CrowdFill: A System for Collecting Structured Data from the Crowd[*]

Hyunjung Park
Stanford University
hyunjung@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

CrowdFill is a system for collecting structured data from the crowd. Unlike a typical microtask-based approach, CrowdFill shows an entire partially-filled table to all participating workers; workers collaboratively complete the table by filling in empty cells, as well as upvoting and downvoting data entered by other workers, using CrowdFill's intuitive data entry interface. CrowdFill ensures data entry is leading to a final table that satisfies prespecified constraints, and its compensation scheme encourages workers to submit useful, high-quality work. We demonstrate how CrowdFill collects structured data from the crowd, from the perspective of a user as well as from the perspective of workers.

Categories and Subject Descriptors

H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Computer-supported cooperative work*

Keywords

crowdsourcing; data collection

1. INTRODUCTION

We consider the problem of collecting high-quality structured data from the crowd, while adhering to constraints on the collected data, capping total monetary cost, and keeping latency low. Most previous work on crowdsourcing structured data, e.g., [4, 5, 7], has focused on a *microtask*-based approach: ask workers for specific pieces of data, then assemble the answers into a complete table. We propose to demonstrate our *CrowdFill* system, which takes a different approach. Instead of partitioning data collection into a set of microtasks, CrowdFill shows an entire partially-filled table to all participating workers. Workers are asked to contribute what they know by filling in empty cells, as well as upvoting and downvoting data entered by other workers. Prespecified constraints on table size and entered values ensure that the final table is populated with useful data.

In a companion full paper [6], we describe in detail how CrowdFill provides an intuitive interface for data entry and voting, while allowing simultaneous operations on the same table by different workers, and ensuring data entry is leading to a final table that satisfies the prespecified constraints. As workers perform actions, CrowdFill propagates them to other workers and seamlessly resolves conflicts due to concurrency. We have also devised a compensation scheme that encourages useful work, provides compensation commensurate with a worker's efforts, yields high-quality data, and adheres to a fixed monetary budget.

2. FORMAL MODEL

We begin by describing CrowdFill's formal model using a running example. For more details see [6].

Table Schema: To collect structured data, a CrowdFill user must provide a *table schema* consisting of:

- **Column definitions:** A column name, data type, and optionally a domain (set of allowed values) for each column.
- **Primary key:** One or more key columns that together should uniquely identify each row in the final table. By default, all columns together are a key, i.e., there should be no duplicate rows in the final table.

As a running example, suppose we are interested in collecting information about international soccer players. We use the following schema:

$$\mathsf{SoccerPlayer(name,\ nationality,\ position,\ caps,\ goals)}$$

Columns name and nationality together are the primary key.

Scoring Function: To ensure the quality of collected data, our model allows workers to provide upvotes and downvotes on data. To aggregate votes, the user provides a *scoring function* $f(u_r, d_r)$ where u_r and d_r denote an upvote count and a downvote count, respectively, for a given row r. The intention is for a higher score to indicate that the row is more likely to be correct. Without any votes, a row must always have a zero score. Also, we require that $f(u, d)$ is a monotonically increasing function of u, and a monotonically decreasing function of d. We will see shortly that CrowdFill only allows rows with positive scores to appear in the final table.

For our running example, we'll use a scoring function that implements a "majority of three or more" voting scheme, with shortcutting:

$$f(u_r, d_r) = \begin{cases} u_r - d_r, & \text{if } u_r + d_r \geq 2 \\ 0, & \text{otherwise} \end{cases}$$

Candidate table: A *candidate* table R is a set of rows, where each row r is annotated with its upvote count u_r and downvote count

[*] This work was supported by the NSF (IIS-0904497), the Boeing Corporation, and a KAUST research grant.

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2577029.

d_r. The candidate table can be modified by performing one of the following *primitive operations*. Note *empty*, *partial*, and *complete* describe rows with no values filled in, some values filled in, and all values filled in, respectively.

- insert(r): Insert a new empty row r into R.
- fill(r,A,v): Fill in an empty column A in row $r \in R$ to have value v.
- upvote(r): Upvote a complete row $r \in R$. Increment u_q for each row $q \in R$ whose value is equal to the value of row r.
- downvote(r): Downvote a partial row $r \in R$. Increment d_q for each row $q \in R$ whose value is equal to or a superset of the value of row r.

We will see in Section 3 that worker actions are described in terms of these primitive operations.

