CrowdFill: Collecting Structured Data from the Crowd*

Hyunjung Park
Stanford University
hyunjung@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

We present *CrowdFill*, a system for collecting structured data from the crowd. While a typical microtask-based approach would pose specific questions to each worker and assemble the answers, CrowdFill shows a partially-filled table to all participating workers. Workers contribute by filling in empty cells, as well as upvoting and downvoting data entered by other workers. The system's synchronization scheme, based on a careful model of primitive operations, enables workers to collaboratively complete the table without latency overhead. CrowdFill allows the specification of constraints on the collected data, and has mechanisms for resolving inconsistencies. Its compensation scheme takes into account each worker's contribution to the final table, and the varying difficulty of data entry tasks. The paper includes some preliminary experimental results.

Categories and Subject Descriptors

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

Keywords

crowdsourcing; data collection; distributed tasks

1. INTRODUCTION

We consider the problem of collecting high-quality structured data from the crowd, while adhering to constraints on the collected data and total monetary cost, and keeping latency low. Most previous work on crowdsourcing structured data, e.g., [11, 16, 23], has focused on a *microtask*-based approach: ask workers for specific pieces of data, then assemble the answers into a complete table. In this paper, we present 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.

CrowdFill's table-filling approach has several advantages over the microtask-based approach. Each worker can observe and learn from data entered by other workers; this transparency also allows workers to satisfy table-wide constraints, e.g., not entering duplicate values. Table-filling also provides workers with a great deal of flexibility: workers can enter data in arbitrary order and at any granularity, as long as there are empty cells. However, the table-filling approach does have some disadvantages. Human attention and comprehension could make the focused microtask-based approach more efficient and/or scalable, and there are challenges in the data-entry interface when the target table is very large. Likewise, scaling the number of workers may be more effective in the microtask-based approach, since conflicting actions can often be avoided. A thorough comparison of the two approaches is an important topic of future work.

Collecting data from the crowd using the table-filling approach introduces a number of challenges. We need to provide 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, we need to propagate them to other workers and resolve conflicts due to concurrency. We also need to devise 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. Overall, our setting incurs many of the *cost-latency-quality* tradeoffs that tend to characterize the overall area of "human computation" [15].

To obtain high-quality structured data with low latency and cost, while addressing the challenges outlined in the previous paragraph, CrowdFill incorporates several novel techniques, discussed next.

Data entry and voting: CrowdFill displays an evolving partially-filled table, with workers filling in whatever empty cells they like. This approach allows workers to identify those parts of the structured data they can contribute to best (rather than randomly-assigned microtasks), and to enter data at any granularity. To ensure quality of the collected data, CrowdFill also asks workers to upvote and downvote data entered by other workers, again allowing workers to select which data they feel most qualified to endorse or refute. Figure 1 shows a screenshot of CrowdFill's data entry interface, where information about soccer players is being collected.

Real-time collaboration: CrowdFill immediately sends each data entry or vote by a worker to a central server, which propagates those

*This work was supported by the National Science Foundation (IIS-0904497), the Boeing Corporation, and a KAUST research grant.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610503.

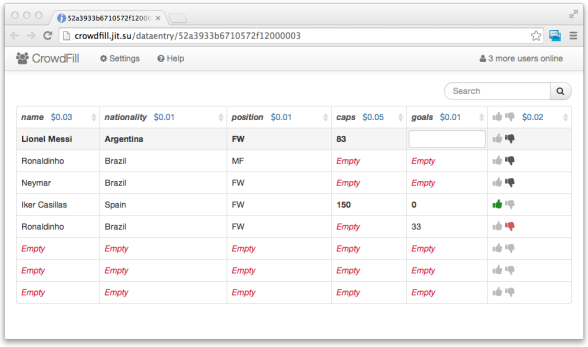

Figure 1: CrowdFill Data Entry Interface

actions to the tables displayed to all other workers. This approach allows workers to enter data in a highly parallel fashion, avoiding the latency overhead when "iterative microtasks" [13] are used. Of course the disadvantage of this approach is the need to merge operations and resolve conflicting operations. We carefully devised our model of table contents and primitive operations to minimize the effects of concurrency: operations are easily merged, and conflicts are resolved seamlessly.

Constrained data entry: Naturally most users of the CrowdFill system will have certain constraints they want the collected data to adhere to. At the most basic level, a minimum number of complete rows is typically desired (eight of them, in Figure 1). Additionally, there may be some values in the table that are prespecified, either to constrain the collected values, or because some data has already been collected earlier or obtained in a different way. CrowdFill supports both *cardinality* and *values constraints*, using a central monitoring client that inserts rows automatically to ensure the final table can satisfy the constraints. We also describe a more general notion of *predicates constraints*, which fall easily into our model but are not present in the current system.

Compensation scheme: 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. 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 (seen in the column headers in Figure 1). The compensation scheme also takes into account varying difficulty of filling in cells—due to inherent difficulty of different types of collected data, as well as increasing difficulty as fewer options remain open.

Implementation and evaluation: The CrowdFill system as described in this paper is fully implemented. Using the system, we have performed some preliminary experiments, focusing on the overall effectiveness of the system for data collection, and an initial quantitative evaluation of our compensation schemes.

1.1 Outline of Paper

The remainder of the paper proceeds as follows.

- In Section 2 we present a formal model for CrowdFill. The model defines "candidate" and "final" tables, it specifies the primitive operations that can be performed by workers, and it explains how those operations propagate to the different copies of the table. The model also includes the specification and interpretation of constraints on the collected data. We conclude Section 2 by addressing how the model is designed to accommodate concurrent operations, and we prove an important consistency result.
- Section 3 covers the implementation of the CrowdFill system. We describe CrowdFill's overall architecture, then explain its server, client, and data-entry interfaces in detail.
- Section 4 details how CrowdFill maintains prespecified constraints during data collection. The goal is to guide worker actions towards a final table that satisfies all constraints, while not introducing any unnecessary restrictions.
- Section 5 describes CrowdFill's compensation scheme. Our overall approach is to ultimately compensate workers for those actions that contribute to the final table, directly or indirectly, while at the same time providing estimated compensation during data collection to keep workers engaged. We describe several possible allocation schemes (compared empirically in Section 6), and describe how compensation is estimated during data collection.
- Our preliminary experimental evaluation is covered in Section 6. We report the effectiveness of CrowdFill's approach for a specific data collection task, we compare our different compensation schemes, and we measure accuracy of compensation estimates.

Related work is covered in Section 7, and we conclude with future directions in Section 8.

2. FORMAL MODEL

We begin in Sections 2.1 and 2.2 by defining our model for tables and primitive operations. We specify the notion of a partially-filled *candidate table* during data collection, the primitive operations that can be performed on candidate tables, and how a *final table* is computed from a candidate table. In Section 2.3 we add *constraints* to the model, so CrowdFill users can provide a minimum required number of rows and restrictions on the collected data values. Finally in Section 2.4 we cover concurrency. We explain how worker actions are transmitted via messages to a central server and other workers, and how conflicts are handled. We prove a strong *convergence* theorem, stating that when all work ceases, all copies of the candidate (and therefore final table) will have the same value.

2.1 Table Specification

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 soccer players who appeared at least 80 times in international matches ("caps"). We use the following schema:

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. Specifically, we take the meaning of score ranges as follows:

- A positive score suggests the row is acceptable.
- A negative score suggests the row is not acceptable.
- A zero score suggests more votes are needed to determine whether the row is acceptable or not.

