowdDB has two new built in functions: `CROWDEQUAL` and `CROWDORDER`. `CROWDEQUAL` takes two parameters (an *lvalue* and an *rvalue*) and asks the crowd to decide whether the two values are equal. `CROWDORDER` is used whenever the help of the crowd is needed to rank or order results.

EXAMPLE 3 The following CrowdSQL query asks for the titles of the 10 most favorable presentations.

```
SELECT title FROM Talk
ORDER BY CROWDORDER(p,
"Which talk did you like better")
LIMIT 10;
```

3. CrowdDB PROTOTYPE

Our first CrowdDB prototype implementation is based on the open-source Java database engine H2 [4]. We decided to build on top of a Java database engine as the range of available Java web-frameworks significantly simplifies the integration of web-services and creation of web-based data entry forms. The high-level components of our prototype are shown in Figure 1. CrowdDB answers queries using data stored in local tables when possible, and invokes the crowd otherwise. At the moment, CrowdDB is able to work with two crowdsourcing platforms: Amazon Mechanical Turk and our own mobile crowdsourcing platform. Results obtained from the crowd are always stored in the database for future use.

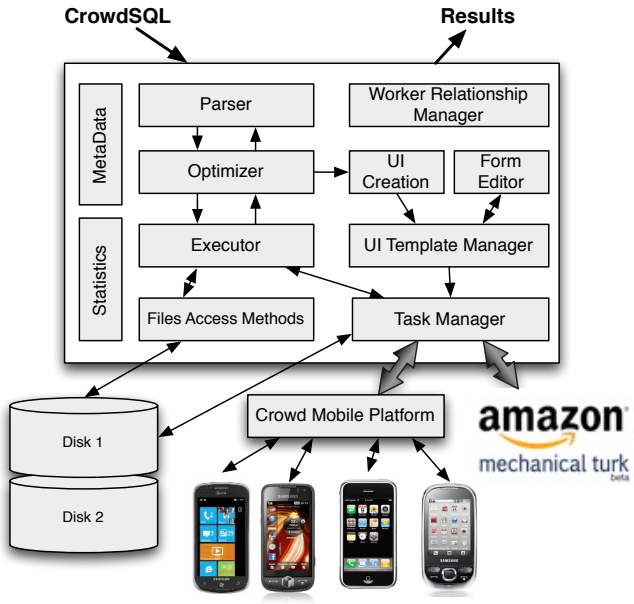

Figure 1: CrowdDB Architecture.

As shown on the left side of the figure, CrowdDB incorporates the traditional query compilation, optimization and execution components from H2. These components are extended to cope with human-generated input as described in Section 3.2. On the right side of the figure are new components that interact with the crowdsourcing platform.

The *Worker Relationship Manager* (WRM) helps to build communities for requesters. Unlike computer processors, crowd workers are not fungible resources and the worker/ requester relationship evolves over time and thus, requires special care. Currently, the WRM component assist the requester with paying workers in time, granting bonuses and reporting and answering worker complaints.

In contrast to physical IO requests, the interface for crowdsourcing data consists of HTML forms and instructions in natural languages. CrowdDB generates the HTML form using the available database schema information. The three components, *UI Creation*, *UI Template Manager*, and*Form Editor*, are responsible for creating, managing and editing user interface templates.

The *Task Manager* provides an abstraction layer that manages the interaction between CrowdDB and the crowdsourcing platforms. It instantiates the user interfaces, makes the API calls to post tasks, assess their status, and obtain results. The Task Manager also interacts with the storage engine to obtain values to pre-load into the task user interfaces and to memorize the results sourced from the crowd.

3.1 User Interface Generation

CrowdDB leverages the available database schema information to automatically generate user interfaces. This generation is a two-step process. At compile-time, the *UI Creation* component creates *templates* to crowdsource missing information from all CROWD tables and all regular tables which have CROWD columns. These user interfaces are HTML templates that are generated based on the CROWD annotations in the schema and optional free-text annotations of columns and tables that can also be found in the schema. All generated templates are centrally managed by the *UI Template Manager*. Furthermore, these templates can be edited by application developers in order to provide additional custom instructions. Finally, at runtime the *Task Manager* instantiates the templates on request of the crowd operators in order to provide a user interface for a concrete tuple or a set of tuples.

Figure 2 shows an example interface based on the schema in Example 1 and the query:

```
SELECT abstract FROM talk
WHERE title = "CrowdDB";
```

for crowdsourcing the missing abstract of the "CrowdDB" talk. The instructions of the HTML ask the worker to enter the missing information for the Table (i.e., *Talk* in this example). In general, user interface templates are instantiated by copying the known field values from a tuple into the HTML form (e.g., "CrowdDB" in this example). Furthermore, all fields of the tuple that have *CNULL* values and are asked by the query become input fields of the HTML form (i.e., *URL* in this example).