In our example SoccerPlayer table, here is one possible candidate table. The columns labeled ↑ and ↓ contain upvote and downvote counts, respectively. Note in particular that candidate tables need not have unique rows with a given primary key; keys are enforced in the final table, defined next.

name	nationality	position	caps	goals	↑	↓
Lionel Messi	Argentina	FW	83	37	2	0
Ronaldinho	Brazil	MF	97	33	3	0
Ronaldinho	Brazil	FW	97	33	2	1
Neymar	Brazil	FW			0	2
Iker Casillas	Spain	GK	150	0	2	0
David Beckham	England	MF	115	17	1	0

Final table: A *final* table S derived from a candidate table R contains each complete row $r \in R$ such that $f(u_r, d_r) > 0$, and $f(u_r, d_r)$ is the highest score of any row with the same primary key as r. Ties are broken arbitrarily, and groups of rows with no positive scores don't contribute to the result. Note a final table respects the primary key constraint by definition.

Based on our example candidate table and scoring function, we obtain the following final table:

name	nationality	position	caps	goals
Lionel Messi	Argentina	FW	83	37
Ronaldinho	Brazil	MF	97	33
Iker Casillas	Spain	GK	150	0

Note Neymar is omitted because the row is not complete, while Beckham is omitted because the score for the row is zero.

Values Constraint: *Constraints* enable CrowdFill users to specify restrictions on the final table of collected data. For the demonstration we focus on *values constraints*, specifying that rows with certain values or combinations of values must be present in the final table. Specifically the user can specify a set T of "initial" rows, which we refer as *template* rows. Template rows can be complete, meaning they should also be present in the final table; they can be partial, with workers expected to fill in missing values; and they can be empty, in which case they are specifying how many additional rows are needed. Note that requiring a certain minimum number of rows—a *cardinality constraint*—is a special case of a values constraint. (In [6] we describe more generalized constraints, e.g., specifying that collected values must satisfy a given predicate.)

Our goal is to obtain a final table S that satisfies the following *values constraint* with template T:

> For each row $t \in T$, there exists a unique row $s \in S$ such that the values in row s are equal to or a superset of the values in row t.

In our running example, if we wish to collect a forward from any country and any player from Brazil and Spain, we would specify the following template:

name	nationality	position	caps	goals
		FW		
	Brazil			
	Spain			

Note the example final table above satisfies the values constraint with this template.

Concurrent operations and convergence: Using the primitive operations defined above, the workers make changes to the candidate table. To support concurrent operations across workers, the CrowdFill *server* has a *master* copy of the candidate table, and each *client* has its own copy, which is initially identical to the master copy. Suppose the worker at client C performs a primitive operation *op*. Client C applies *op* to its own copy of the table, then sends a corresponding *message* m to the server. Once the server receives message m from client C, it first processes m on the master table, then forwards m to all clients except C. Finally all clients except C receive m and process m on their copies of the table. Details of message generation, and the application of operations and messages to a table, are covered in [6].

Clients generate and process messages concurrently, so CrowdFill's protocol must ensure that the result is consistent and correct. We can see easily that the server and all clients process the same set of messages once and only once; however, the server and each client may process the messages in a different order. The primary potential conflict occurs when two different workers fill in empty values in the same row at the same time—either for the same column or for different columns. CrowdFill avoids this conflict by effectively generating a new row for each new column value, instead of filling the value into the existing row; details appear in [6]. The full paper [6] includes a *convergence theorem*, guaranteeing that the server and all clients always have the same candidate table whenever the system "quiesces" (i.e., all generated messages are propagated and processed).

3. SYSTEM OVERVIEW

We have implemented a fully-functional prototype of the CrowdFill system, based on the formal model of Section 2. In this section, we describe CrowdFill's overall architecture and key components.

3.1 Architecture

Figure 1 shows the overall architecture of the CrowdFill system. It consists of several major components: a web interface for users, a front-end server, a back-end server, and one or more worker clients. It also connects with one or more crowdsourcing marketplaces (only one is shown in our diagram), and a database. In the course of data collection, these components interact with each other as follows.

1. Using the web interface, a user sends a table specification to the front-end server to launch data collection.
2. The front-end server creates one or more tasks in the crowdsourcing marketplace.
3. Each worker accepting a task is redirected to the back-end server and establishes a bidirectional persistent connection to the back-end server.
4. Workers perform actions through their data entry interfaces (see Figure 2c), until the back-end server determines that enough data has been collected.