Without any votes, a row must always have a zero score, i.e., we require $f(0,0) = 0$. Also, we require that $f(u,d)$ is a monotonically increasing function of u, and a monotonically decreasing function of d, i.e., $u_1 \leq u_2$ implies $f(u_1,d) \leq f(u_2,d)$, and $d_1 \leq d_2$ implies $f(u,d_1) \geq f(u,d_2)$. As a default, if the user does not provide a function then $f(u_r,d_r) = u_r - d_r$.

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

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

2.2 Table State and Primitive Operations

In a table, every row can be *empty*, *partial*, or *complete*:

- Empty row: a row with no values
- Partial row: a row with one or more values
- Complete row: a row without empty values

Note a complete row is also a partial row by definition.

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 d_r. The candidate table can be modified by performing one of the following *primitive operations*:

- insert(r): Insert a new empty row r into R, with $u_r = d_r = 0$.
- 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 Sections 3 and 4 that worker actions correspond to fill, upvote, and downvote operations, while insert operations are issued only by the system, to control the number of empty rows in the table. Also note our model incorporates voting at the row level rather than the cell level, enabling workers to vote on combinations of values rather than individual values; see Section 8 for further discussion.

In our example SoccerPlayer table, here is one possible candidate table. Symbols ↑ and ↓ indicate 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
Iker Casillas	Spain	GK	150	0	2	0
David Beckham	England	MF	115	17	1	0
Neymar	Brazil	FW			0	1
Zinedine Zidane					0	0
	France	DF			0	0
					0	0
					0	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 the five incomplete rows are omitted, while Beckham is omitted because the score for the row is zero.

2.3 Constraints

We now describe *constraints*, which enable CrowdFill users to specify restrictions on the final table of collected data. *Cardinality constraints* specify that the final table must contain a minimum number of rows. *Values constraints* specify that rows with certain values or combinations of values must be present in the final table. *Predicates constraints*, not yet implemented in the CrowdFill system, specify that values in the final table must satisfy certain conditions. We will see that cardinality constraints can be considered a special case of values constraints, and values constraints are a special case of predicates constraints. Nevertheless, from both a user and system perspective, it is useful to distinguish the three concepts.

In the remainder of the paper, we sometimes need to distinguish the identifier of a row r from its value. In such cases, we use r to denote the identifier and $\overline{r}$ to denote its value. More generally, we use $\overline{v}$ to denote a vector of values corresponding to a subset of the columns in the schema.

Cardinality constraint: A *cardinality constraint* with nonnegative integer n simply states that the final table S must contain at least $n \geq 0$ rows. Since CrowdFill aims to keep latency and cost as low as possible, typically the final table will contain exactly n rows.

Values constraint: In many cases a user or application may wish to start with a partially-filled table, using the crowd to complete the missing data. A common example is to have a set of values for the keys (soccer player names and nationalities, for example), and use the crowd to fill in missing values (position, caps, goals). Another case is when the table has a set of rows, but the user wishes to crowdsource additional rows. For this type of scenario, 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. Given the latter case, we see that cardinality constraints are in fact a special case of values constraints.

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 $\overline{s} \supseteq \overline{t}$, i.e., the values in row s are equal to or a superset of the values in row t (or row s *subsumes* row t as in [24]).

We assume that there does exist a final table that satisfies the values constraint—for example, we are not given a template that has multiple rows with the same key, or incorrect partial data.

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 final table in Section 2.2 satisfies the values constraint with this template.

Predicates constraint: Instead of specific values, we might want to specify template entries that are predicates, indicating that the collected values must satisfy the corresponding predicates. Note that predicates constraints subsume values constraints, since a value v in a template row is equivalent to the predicate "$=v$". The generalization of values constraints says that our goal is to obtain a final table S that satisfies the following *predicates constraint* with template T:

> For each template row $t \in T$, there exists a unique row $s \in S$ such that $\overline{s} \supseteq^* \overline{t}$, where $\overline{s} \supseteq^* \overline{t}$ states that each value in s satisfies the corresponding predicate in t, if one is present.

As with the values constraint, we assume that there does exist a final table that satisfies the predicates constraint.

In our running example, if we wish to further refine our template from the values constraint so the forward and Brazilian player must have $\geq$30 goals, and the Spanish player must have $\geq$100 caps, we would specify the following template:

name	nationality	position	caps	goals
		='FW'		$\geq$30
	='Brazil'			$\geq$30
	='Spain'		$\geq$100	

Note the final table in Section 2.2 satisfies the predicates constraint with this template.

2.4 Concurrent Operations

As briefly described in Section 1, CrowdFill shows an up-to-date version of the evolving table to each participating worker. The workers make changes to the table using the primitive operations defined in Section 2.2. In this section, we first explain the client-server structure of the system, and specify a "vote history" that is needed to maintain consistency. Then we explain how operations at one client are propagated to other versions of the table, and how they are applied when they arrive. Lastly, we show formally that our execution model handles concurrency elegantly: We prove a theorem stating that when the system quiesces, all copies of the table are identical.

Execution overview: The CrowdFill system consists of *clients* (the workers' web browsers) and a *server* to which all clients are connected. We assume that message deliveries between the server and clients are reliable and in-order. The 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 an 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 momentarily.

Vote history: To help maintain consistency across the server and all clients, we define data structures *UH* and *DH* (for "upvote history" and "downvote history"), mapping value-vectors to numbers of upvotes and downvotes, respectively. We use $UH[\overline{v}]$ and $DH[\overline{v}]$ to represent the numbers of upvotes and downvotes, respectively, that have been cast for a specific value-vector $\overline{v}$. Similarly to the candidate table, the server and each client maintain their own upvote and downvote histories (according to the specification below).

Applying locally-generated operations: Suppose the worker at client C performs a primitive operation *op*. Let R_C denote C's local copy of the candidate table. Also, let UH_C and DH_C denote C's upvote and downvote histories, respectively. For each operation type, client C applies *op* locally and sends a corresponding message m to the server as follows:

- insert(r): Insert a new empty row r into R_C, with $u_r{=}d_r{=}0$. Send message insert(r) to the server.
- fill(r,A,v): Delete row $r \in R_C$ from R_C. Construct a new row q whose value $\overline{q}$ is the same as $\overline{r}$ but with column A filled in with value v. Insert row q into R_C. If row q is now a complete row, set $u_q{=}UH_C[\overline{q}]$; otherwise, set $u_q{=}0$. Set $d_q{=}\sum_{\overline{w}\subseteq\overline{q}} DH_C[\overline{w}]$. Send message replace($r$,$q$,$\overline{q}$) to the server.
- upvote(r): Increment u_q for each row $q \in R_C$ whose value $\overline{q}$ is the same as $\overline{r}$. Increment $UH_C[\overline{r}]$. Send message upvote($\overline{r}$) to the server.
- downvote(r): Increment d_q for each row $q \in R_C$ such that $\overline{q} \supseteq \overline{r}$. Increment $DH_C[\overline{r}]$. Send message downvote($\overline{r}$) to the server.

We assume that insert and fill operations generate globally-unique row identifiers for their newly-constructed rows.

Processing received messages: Now suppose the server S receives a message m from client C. Let R_S denote the master copy of the candidate table. Also, let UH_S and DH_S denote the upvote and downvote histories at S. For each message type, the server processes message m as follows:

- insert(r): Insert an empty row r into R_S, with $u_r{=}d_r{=}0$.
- replace(r,q,$\overline{v}$): If row r is present in R_S, delete r from R_S. Construct row q whose value $\overline{q}$ is the same as $\overline{v}$. Insert row q into R_S. If row q is a complete row, set $u_q{=}UH_S[\overline{v}]$; otherwise, set $u_q{=}0$. Set $d_q{=}\sum_{\overline{w}\subseteq\overline{q}} DH_S[\overline{w}]$.
- upvote($\overline{v}$): Increment u_q for each row $q \in R_S$ whose value $\overline{q}$ is the same as $\overline{v}$. Increment $UH_S[\overline{v}]$.
- downvote($\overline{v}$): Increment d_q for each row $q \in R_S$ such that $\overline{q} \supseteq \overline{v}$. Increment $DH_S[\overline{v}]$.

