Crowdstore: A Crowdsourcing Graph Database

Vitaliy Liptchinsky[1,2(✉)], Benjamin Satzger[2], Stefan Schulte[1], and Schahram Dustdar[1]

[1] Distributed Systems Group, TU Wien, Vienna, Austria
{liptchinsky,schulte,dustdar}@dsg.tuwien.ac.at
[2] Microsoft, Redmond, USA
benjamin.satzger@microsoft.com

Abstract. Existing crowdsourcing database systems fail to support complex, collaborative or responsive crowd work. These systems implement human computation as independent tasks published online, and subsequently chosen by individual workers. Such pull model does not support worker collaboration and its expertise matching relies on workers' subjective self-assessment. An extension to graph query languages combined with an enhanced database system components can express and facilitate social collaboration, sophisticated expert discovery and low-latency crowd work. In this paper we present such an extension, CRowdPQ, backed up by the database management system Crowdstore.

Keywords: Database theory · Graph query languages · Crowdsourcing

1 Introduction

Crowd-powered hybrid databases have gained momentum in recent years [8,16,17] due to their ability to combine human and machine computation. These database engines allow the specification of human-computable predicates that transform into Human Intelligence Tasks (HITs), which are posted online and are expected to be picked up by workers. In spite of being cumbersome for workers, as browsing HITs is time-consuming [10], such pull model has limitations for collaborative and expert work. The better-suited push model requires the crowd platform to support sophisticated worker discovery capabilities in order to assign or recommend tasks to workers.

Plethora of tasks require synchronous collaboration [9,10]. Successful collaboration can be largely influenced by social relations between human workers. Moreover, reusing teams that exhibited successful collaboration in the past can greatly increase chances of success for new assignments.

Realtime crowdsourcing is based on the concept of flash crowds [10]: groups of individuals who respond moments after a request and can work synchronously. The benefits of realtime crowdsourcing have been shown in [2,3]: paying workers a small wage to stay on call is enough to draw a crowd together within seconds.

In this paper we show how a graph query language can be extended to express synchronous collaboration, social formations (teams) of crowd workers, sophisticated worker discovery, as well as complex crowdsourcing patterns, such as

S. Guo et al. (Eds.): CollaborateCom 2015, LNICST 163, pp. 72–81, 2016.
DOI: 10.1007/978-3-319-28910-6_7

iterative computation, control groups, ranking, etc. Also, we show how classical database engine components, such as indexes, caches, and the buffer pool manager, can be extended to improve worker discovery, team formation, and flash crowds management.

2 Motivation

Consider following examples of crowdsourcing tasks:

1. Implement a web page, and create images for it. These two tasks require distinct skill sets. Also, to improve collaboration, workers assigned to the tasks should be socially related, or at least in close social vicinity.
2. You have a recorded melody fragment and want to know what song it belongs to. You want only those workers to work on it who like rock music.
3. You need a set of hand-drawn paintings. You ask crowd workers to draw and rate the paintings. To avoid biased ratings, workers rating the paintings should not be socially related to workers drawing the paintings.

Existing crowdsourcing query languages fall short of expressing complex relations between crowd workers working on related tasks, e.g., in the first example. Moreover, discovery of proper workers might be a crowdsourcing task itself, e.g., in the second example. Finally, even trivial crowdsourcing patterns, as in Example 3, are cumbersome to express in existing crowdsourcing query language adaptations.

In the next section we review existing query languages employed for crowdsourcing database scenarios.

3 Related Work

Table 1 provides an overview of existing hybrid human-machine databases. We analyze their ability to express complex crowdsourcing workflow patterns (such as control groups and ranking), social formations between workers, and sophisticated worker discovery. Also, we overview techniques they employ for query optimizations, i.e., to minimize the number of generated HITs under cooperative/collaborative scenarios, and approaches they utilize to enable realtime crowdsourcing.

Qurk [14,15] and hQuery [16] were among the first attempts on expressing crowdsourcing tasks as declarative queries. The SQL-based query language in Qurk [14] exploits user-defined scalar and table functions (called *TASK*) to retrieve, join, sort and filter data from the crowd. Qurk also extends SQL with a *POSSIBLY* clause to reduce the number of join candidates. Join optimizations in Qurk consider batching of items, thus minimizing the number of generated HITs. hQuery [16], a Datalog-like declarative model, features human-based and algorithm-based predicates. Authors focus on presence of uncertainty in

Table 1. Supported features in selected crowdsourcing databases

	Workflow	Query optimizations	Worker discovery	Social formations	Realtime crowdsourcing
Qurk [15]	+/–	+/–	–	–	–
hQuery [16]	+/–	+/–	–	–	–
CrowdDB [8]	+/–	–	–	–	–
CrowdSPARQL [1]	+/–	–	–	–	–
Deco [17]	+/–	–	–	–	–
CrowdSearcher [4]	+/–	–	–	+/–	–
Join optimizations [13,18,19]	N/A	+/–	–	–	–

the result set as well as optimization challenges, such as trade-offs between the number of certain answers, time allocated and monetary cost.