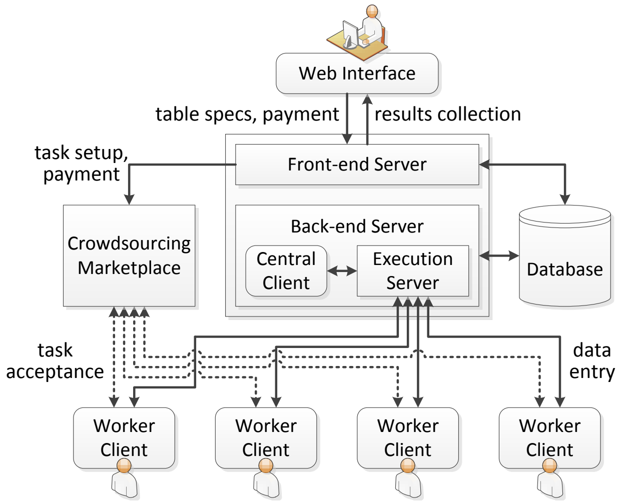


Figure 1: CrowdFill Architecture

5. Using the web interface, the user retrieves collected data from the front-end server and pays workers through the crowdsourcing marketplace.

We built the front-end and back-end servers using Node.js [2], with the Socket.IO library [3] for connections between the back-end server and clients. All metadata and collected structured data are stored in a MongoDB [1] database.

3.2 Data Entry Interface

Each worker client provides its worker with a data entry interface running in a web browser. Through this interface, workers can perform three kinds of actions: fill, upvote, and downvote. These actions correspond to the primitive operations from Section 2 with the same names, with some restrictions on vote operations mentioned below. Note worker clients never generate insert operations. For now suppose that there are enough incomplete rows in the candidate table; we will discuss this issue further in Section 3.3.

Fill action: As shown in Figure 2c, the main part of this interface is an HTML table. This table shows an up-to-date local copy of the candidate table, and it allows workers to fill in empty cells in-place. Filling in an empty cell generates a fill operation as described in Section 2.

Upvote and downvote actions: The rightmost column in the HTML table contains thumb-up and thumb-down icons for each row. Clicking these icons generates upvote and downvote operations, respectively, on the corresponding row.

Although the formal model in Section 2 does not prevent a single worker from contributing multiple upvotes and/or downvotes to the same row, the CrowdFill data entry interface intentionally prohibits this behavior: each worker may provide, directly or indirectly, at most one vote for each row. Thus, upvote and downvote counts represent the number of different workers who approve or disapprove of a given set of values. To further enforce this semantics, when a worker provides the last value that completes a row, that worker automatically upvotes the row, without additional payment.

3.3 Satisfying the Constraints

Recall from Section 2 that our overall goal is to obtain a final table satisfying the values constraints with a template. To guide the final table towards the template, and to minimize wasted work, the CrowdFill system only allows new rows to be inserted into the candidate table by a special client, which we call *CC* (for "Central Client"), in the back-end server (Figure 1). With this approach, workers never need to add rows, and they need not be aware of the constraints, allowing them to simply fill in empty values in existing rows, and cast votes.

The overall objective of the special client CC, as it adds rows to the candidate table, is to keep the table in a state where filling in empty values might produce a final table satisfying the constraint. In [6] we have defined the notion of a row being *probable*—informally, given the current state of the candidate table, a probable row may eventually contribute to the final table. Through special client CC, the CrowdFill system maintains the following invariant at all times, based on the values constraint.

> **Probable Rows Invariant (PRI):** Each template row t corresponds to a unique probable row r in the candidate table such that the values in row r are equal to or a superset of the values in row t.

Maintaining the PRI turns out to be an interesting application of the maximum bipartite matching, and is covered in detail in [6].

3.4 Compensating Workers

Instead of offering fixed compensation for each data entry or voting action, CrowdFill's compensation scheme is based on each worker's overall contribution to the final table. This approach encourages workers to submit useful, high-quality work, while making the total monetary cost more predictable.

CrowdFill allows a user to simply specify a total monetary budget. When the table is complete, the system calculates the final compensation for each worker based on how and when they contributed to the table. Our approach also can take into account variability in the difficulty of providing values for different columns, and the fact that entering new key values can get progressively more difficult as the table fills up. At the same time, CrowdFill needs to keep workers engaged and focused on entering the needed data, so it displays estimated compensation for individual actions during table-filling.

In [6] we define the notion of contribution, describe how final compensation is calculated after the table is complete, and discuss compensation estimation during data collection.