In addition, the server forwards message m to all clients except C. When client $C'(\neq C)$ receives message m, C' processes m in exactly the same fashion as the server, as specified above.

2.4.1 How the Model Supports Concurrency

Our model was designed carefully so workers can operate independently, performing operations on copies of the same table with conflicts resolved in an intuitive and seamless fashion. It is easy to see (though formally proven in the next section) that insert and voting operations can be performed on independent copies and propagate to the other clients without conflict. Thus, the only source of potential conflict is 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.

Suppose client C performs a fill(r,A,v) operation. According to our formal model, in C's copy of the table, row r is replaced by a newly-constructed row q, instead of adding value v in place to row r. This replacement propagates via replace messages to the server and then all other clients. Generating a new row for each new column value, instead of filling the value into the existing row, turns out to be the key ingredient to enabling concurrency.

Suppose another client C' performs a fill(r,A',v') operation at the same time. If $A = A'$, i.e., they fill in the same column, eventually all clients have two copies of the original row for the two, possibly different, values. If one of the values is correct and the other isn't, further completion of the better row and voting should eventually yield the correct answer. If neither value is correct, neither row should be completed or upvoted. If both values are correct, one unnecessary row is created, but will not affect final correctness.

Now suppose $A \neq A'$, i.e., the clients fill in two different columns. If the two new values are incompatible, again eventually the correct row (if any) should emerge. If the two values are compatible, we rely on workers to eventually combine (by copying) the correct values into a single row. The alternative approach of always placing both new values in the same row would be advantageous when the values are compatible, but significantly disadvantageous when the values are incompatible, since the entire row would eventually be downvoted.

As an example when $A \neq A'$, suppose the candidate table currently contains the following row r:

name	nationality	position	caps	goals	↑	↓
		FW			0	0

If two operations, fill(r,name,Lionel Messi) and fill(r,nationality, Brazil), are performed by two different clients concurrently, we will ultimately get the following candidate table with rows q_1 and q_2:

name	nationality	position	caps	goals	↑	↓
Lionel Messi		FW			0	0
	Brazil	FW			0	0

Note had the two values been added in place to row r, the result would have been an incorrect row that neither client intended.

2.4.2 *Convergence*

In the specifications above, applying a locally-generated operation at client C is equivalent to processing its corresponding message m, as if C received m from the server. Thus, in the rest of this section, we use applying an operation and processing its corresponding message interchangeably.

When clients concurrently generate and process messages according to the specifications above, the server and all clients process each generated message once and only once. However, the server and each client may process the messages in a different order. Despite this difference, our convergence theorem guarantees 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).

Theorem: Suppose the server and all clients initially have an identical candidate table R_0 (both data rows and vote counts) as well as identical upvote and downvote histories, without any outstanding messages that need further processing. Suppose the clients together generate a set M of messages. If the system quiesces again after processing all messages in M, the server and all clients have an identical candidate table R_f, as well as identical upvote and downvote histories.

Proof: The proof proceeds in two parts. In part 1, we show that the set of rows in every copy of the candidate table converges, ignoring upvotes and downvotes. In part 2, we show that the upvote and downvote histories are identical across the server and all clients, and consequently the upvote and downvote counts for each row converge. We first introduce two lemmas and prove part 1; then we introduce another lemma and prove part 2.

Lemma 1: Associated with every row identifier r is a value $\overline{r}$. Every copy of the table containing row r at any time always has the same value $\overline{r}$ for r.

Proof of Lemma 1: Processing an insert message always creates an empty row. Based on the specification of processing replace messages, modifying value $\overline{r}$ for row r also assigns a new row identifier. Thus row identifier r cannot be associated with more than one value $\overline{r}$.

Lemma 2: For any row r, if there are two messages m_1 and m_2 in M such that m_1=insert(r) or replace(p,r,$\overline{r}$), and m_2=replace(r,q,$\overline{q}$), then message m_1 is processed before message m_2 at the server as well as at each client.

Proof of Lemma 2: If both m_1 and m_2 are generated at a single client C_i, m_1 obviously precedes m_2 at C_i. By in-order message delivery and processing, m_1 precedes m_2 at the server and at any other client C_j ($j \neq i$).

Otherwise, suppose m_1 and m_2 are generated at client C_i and C_j ($j \neq i$), respectively. Before C_j generates m_2, the server must have received and processed m_1 from C_i (and forwarded it to C_j), because m_2 refers to row identifier r. Thus m_1 precedes m_2 at C_i, C_j, and the server. By in-order message delivery and processing, m_1 precedes m_2 at any other client C_k ($k \neq i, j$).

Proof of convergence theorem (part 1): In this part we show that the server and all clients have the same set of rows in their candidate tables (ignoring upvotes and downvotes) after all messages in M have been processed. Notice that we only need to consider insert and replace messages, because upvote and downvote messages do not change the rows themselves.

Let R_A and R_B denote the sets of rows in candidate tables at two different locations (either two clients, or the server and one client), after processing all messages in M on the initial table R_0. By Lemma 1, to prove R_A=R_B, it suffices to show that there is no $r \in R_A$ such that $r \notin R_B$. Suppose, for sake of contradiction, that there exists $r \in R_A$ such that $r \notin R_B$.

We first show that row r cannot be in R_0. If r is in R_0, by $r \notin R_B$, M must include a message m such that m=replace(r,q,$\overline{q}$). Once m is processed on R_A, we have $r \notin R_A$, which contradicts our assumption $r \in R_A$.

From $r \notin R_0$ and $r \in R_A$, there exists a message $m_1 \in M$ such that m_1=insert(r) or replace(p,r,$\overline{r}$). R_B does not contain row r despite message m_1, so there must be another message $m_2 \in M$ (following m_1 at B) such that m_2=replace(r,q,$\overline{q}$). To satisfy $r \in R_A$, m_2 must precede m_1 at A; however, this processing order contradicts Lemma 2. Therefore, R_A and R_B are identical, which means the set of rows in the candidate tables converge across the server and all clients.

Lemma 3: The following invariants hold for each $r \in R$ at the server and each client:

$$u_r = UH[\overline{r}] \qquad d_r = \sum_{\overline{v} \subseteq \overline{r}} DH[\overline{v}]$$

where $UH[\overline{v}]$=0 if $\overline{v}$ is never upvoted, and $DH[\overline{v}]$=0 if $\overline{v}$ is never downvoted.

Proof of Lemma 3: Based on the specifications for processing insert and replace messages, when a new row r is created by processing either type of message, both u_r and d_r are initialized according to the invariants. Examining the specifications for processing upvote and downvote messages indicates that the invariants are maintained for all rows.