3.2 Query Processing

Query plan generation and execution follows largely a traditional approach. In particular, as shown in Figure 1, CrowdDB uses the database parser, optimizer, and runtime system from H2. The main differences are that CrowdDB has additional operators that affect crowdsourcing (in addition to the traditional relational algebra operators found in a traditional database system), and the CrowdDB optimizer includes special heuristics to generate plans with these additional Crowd operators.

3.2.1 Crowd Operators

CrowdDB extends the relational algebra operators (e.g. join, scan) from H2, with a small set of operators that supports crowdsourcing. The basic functionality of all Crowd operators is the same: At runtime, they consume a set of tuples, e.g., *Talk*. Depending on the Crowd operator, crowdsourcing can be used to source missing values of a tuple (e.g., the *abstract* of a *Talk*) or to source new tuples (e.g. new *NotableAttendee*). In addition to the creation of tasks, each Crowd operator consumes and cleanses results returned by the crowd. The current version of CrowdDB has three Crowd operators:

CrowdProbe: This operator crowdsources missing data from CROWD columns and new tuples.

CrowdJoin: This operator implements an index nested-loop join over two tables, at least one of which is marked as crowdsourced.

CrowdCompare: This operator uses crowdsourcing to compare data. It can be used inside another "traditional" operator, such as sorting or predicate evaluation. For example, an operator that implements quick-sort can use CrowdCompare to perform the required binary comparisons.

Note that since human inputs are inherently error prone and diverse in formats, answers from the crowd workers can never be assumed to be complete or correct. The above operators also have majority-vote driven quality control measures built-in.

Figure 2: Mechanical Turk Task.

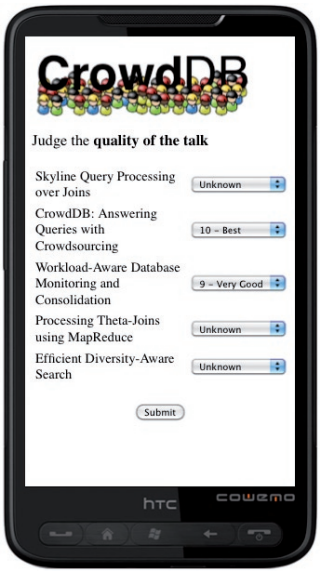

Figure 3: Mobile Task.

3.2.2 Physical Plan Generation

Similar to traditional query processing, the physical plan generation involves three stages. First, CrowdDB generates the *logical plan* by parsing the query. Second, this logical plan is optimized using traditional and crowd-specific optimizations such as predicate push-down. Finally, the logical plan is translated into a *physical plan* which can be executed by the CrowdDB runtime system. As part of this step, Crowd operators and traditional operators of the relational algebra are instantiated.

The current CrowdDB compiler is based on a simple rule-based optimizer. The optimizer implements several essential query rewriting rules such as predicate push-down, stopafter push-down, join-ordering and determining if the plan is bounded. The last optimization deals with the open-world assumption by ensuring that the amount of data requested from the crowd is bounded. Thus, the heuristic first annotates the query plan with the cardinality predictions between the operators. Afterwards, the heuristic tries to re-order the operators to minimize the requests against the crowd and warns the user at compile-time if the number of requests cannot be bounded.

4. DEMONSTRATION DETAILS

We plan an end-to-end demonstration, which visualizes the whole workflow from formulating the query, to compiling and creating the user interfaces, posting the tasks, collecting the answers and finally showing the query result.

Users will have the chance to choose from two crowdsourcing platforms: Amazon Mechanical Turk and our locality-aware mobile crowdsourcing platform [2]. While the first addresses a general crowd with people from all over the world, the latter allows to constrain the workers to the attendees at VLDB. Figures 2 and 3 show screenshots for one of the queries from Section 2 on the Mechanical Turk platform respectively the mobile crowdsourcing platform. The mobile platform can be used without registration and everybody with a mobile phone is welcome to join. This platform will allow us to access the knowledge of the VLDB crowd as part of the demo.

We will use the examples from Section 2 as well as additional conference-specific queries, such as nearby restaurant recommendations, as our main queries for the demo. In addition, we plan to demonstrate the capabilities of CrowdDB to combine existing electronic data with the crowdsourced information. Thus, we will pre-load different tables, such as VLDB talks, restaurants or companies near the VLDB conference location, into CrowdDB. Those tables can then be extended by the audience with crowd columns or be joined with new crowd tables. By means of the individual queries from the audience as well as the examples from Section 2, we will present the query compilation and execution process.

5. REFERENCES

[1] Amazon Mechanical Turk. http://www.mturk.com, 2010.
[2] CrowdDB Mobile Service. http://www.crowddb.org/mobile.
[3] M. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering Queries with Crowdsourcing. In *SIGMOD*, pages 61–72, 2011.
[4] H2 Database Engine. http://www.h2database.com/.
[5] A. Marcus, E. Wu, D. Karger, S. Madden, and R. C. Miller. Demonstration of Qurk: A Query Processor for Human Operators. In *SIGMOD*, pages 1315–1318, 2011.