4. DEMONSTRATION

In the demonstration, we will show CrowdFill's data collection process, from the perspective of a user as well as from the perspective of workers. Here we walk through each step and describe the aspects of the system we will highlight.

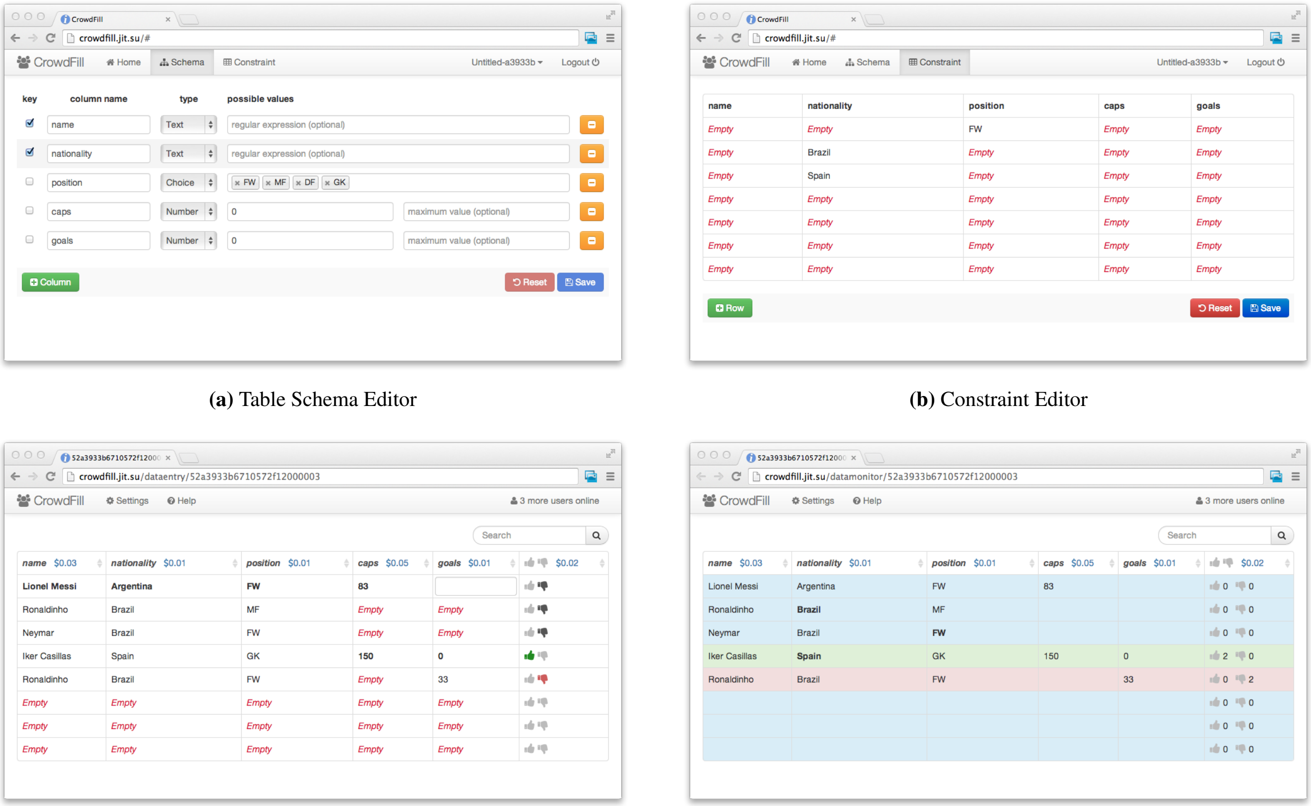
Specifying table: To begin with, we, as a user, provide a table specification, including a table name, instruction for workers, a table schema, a scoring function, and a total monetary budget. We plan to use the example SoccerPlayer schema from Section 2:

SoccerPlayer(name, nationality, position, caps, goals)

Figure 2a shows CrowdFill's Table Schema Editor, with our SoccerPlayer schema filled in.

Once we save the table schema, we populate a template for the values constraint. Figure 2b shows CrowdFill's Constraint Editor. Then, we launch data collection. At this point, data entry interfaces become available to workers, and the user waits until CrowdFill's data collection finishes.