Proof of convergence theorem (part 2): Since the server and all clients process the same set of upvote and downvote messages

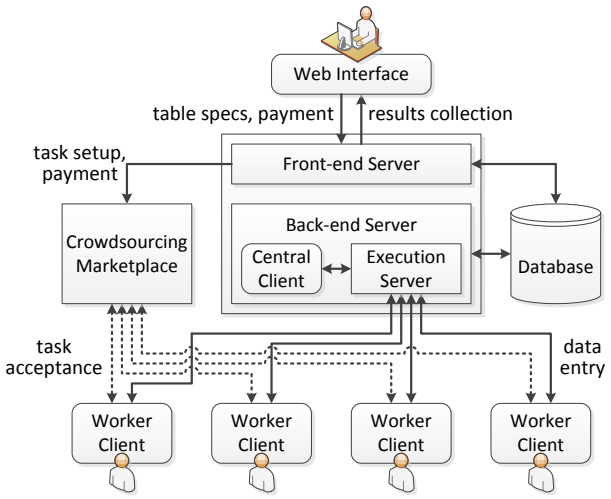

Figure 2: CrowdFill Architecture

in M, they all end up with the same UH and DH. Therefore, by Lemma 3, they all have the same upvote and downvote counts after all messages in M are processed. □

3. SYSTEM OVERVIEW

We have implemented a fully-functional prototype of the CrowdFill system. The system is based on the formal model of Section 2, although as expected a number of additional system-oriented details were needed. This section provides an overview of the CrowdFill system. Some of the most complicated aspects—constraint-satisfaction and worker compensation—are covered in detail in Sections 4 and 5.

3.1 Architecture

Figure 2 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. Note "user" refers to the entity (human or application) wishing to perform data collection, as distinct from a crowdsourced "worker".

1. Using the web interface, a user sends a table specification to the front-end server to launch data collection. Figure 3 shows CrowdFill's table schema editor.
2. The front-end server creates one or more tasks in the crowdsourcing marketplace. (The number of tasks is marketplace dependent. So far we have used Amazon Mechanical Turk [1] exclusively.)
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 (recall Figure 1), until the back-end server determines that enough data has been collected.
5. Using the web interface, the user retrieves collected data from the front-end server and pays workers through the crowdsourcing marketplace.

Next we describe the front-end server, back-end server, and worker client in more detail.

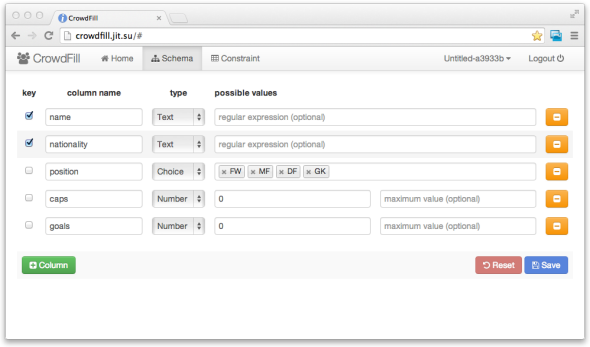

Figure 3: Table Schema Editor

3.2 Front-end Server

The CrowdFill server logically consists of a front-end server interacting with applications, and a back-end server interacting with workers. The front-end server provides applications with the CrowdFill REST API, which supports creating, updating, and deleting table specifications (including table schemas, scoring functions, and constraint templates), controlling the actual data collection, and retrieving collected data. Using the CrowdFill API, we built a web-based graphical user interface. The server stores all metadata and collected structured data in a MongoDB [4] database.

The front-end server communicates with one or more crowdsourcing marketplaces to attract workers (to the back-end server) and to pay those workers once enough data is collected. The current version of CrowdFill only supports Amazon Mechanical Turk, however additional marketplaces can easily be added, as long as the marketplace allows us to host questions "externally" and make "bonus" payments to workers, similar to Mechanical Turk [1].

3.3 Back-end Server

The back-end server corresponds to the server in CrowdFill's formal model described in Section 2.4: It maintains the master copy of the candidate table and broadcasts each incoming message to all clients except the one originating the message. We built the server using Node.js [7]; for connections between the back-end server and clients, we chose the Socket.IO library [8] so that most web browsers could participate as worker clients.

In addition to the basic functionality described as part of the formal model, the back-end server is responsible for populating a candidate table, determining the amount of monetary compensation for each worker, and storing a complete trace of worker actions for bookkeeping. We will discuss these aspects in Sections 4 and 5.

3.4 Worker Client

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 4.

Fill action: As shown in Figure 1 (Section 1), 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. (In this regard, this interface bears much similarity to online spreadsheets such as Google Docs [3] spreadsheet.) Filling in an empty cell generates a fill operation as described in Section 2.4. To encourage workers to fill in different parts of the table, each

client randomizes the order of rows in the local copy of the table presented to the worker.

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. (Implementing this behavior requires maintaining a log of worker identifiers and votes.) 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. Also, a single worker may not upvote more than one row with the same primary key. Finally, CrowdFill provides a feature allowing users to set a maximum number of votes per row, to prevent excessive voting.

4. SATISFYING THE CONSTRAINTS

Recall from Section 2.3 that our overall goal is to obtain a final table satisfying the *cardinality* and *values constraints*. Also recall that cardinality constraints are a special case of values constraints: we can include in the values constraint template T additional empty rows when the original template rows are fewer than the cardinality constraint. In the rest of this section, we assume cardinality constraints are absorbed by the values constraint in this fashion. As a reminder, a values constraint with template T says: for each template row $t \in T$, there exists a unique row s in the final table such that $\overline{s} \supseteq \overline{t}$.

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 2). 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.

4.1 Probable Rows Invariant

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. We first define 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. Based on the derivation of a final table from a candidate table as defined in Section 2.2, we say that row r is probable if it satisfies one of the following conditions:

1. Row r contains empty values for some primary key columns and has a zero score from its upvote and downvote counts. (Recall from Section 2.1 the score is computed by $f(u_r, d_r)$.)
2. Row r contains no empty values for the primary key columns and has a zero score, but no other row with the same primary key has a positive score.
3. Row r is a complete row with a positive score, and no other row with the same primary key has a greater score. If there are other rows with the same primary key and an equal score, only one row in the group is probable, and we assume ties are broken deterministically.

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 $\overline{r} \supseteq \overline{t}$.

Note there may be probable rows that do not correspond to any template rows. By the definition of probable rows and the values constraint, we have the following theorem.

Theorem: Suppose we have a candidate table R and a values constraint with a set T of template rows. Further suppose the PRI holds. Let $P' \subseteq R$ be the set of probable rows in the correspondence of the PRI. If every $p \in P'$ is a complete row, then the final table S derived from R satisfies the values constraint.

Proof: By the third condition in the definition of probable rows above, along with the final table derivation from Section 2.2, every complete probable row in R is in S. Since every $p \in P'$ is a complete row, we have $P' \subseteq S$. By PRI on R and T, each $t \in T$ corresponds to a unique $p \in P'$ such that $\overline{p} \supseteq \overline{t}$. From $P' \subseteq S$, each $t \in T$ corresponds to a unique $s \in S$ such that $\overline{s} \supseteq \overline{t}$. Therefore, the final table satisfies the values constraint. □

4.2 Maintaining the Invariant

We now explain how the CrowdFill system maintains the PRI. Initially client CC populates its candidate table R with the set T of template rows. In addition, client CC upvotes all complete template rows, as if those rows were completed by workers (recall Section 3.4). CC's initialization operations—as well as later operations—are propagated to the server and all other clients, as if CC were a worker client.

After initialization, all rows in R have nonnegative scores since no downvotes have been cast. In addition, recall from Section 2.3 that there does exist a final table satisfying the values constraint; this assumption implies that no two rows in T with complete values for the primary key columns have the same key. Thus, all rows in R are probable, and the PRI trivially holds.

To discuss subsequent states, we model relations between template rows and probable rows with a bipartite graph G. Let P denote the set of probable rows in R. The vertices are all template rows in T and all rows in P. There is an edge between $t \in T$ and $p \in P$ whenever $\overline{p} \supseteq \overline{t}$. (For implementation, graph G is simply stored as extra attributes in the candidate table.) In the context of graph G, the PRI is equivalent to this statement: A maximum bipartite matching for G, i.e., a largest set of edges where no two edges share a vertex, contains exactly $|T|$ edges.

As workers perform actions, graph G changes as the set P of probable rows changes. Client CC incrementally computes a maximum bipartite matching for G after each change in G. Whenever a change in G makes the size of a maximum bipartite matching for G smaller than $|T|$, client CC inserts probable rows into R so that the maximum bipartite matching contains exactly $|T|$ edges, as follows. When a template row $t \in T$ becomes "free" (i.e., does not appear in the matching), client CC performs a breadth-first search from t to a free probable row in P to find an augmenting path (i.e., a path between two free nodes where edges are alternately in and out of the matching). In the worst case, this BFS takes $O(|P||T|)$ time; however, if no probable rows can be connected to two or more template rows with distinct values (e.g., the template has only empty rows, or all primary keys are completely filled in), it takes $O(|P|)$ time.

By Berge's theorem [9], if an augmenting path is found, we can increase the size of the bipartite matching back to $|T|$. Otherwise, maintaining the PRI requires inserting another row q into R. Usually we can use $\overline{t}$ for the value of the new row q, in which case an edge between t and q is added to the matching. However, inserting row q with value $\overline{t}$ does not always make q probable. Let

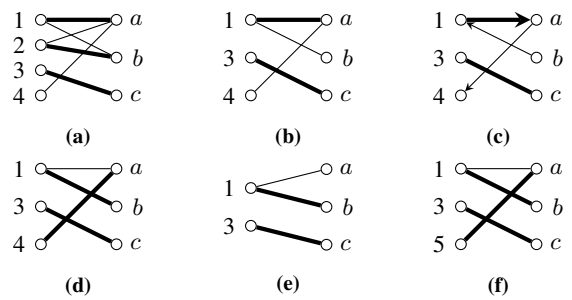

Figure 4: Bipartite Graph Representation of the PRI Maintenance

us see what can go wrong. First, value $\bar{t}$ might have been downvoted by several workers, giving potential row q a negative score. This case usually indicates the template row t has incorrect values. Second, value $\bar{t}$ may have all primary key columns filled in, while there is already a probable row with the same primary key and a higher score. In either case, client CC first attempts to shuffle the matching so that another template row $t' \in T$ becomes free. If CC cannot find another probable template row t', to maintain the PRI CC has no option but to remove t from T, perhaps violating the user's original intention. When this case occurs, our current system continues data collection with the reduced template; it might instead be appropriate to raise an error and abort data collection, depending on the user's preference.

4.3 Example

Suppose we have a values constraint with the template T from Section 2.3, which we repeat here labeling the template rows a, b, and c:

#	name	nationality	position	caps	goals
a			FW		
b		Brazil			
c		Spain			

Further suppose the candidate table R currently contains the following four rows, labeled 1, 2, 3, and 4:

#	name	nationality	position	caps	goals	↑	↓
1	Neymar	Brazil	FW			0	0
2	Ronaldinho	Brazil	FW			0	1
3		Spain				0	0
4	Messi		FW			0	0

Recall from Section 2.1 that the scoring function $f(u_r, d_r)$ for our example is $u_r - d_r$ if $u_r + d_r \geq 2$, and 0 otherwise. Thus all four rows above are probable, i.e., $P = \{1, 2, 3, 4\}$. Figure 4a shows the bipartite graph G corresponding to P and T. The maximum bipartite matching incrementally maintained by client CC is denoted by the thick edges in the graph—three edges in Figure 4a.

Now suppose row 2 is downvoted by one more worker, so its score becomes -2. Figure 4b shows graph G after row 2 is removed from P. Since the template row b is now free, client CC starts a breadth-first search from row b and finds an augmenting path b–1–a–4 (Figure 4c). Based on this path, CC updates the matching to include 4–a, 1–b, and 3–c (Figure 4d). In this case, the PRI is maintained without inserting an additional row into R.

Now suppose a worker fills in Messi's caps as 82, replacing row 4 with row 4' shown below. Further suppose row 4' is downvoted by two workers, so it is removed from P (Figure 4e). Although the template row a is free, there is no augmenting path starting from row a. Thus CC inserts the value of row a into R as row 5, adding 5–a to the maximum bipartite matching (Figure 4f). The resulting candidate table is now:

#	name	nationality	position	caps	goals	↑	↓
1	Neymar	Brazil	FW			0	0
2	Ronaldinho	Brazil	FW			0	2
3		Spain				0	0
4'	Messi		FW	82		0	2
5			FW			0	0

5. COMPENSATING WORKERS

So far we have described the machinery for obtaining a final table satisfying the constraints, assuming there are workers willing to perform the necessary actions. In this section, we present CrowdFill's compensation scheme, to motivate workers to perform "useful" actions. We first discuss several challenges in designing an effective compensation scheme, and our overall approach to tackling them. Then, we describe how CrowdFill determines monetary compensation for each worker, based on the final table. Lastly, we describe how CrowdFill provides workers with estimated incremental compensation, to keep them engaged during data collection.

5.1 Challenges and Approach

In paid crowdsourcing, workers are motivated by monetary compensation, and typically have a desire to maximize their total earnings. On the user side, our challenge is an example of the *cost-latency-quality* tradeoffs discussed in [25]: We want to exploit workers' desire to earn money in order to obtain a final table of high quality, without too much cost or latency. We describe how CrowdFill's compensation scheme addresses this challenge.

Roughly, our overall scheme is based on compensating workers for those data entries that actually contribute to the final table, directly or indirectly.Under this scheme (combined with row-wise voting), a worker entering correct value has a much better chance of getting paid than one entering incorrect data. Since populating a candidate table as described in Section 4 minimizes wasted work, this scheme does not penalize "good" workers. Moreover, this scheme makes the overall 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 current 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. In addition to final payment, it is necessary during data collection to provide estimated monetary value for each action. We currently use a relatively simple approach, with more complex algorithms slated for future work. Experimentally, our simple approach produces estimates that are reasonably close to the actual compensation for each worker; see Section 6.

We first detail how final compensation is calculated after the table is complete in Section 5.2. Compensation estimation during data collection is covered in Section 5.3.

5.2 Allocating Total Budget to Workers

Suppose the user specified a total monetary budget B, and CrowdFill has obtained a final table S satisfying the constraints. The back-end server stores a complete trace of worker actions in terms of a set M of messages received from all worker clients (as in Section 2.4), where each message in M is uniquely timestamped and annotated with the worker identifier originating the message. Note messages from special client CC (Section 4) are not included in M. Our goal is to determine overall compensation for each worker who participated in data collection, given B and M.

Our overall strategy proceeds as follows. Let C denote the set of cells in S whose values have been entered by workers. (Recall some cell values in S are from template rows and entered by CC.)

1. For each cell $c \in C$, we find exactly one replace message in M that directly contributed to c, and at most one replace message in M that indirectly contributed to c. We will formalize these concepts below.
2. We compute U, the set of upvote messages in M that contributed to a row in S.
3. We compute D, the set of downvote messages in M that (indirectly) contributed to the final table S.
4. We distribute the total budget B across all cells in C, all upvote messages in U, and all downvote messages in D.
5. For each cell $c \in C$, we allocate the portion of B assigned to c to the one or two replace messages contributing to c.
6. We calculate overall compensation for each worker by summing compensations from Steps 4 and 5 for their messages.

We elaborate Steps 1–3 in Section 5.2.1, Step 4 in Section 5.2.2, and Step 5 in Section 5.2.3. Step 6 is straightforward.

5.2.1 Defining the Notion of Contribution

We specify which messages in M contributed to the final table S. Recall from Sections 2.4 and 3.4 that worker clients send three types of messages to the back-end server:

- replace(r,q,$\overline{q}$), generated by a fill(r,A,v) operation
- upvote($\overline{r}$), generated by an upvote(r) operation
- downvote($\overline{r}$), generated by a downvote(r) operation

We now discuss four classes of contribution: direct and indirect for replace, plus upvote and downvote.