Entering and voting on data: Instead of attempting to hire workers through crowdsourcing marketplaces during the demonstration,



key	column name	type	possible values	
☑	name	Text	regular expression (optional)	
☑	nationality	Text	regular expression (optional)	
☐	position	Choice	× FW × MF × DF × GK	
☐	caps	Number	0	maximum value (optional)
☐	goals	Number	0	maximum value (optional)

(a) Table Schema Editor

name	nationality	position	caps	goals
Empty	*Empty*	FW	*Empty*	*Empty*
Empty	Brazil	*Empty*	*Empty*	*Empty*
Empty	Spain	*Empty*	*Empty*	*Empty*
Empty	*Empty*	*Empty*	*Empty*	*Empty*
Empty	*Empty*	*Empty*	*Empty*	*Empty*
Empty	*Empty*	*Empty*	*Empty*	*Empty*
Empty	*Empty*	*Empty*	*Empty*	*Empty*

(b) Constraint Editor

name $0.03	*nationality* $0.01	*position* $0.01	*caps* $0.05	*goals* $0.01	👍 👎 $0.02
Lionel Messi	**Argentina**	**FW**	**83**		👍 👎
Ronaldinho	Brazil	MF	*Empty*	*Empty*	👍 👎
Neymar	Brazil	FW	*Empty*	*Empty*	👍 👎
Iker Casillas	Spain	GK	**150**	**0**	👍 👎
Ronaldinho	Brazil	FW	*Empty*	33	👍 👎
Empty	*Empty*	*Empty*	*Empty*	*Empty*	👍 👎
Empty	*Empty*	*Empty*	*Empty*	*Empty*	👍 👎
Empty	*Empty*	*Empty*	*Empty*	*Empty*	👍 👎

(c) Data Entry Interface

name $0.03	*nationality* $0.01	*position* $0.01	*caps* $0.05	*goals* $0.01	👍 👎 $0.02
Lionel Messi	Argentina	FW	83		👍 0 👎 0
Ronaldinho	**Brazil**	MF			👍 0 👎 0
Neymar	Brazil	**FW**			👍 0 👎 0
Iker Casillas	**Spain**	GK	150	0	👍 2 👎 0
Ronaldinho	Brazil	FW		33	👍 0 👎 2
					👍 0 👎 0
					👍 0 👎 0
					👍 0 👎 0

(d) Data Monitor

Figure 2: CrowdFill Screenshots

we will act as workers ourselves through a proxy marketplace, entering and voting on data. More interestingly, our proxy marketplace and CrowdFill server will be hosted on the web, so conference attendees will be able to participate as workers in the demonstration. Using multiple worker clients, we will see how a fill action performed at one client propagates to the other clients. We will also demonstrate how conflicting fill actions are seamlessly resolved by duplicating rows. Since data entry interfaces do not show total upvote and downvote counts, we will use CrowdFill's Data Monitor (Figure 2d) to confirm that upvote and downvote actions are being propagated.

Observing the Probable Row Invariant: CrowdFill's Data Monitor (Figure 2d) not only shows the candidate rows along with their vote counts, but it also allows us to observe the status of rows. Each candidate row is color-coded based on the following criteria:

- Red: rows with negative scores.
- Yellow: rows with non-negative scores, but another row with the same primary key has a higher score.
- Blue: rows with zero scores, and no other row with the same primary key has a higher score.
- Green: complete rows with positive scores, and no other row with the same primary key has a higher score.

Both green and blue rows are considered probable, based on our formal definition of probable rows [6].

As workers fill in empty cells and vote on rows, we will observe on the Data Monitor that each row goes through different colors over time. Whenever there are not enough probable rows in the candidate table to satisfy the PRI, CrowdFill inserts some of the template rows into the table, to maintain the PRI.

Getting compensated: To keep workers engaged during data collection, CrowdFill shows estimated compensation for each action, in the table header of the Data Entry Interface (Figure 2c). We will see these estimates stabilize over time as the table becomes complete.

Once enough data has been collected to satisfy the values constraint, CrowdFill terminates data collection and notifies each worker client with the final compensation for the client. At this point we will compare expected compensation (based on the estimates provided during data collection) to the final compensation. In addition, the final status of all rows is broadcast to all clients using our color scheme. We will observe that workers are compensated based on their contribution to the final table.

5. REFERENCES

[1] MongoDB. *http://www.mongodb.org/*.

[2] Node.js. *http://www.nodejs.org/*.

[3] Socket.IO. *http://socket.io/*.

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

[5] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. An overview of the deco system: data model and query language; query processing and optimization. *SIGMOD Record*, 41(4):22–27, 2012.

[6] H. Park and J. Widom. CrowdFill: Collecting structured data from the crowd, http://ilpubs.stanford.edu:8090/1079/. Technical report, Stanford InfoLab, 2013.

[7] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.