Direct contribution of replace: Consider a cell $c \in C$ that corresponds to column A of row $s \in S$, hereafter called $s.A$. The replace(r,q,$\overline{q}$) message in M that directly contributes to cell c is the one that filled in the A value in the row that eventually became row s. Formally, replace(r,q,$\overline{q}$), generated by a fill(r,A,v) operation, directly contributes to a cell $c \in C$ for $s.A$ if there exists a series of $k \geq 1$ messages, replace(r_0,r_1,$\overline{r_1}$), replace(r_1,r_2,$\overline{r_2}$),..., replace(r_{k-1},r_k,$\overline{r_k}$) in M, such that $r_0 = r$, $r_1 = q$, and $r_k = s$.

Given a cell $c \in C$, there is exactly one message in M directly contributing to c: Having no directly contributing replace message contradicts $c \in C$, while having two or more contributing replace messages contradicts the fact that fill operations generate globally-unique row identifiers.

Indirect contribution of replace: Consider the case where a worker enters a "correct" value v for a column A, creating a partial row q with value $\overline{q}$. Suppose row q does not evolve through additional fill operations to be part of the final table, yet value $\overline{q}$ is a subset of a final row. If the worker was the first to enter value v for column A, we should give some compensation for that indirect contribution. More formally, replace(r,q,$\overline{q}$), generated by a fill(r,A,v) operation, indirectly contributes to a cell $c \in C$ for $s.A$ if $\overline{q} \subseteq \overline{s}$ holds, and there is no replace(p,o,$\overline{o}$) in M that is generated by a fill(p,A,v) operation and has a timestamp older than replace(r,q,$\overline{q}$).

Given a cell $c \in C$ for $s.A$, whose value is v, there is no message indirectly contributing to c if v is from a template row (in which case client CC was the first to provide value v), or the first replace(r,q,$\overline{q}$) to enter value v to column A does not satisfy $\overline{q} \subseteq \overline{s}$. Otherwise, there is exactly one message in M indirectly contributing to c. Note a single replace message may contribute to c both directly and indirectly: if it was the first to enter value v, and the row eventually became row s.

Contribution of upvote: An upvote($\overline{r}$) message in M directly contributes to a given row $s \in S$ if it increased the upvote count of row s, i.e., $\overline{r} = \overline{s}$. One exception is the case when the upvote message was automatically sent by a fill action that completed a row by providing the last value (Section 3.4), which we do not count as a separate contribution. Note each upvote message can contribute to at most one row in S, since S does not have duplicate rows due to the primary key constraint.

Contribution of downvote: A downvote($\overline{r}$) message contributes to the final table S if it is consistent with all rows in S, i.e., if there is no row $s \in S$ such that $\overline{s} \supseteq \overline{r}$.

5.2.2 Allocation Schemes

We now present three schemes with different levels of sophistication that distribute the total budget B across all cells in C and all messages in $U \cup D$. Recall U and D are the sets of upvote and downvote messages in M, respectively, that contributed to the final table S.

Uniform allocation: As a simple baseline, we allocate the budget B uniformly across all cells in C and all messages in $U \cup D$. In this case, compensation for each cell in C and each message in $U \cup D$ is $b = \frac{B}{|C|+|U|+|D|}$.

Column-weighted allocation: We now take into account the possibility that some columns are inherently more difficult to fill in than others, and that upvoting and downvoting may not have equal difficulty nor difficulty similar to filling in values. We assign *weights* to each column, and to upvoting and downvoting. We will explain how to choose these weights shortly.

The column-weighted allocation scheme allocates the budget B to all cells in C and all messages in $U \cup D$ proportionally to their corresponding weights. Suppose column A_i ($1 \leq i \leq m$) has weight y_i, and upvote and downvote have weights $y_\uparrow$ and $y_\downarrow$, respectively. Let C_i denote a set of cells in C for column A_i, and let Y denote the sum of weights across all cells in C and all messages in $U \cup D$, i.e., $Y = \sum_{j=1}^{m} y_j|C_j| + y_\uparrow|U| + y_\downarrow|D|$. Compensation for each $c \in C$ for column A_i is $y_i B/Y$, and compensation for each upvote and downvote are $y_\uparrow B/Y$ and $y_\downarrow B/Y$, respectively. Note our uniform allocation scheme is a special case of column-weighted allocation where all weights are equal.

There are many possible ways to determine the weights, including asking the user to designate weights as part of the table specification. In our current system, we automatically choose the weights using the trace of worker actions stored in M. Our goal is to set y_i ($1 \leq i \leq m$), $y_\uparrow$, and $y_\downarrow$ to the median times taken for workers to generate replace messages for filling in A_i, upvote messages, and downvote messages, respectively, that contribute to the final table. (Using medians rather than averages makes the weights more resilient to outliers.) For now, we use the difference of timestamps in two consecutive messages from the same worker as the time taken for generating the second message, but we are aware of its flaws. As future work we can incorporate a more sophisticated mechanism in the client side to address this limitation.

Dual-weighted allocation: Column-weighted allocation assumes that filling in a particular column in different rows is equally difficult. However, for the case of primary key columns in particular, entering new values tends to get progressively more difficult as the table fills up. In this case, the column-weighted allocation scheme overcompensates primary key values completed earlier and undercompensates primary key values completed later.

The dual-weighted allocation scheme addresses this problem by assigning progressively higher weights to the cells for the primary key columns. Specifically, for each primary key column A_i, we

assign linearly increasing weights from $(1 - z_i)y_i$ to $(1 + z_i)y_i$, to all cells in C_i in the order that their values first appeared in column A_i of the candidate table: Compensation for the cell containing the k-th value in key column A_i is $b_k = (1 + \frac{2z_i}{|C_i|-1}(k - \frac{|C_i|+1}{2}))y_i B/Y$. In the current system, compensation for the other cells, upvotes, and downvotes remains the same as in column-weighted allocation.

Again there are many possible ways to determine parameter z_i. To minimize user intervention, our goal is to estimate a reasonable z_i value from M. To do so, we fit b_k to the times taken to complete k-th value ($k = 1, \ldots, |C_i|$), using linear least squares regression. Since we require each z_i to be nonnegative and less than 1, we override a negative z_i with 0, and z_i greater than 1 with 1.

5.2.3 Splitting Cell Compensation

Finally, we allocate the portion of B assigned to each cell $c \in C$ to the one or two replace messages contributing to c. Given a cell $c \in C$, let b_c denote the portion of B assigned to c. Let h_c be a "splitting factor" between 0 and 1, discussed momentarily. We allocate $h_c b_c$ to the message directly contributing to c, and $(1-h_c)b_c$ to the message indirectly contributing to c, if any. (Note as discussed in Section 5.2.1 some cells may not have any indirectly contributing messages, so this scheme does not necessarily exhaust B.)

To determine default values for h_c, for now we use an ad-hoc scheme based primarily on intuition. For cell c in a primary key column, our intuition says that the indirect contribution is most important, because it increases the number of distinct keys. Thus we set $h_c = 0.25$ by default, giving the directly contributing message $0.25b_c$ and the indirectly contributing message (if any) $0.75b_c$. For cell c in a non-key column, the indirect contribution is not as valuable as in the case of a primary key column since we are not looking for unique values; however, it is still (at least) as valuable as the direct contribution. In this case we set $h_c = 0.5$ by default, giving each of the directly and indirectly contributing messages $0.5b_c$. We allow the user to override this scheme by setting h_c for each column, and we plan to investigate more complex schemes as future work.

5.3 Estimating Compensation

As described in Section 5.2, CrowdFill pays workers who participate in data collection based on their contribution to the final table S. Now consider the worker perspective. To keep workers engaged during data collection, it is necessary to provide estimated monetary compensation for each action. This estimation problem is quite challenging on its own, and a complete exploration is beyond the scope of this paper. Here we describe our intuitive initial approach, which we evaluate empirically in Section 6. Note our approach treats all workers as equally likely to perform useful actions; if we kept track of worker's past performance we could adjust our estimates accordingly.

In our current system, estimated compensation is provided based on the following two assumptions. First, we calculate the estimated compensation for each action assuming that the action will eventually contribute to S. Second, the estimated compensation for a fill action assumes both direct and indirect contribution. Thus, filling in a value for a column when the same value is already present elsewhere in the column may result in smaller compensation than estimated.

Now we describe how CrowdFill estimates compensation for each action, for the three allocation schemes from Section 5.2.2.

Uniform allocation: Recall that compensation for each cell in C and each message in $U \cup D$ is $b = \frac{B}{|C|+|U|+|D|}$. Thus we need to estimate $|C|$, $|U|$, and $|D|$. For $|C|$, we use the number of empty values in the template T, which is accurate in most cases. For $|U|$, we start with $(u_{min} - 1) \times |T|$, where u_{min} is the smallest number such that $f(u_{min}, 0) > 0$, then increase the estimate as probable rows in R get more upvotes. Lastly, for $|D|$, we use the number of downvotes that are consistent with all currently probable rows.

Column-weighted allocation: In column-weighted allocation, we need to estimate weights $y_1, \ldots, y_m$, $y_\uparrow$, and $y_\downarrow$, in addition to $|C|$, $|U|$, and $|D|$ as described above. We begin with the same simple estimates as uniform allocation, which may be far off. Then, whenever new latency information is accumulated based on worker actions, we update $y_1, \ldots, y_m$, $y_\uparrow$, and $y_\downarrow$ to the median times taken to generate replace messages for filling in A_i, upvote messages, and downvote messages, respectively, that contribute to the current set of probable rows. Thus, estimates get more accurate over time, eventually converging to the actual weights used for calculating final compensation.

Dual-weighted allocation: We need to estimate z_i for each primary key column A_i, in addition to all parameters needed in column-weighted allocation. Again we initially assume uniform column weights, and we further assume that all z_i's are zero. Whenever a primary key column A_i is filled in, we first fit z_i to the times taken to complete the k-th value in the current probable rows, using linear regression. Then we update y_i, taking z_i into account since latencies to be observed would be higher than latencies already observed. For all other columns and votes, the column weights along with $|C|$, $|U|$, and $|D|$ are calculated as in column-weighted allocation.

6. EXPERIMENTAL EVALUATION

In this section we present our preliminary experimental evaluation of the CrowdFill system to provide initial validation of our approach. First, we briefly describe and assess the overall effectiveness of CrowdFill's table-filling approach for a data-collection task. Then we focus on our compensation schemes, comparing the three allocation schemes from Section 5.2, as well as measuring the accuracy of estimated compensation from Section 5.3.

Experimental setup: For our experiments we used Amazon Mechanical Turk's *developer sandbox*, a non-production environment for testing crowdsourcing applications [2]. Human workers were recruited locally and worked on our data collection tasks exclusively. All workers were volunteers: although compensation amounts were calculated, the workers were not actually compensated. Although this setup doesn't exactly reflect crowdsourcing scenarios in practice, it enabled us to control the number of concurrent workers and run many rounds of experiments without delay or escalating monetary costs.

We deployed the CrowdFill front-end and back-end servers on Nodejitsu [6], a Node.js [7] hosting platform in the cloud; CrowdFill ran on a single "drone" (an individual unit of computing power). CrowdFill's database was hosted on MongoLab [5], a database-as-a-service provider for MongoDB [4].

Schema and constraints: We used the running example SoccerPlayer schema from Section 2, with an additional date-of-birth (dob) column:

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

Recall from Section 2.1 that our scoring function implements a "majority of three or more" voting scheme.

Our goal was to collect information about 20 soccer players with caps between 80 and 99 inclusive, starting from an empty table. (Players with caps ≥ 100 were excluded because their information is readily available from the FIFA Century Club.)

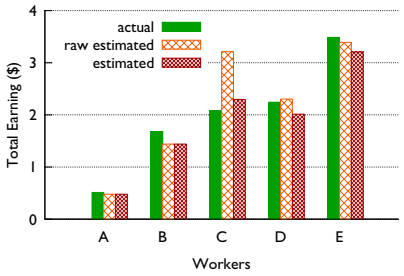

Figure 5: Accuracy of estimated compensation

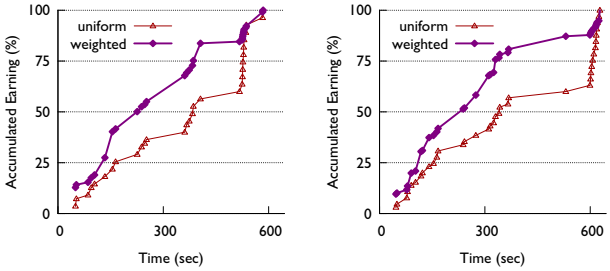

Figure 6: Earning rates for uniform and weighted allocation, two workers

Overall effectiveness: We report results for a representative run that collected data using five human workers. Note that results may vary significantly based on the workers participating in an experiment; drawing meaningful conclusions would require a large number of trials using different sets of workers. In our representative run it took 10 minutes 44 seconds to obtain a final SoccerPlayer table with 20 complete rows. At completion, the candidate table contained 23 rows; two rows were downvoted twice or more, and one extra row was added by a conflict. In this run, all 20 final rows were accurate, although we occasionally observed inaccurate rows in other runs.

Worker compensation: Continuing with our representative run, we set our total monetary budget to $10, and we used our most sophisticated allocation scheme: dual-weighted allocation. Under this scheme, the five workers had a wide range of compensation: $0.51, $1.68, $2.08, $2.24, and $3.49. The worker who earned $3.49 performed 54 actions (fill, upvote, and downvote combined), while the worker who earned $0.51 performed only 9 actions. The significant variation in compensation demonstrates that our approach does reward those workers who contribute more to the final table.

Although we used dual-weighted allocation, in our runs CrowdFill did not observe that it took progressively longer to obtain new primary keys, one of the bases for the allocation scheme. The lack of "slowdown" is probably because we were collecting data for only 20 players, while we estimate there are more than 200 players whose caps value is in the desired range (making it easy to come up with 20). Thus, the compensation amounts would have been exactly the same using column-weighted allocation.

Accuracy of estimated compensation: Now we evaluate whether workers earned what they expected, based on the estimates CrowdFill provided during data collection. Figure 5 shows actual and estimated compensation in our representative run for each of the five workers. The middle bar for each worker represents the sum of estimates shown when actions were performed. These raw estimates were reasonably close to the actual compensation across all five workers, with a mean absolute percentage error of 16.1%.

As discussed in Section 5.3, the estimated compensation for each action is calculated assuming the action will eventually contribute to the final table. If a worker consistently provides incorrect values, the estimation error can be arbitrarily large. The rightmost bar for each worker shows the sum of estimates only for those actions that contributed to the final table. Using these corrected estimates, we observed a mean absolute percentage error of 9.9%.

In general, we observed that accuracy of estimated compensation largely depended on the allocation scheme. With uniform, column-weighted, and dual-weighted allocation schemes, we observed mean absolute percentage errors of about 3%, 16%, and 25%, respectively, across many experiments using different schemas and workloads. It's not surprising that the more complex schemes are more difficult to estimate; improving our estimates for the more complex schemes is an important area for future work.

Comparing allocation schemes: If we had used uniform allocation (and ignoring the fact that workers may have behaved differently under a different scheme), the five workers would have earned $0.59, $2.01, $1.54, $2.38, and $3.48. Notice that some, but not all, values are quite different from the previous calculation. For the third worker, the difference is more than 25%. It turns out that the third worker never carried out upvote or downvote actions; since in this run voting was considered easier than filling in most columns, the worker was penalized by the uniform allocation scheme.

Another comparison we can make across compensation schemes is "stability," i.e., whether workers earn at a steady rate throughout the table-filling task. Figure 6 shows earning rates of two representative workers during our representative run. The x-axis represents elapsed time from the start of data collection, while the y-axis represents accumulated earning as percentage of the eventual total. Thus, the slope of a line in the graph corresponds to the earning rate. We plot the actual earning rates using dual-weighted allocation (which, recall, is equivalent to column-weighted allocation in this run), along with what the earning rates would have been under uniform allocation. We can see in this experiment that weighted allocation appears to be somewhat more stable than uniform allocation in terms of earning rate. Clearly, more extensive experiments would be needed to fully understand earning rates in a wide variety of settings.

7. RELATED WORK

Some recent systems have proposed to incorporate crowdsourcing into relational database systems, providing a convenient means of collecting structured data from the crowd. In particular, CrowdDB [11] and Deco [16] are most relevant to CrowdFill because they can extend a relation both "vertically" (by adding new tuples) and "horizontally" (by filling in missing attributes). As briefly discussed in Section 1, both systems use a microtask-based approach that poses specific questions to workers, which is fundamentally different from CrowdFill's table-filling approach. Also, users in these systems express their data collection goals as SQL queries, while CrowdFill users rely on constraint templates. At a high level, CrowdFill's maintenance of the *Probable Rows Invariant* described in Section 4 bears some similarity to Deco's optimal degree of parallelism [17]: they both attempt to minimize wasted work while creating enough tasks for workers.

There has been a large body of prior work on various aspects of monetary compensation in crowdsourcing environments. Many papers, e.g., [14, 18, 19], have studied specific compensation schemes to elicit desired behaviors from workers. Reference [19] compared several different compensation schemes in terms of accuracy of worker answers; CrowdFill's compensation scheme is most analogous to "reward-agreement", which rewards workers for answers agreeing with the majority. Another important problem in crowdsourcing marketplaces is task pricing. While reference [12] de-

scribes a method for estimating a worker's reservation wage (i.e., the lowest wage a worker is willing to accept), more recent studies [20, 21] propose an alternative approach where workers indicate their desired wage by submitting a bid. Pursuing these directions may allow CrowdFill to improve its allocation scheme, with an aim of minimizing total monetary cost without a prespecified budget.

Real-time cooperative editing systems [3, 10, 22] allow multiple clients to concurrently edit the same evolving document. Like CrowdFill, these systems have a notion of convergence, and of intention preservation; they incorporate a technique called "operational transformation" to maintain those properties. CrowdFill's specific setting of structured data, and potentially inconsistent data entries, led us to the formal model in Section 2, which handles potentially-conflicting operations intuitively and seamlessly.

8. CONCLUSIONS AND FUTURE WORK

We presented CrowdFill, a system for collecting structured data from the crowd. To obtain high-quality data quickly, CrowdFill uses a novel table-filling approach based on a model that enables real-time collaboration among workers, and that resolves conflicts in an intuitive fashion. CrowdFill allows the specification of constraints on the collected data, and guides data collection towards the constraints while providing an intuitive data-entry interface. CrowdFill's compensation scheme distributes a specified monetary budget to workers in a way that rewards those worker operations that contribute to the final result, while encouraging useful work along the way.

As immediate future work, the system itself would benefit from several extensions:

- The predicates constraint described in Section 2.3 would add significant power.
- In addition to the three worker actions, we could combine the primitive operations to make data entry more convenient for workers. For example, we could provide a worker-level "modify" action that overwrites a non-empty cell, translating underneath to a series downvote, insert, and fill operations.
- As a practical matter, we should allow workers to "undo" actions, starting with upvotes and downvotes.
- We might have the system recommend certain cells to individual workers, guiding workers to fill in different parts of the table. Our current approach randomizes the presentation of rows to each worker, but a more sophisticated strategy would take into account workers' skills and the current state of the table, making the whole data collection process more efficient.

Other than system improvements, we see four major directions.

- Although we have shown empirically that our current compensation scheme works reasonably well, much of it is ad-hoc based on intuition; further study is needed to fully understand and refine the current scheme.
- Another extremely important area of investigation is the potential effect of spammers in our system, i.e., workers trying to obtain payment for insincere work. Our compensation scheme discourages incorrect answers, but the transparent nature of our table-filling approach may enable spammers to hinder data collection, both individually and collectively, and to steal credit by copying potentially correct answers from other workers. A full understanding of the potential for such actions in CrowdFill, and how to inhibit or eliminate them through modifications to the compensation scheme and/or data-entry interface, is a significant challenge for the future.
- Our experimental evaluation described in Section 6 is preliminary; more comprehensive evaluation should provide valuable insights into the CrowdFill system and the table-filling approach in general. As a first step, larger-scale evaluations are in order, including larger table sizes, more concurrent workers, and a variety of data domains. Unfortunately, as with much work in crowdsourcing, the process of conducting large-scale empirical evaluations in a realistic setting can be expensive and time-consuming.
- Finally, we developed CrowdFill as an alternative approach to previously-proposed microtask-based approaches for collecting structured data from the crowd [11, 16, 23]. An important future direction is a thorough comparison of the two approaches, perhaps largely through empirical studies, considering all three axes of cost, latency, and quality.

9. ACKNOWLEDGMENTS

We are grateful to Hector Garcia-Molina and Jaeho Shin for many useful discussions.

10. REFERENCES

[1] Amazon Mechanical Turk. *http://mturk.com/*.
[2] Amazon Mechanical Turk Developer Sandbox. *https://requestersandbox.mturk.com/*.
[3] Google Docs. *http://docs.google.com/*.
[4] MongoDB. *http://www.mongodb.org/*.
[5] Mongolab. *http://www.mongolab.com/*.
[6] Nodejitsu. *http://www.nodejitsu.com/*.
[7] Node.js. *http://www.nodejs.org/*.
[8] Socket.IO. *http://socket.io/*.
[9] C. Berge. Two theorems in graph theory. *PNAS*, 43(9):842–844, 1957.
[10] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.
[11] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, 2011.
[12] J. Horton and L. Chilton. The labor economics of paid crowdsourcing. In *EC*, 2010.
[13] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP*, 2009.
[14] W. Mason and D. J. Watts. Financial incentives and the "performance of crowds". In *HCOMP*, 2009.
[15] A. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR*, 2011.
[16] 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.
[17] H. Park, A. Parameswaran, and J. Widom. Query processing over crowdsourced data, http://ilpubs.stanford.edu:8090/1052/. Technical report, Stanford InfoLab, 2012.
[18] O. Scekic, H. L. Truong, and S. Dustdar. Incentives and rewarding in social computing. *Communications of the ACM*, 56(6):72–82, 2013.
[19] A. D. Shaw, J. J. Horton, and D. L. Chen. Designing incentives for inexpert human raters. In *CSCW*, 2011.
[20] Y. Singer and M. Mittal. Pricing mechanisms for crowdsourcing markets. In *WWW*, 2013.
[21] A. Singla and A. Krause. Truthful incentives in crowdsourcing tasks using regret minimization mechanisms. In *WWW*, 2013.
[22] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.
[23] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.
[24] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.
[25] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, 2012.