CrowdDB [8] introduces extensions to the SQL data definition language to define *CROWD*-enabled columns and tables, i.e., which should be fetched from an underlying crowdsourcing platform. Also, it introduces CROWDEQUAL and CROWDORDER extensions to the SQL data modification language.

CrowdSPARQL [1] introduces a hybrid query engine that allows executing SPARQL queries as a combination of machine- and human-driven functionality. Similar to *CROWD*-enabled columns and tables in CrowdDB, CrowdSPARQL defines crowdsourced *classes* and *properties* in VoID (Vocabulary of Interlinked Datasets). Also, CrowdSPARQL defines an *ORDER BY CROWD* operator.

The Deco [17] database semantics are defined based on the so-called *Fetch-Resolve-Join* sequence, i.e., data is fetched using *Fetch* rules, then data inconsistencies are resolved using *Resolution* rules and afterwards conceptual relations are produced by outer-joining the resolved tables.

CrowdSearcher [4] allows putting constraints on crowd workers via a mapping model, e.g., friends of a specific user, geo-localized people, workers on a selected work platform. However, it is not possible to specify either relations between workers, or social formations.

Neither of the query languages above allow specifying *CROWD*-enabled constraints on workers, nor relations between crowd workers themselves. Hence, these query languages cannot support examples provided in the previous section. Moreover, they lack capabilities to express complex workflows in a natural way.

Multiple papers discuss the problem of minimizing the number of HITs required to resolve JOIN operations. We have grouped those papers in the table under the "Join Optimizations" row. CrowdER [18] suggests using a hybrid human-machine approximation approach to filter out non-matching join pairs (with similarity ratio below certain threshold), aiming to minimize the number of HITs required to join entities. Wang et al. [19] discuss join optimization based on transitive relations. Contrary, in [13] authors discuss selectivity estimation performed by the crowd, which implies optimal join ordering. All these

approaches focus on crowd-based and automatic join resolution, neglecting a query writer, who can have better insight into selectivity of the data queried and optimal join ordering.

While simple caching of results produced by HITs has been discussed (e.g., [15]), it has not been discussed how to cache successful workers and social formations (teams). Also, to the best of our knowledge, no papers have suggested application of classical database techniques and algorithms for realtime crowdsourcing.

4 Query Language

Efficient specification of social collaboration largely depends on the ability to specify complex social formations of crowd workers. Social formations can be intuitively represented as graph patterns [12], which makes graph query languages a natural choice for describing social collaboration. In this section we show how Conjunctive Regular Path Queries, a formalism behind many graph query languages [20], can be extended to overcome their shortcoming for incorporating free-text conditions and relations between data to be fetched and workers who fetch the data.

4.1 Preliminaries

A database is defined as a directed graph $K = (V, E)$ labeled over the finite alphabet Σ. If there is a path between node a and node b labeled with $p_1, p_2, ..., p_n$ we write $a \xrightarrow{p_1 p_2 \ldots p_n} b$. In the remainder of this section we give definitions of (conjunctive) regular path queries, similar to other works, like [6].

Definition 1 (Regular Path Queries)**.** A regular path query (RPQ) $Q^R \leftarrow R$ is defined by a regular expression R over Σ. The answer $ans(Q^R, K)$ is the set connected by a path that conforms to the regular language $L(R)$ defined by R:

$$ans(Q^R, K) = \{(a, b) \in V \times V \mid a \xrightarrow{p} b \text{ for } p \in L(R)\}.$$

Conjunctive regular path queries allow to create queries consisting of a conjunction of RPQs, augmented with variables.

Definition 2 (Conjunctive Regular Path Queries)**.** A conjunctive regular path query (CRPQ) has the form

$$Q^C(x_1, ..., x_n) \leftarrow y_1 R_1 y_2 \wedge ... \wedge y_{2m-1} R_m y_{2m},$$

where $x_1, \ldots, x_n, y_1, \ldots, y_m$ are node variables. The variables x_i are a subset of y_i (i.e., $\{x_1, \ldots, x_n\} \subseteq \{y_1, \ldots, y_m\}$), and they are called distinguished variables. The answer $ans(Q^C, K)$ for a CRPQ is the set of tuples $(v_1, ..., v_n)$ of nodes in K such that there is a total mapping σ to nodes, with $\sigma(x_i) = v_i$ for every distinguished variable, and $(\sigma(y_i), \sigma(y_{i+1})) \in ans(Q^R, K)$ for every RPQ Q^R defined by the term $y_i R_i y_{i+1}$.

4.2 CRowdPQ

CRowdPQ is derived from CRPQ by extending the notion of RPQ with DRPQ and RRPQ defined as follows.

Definition 3 (DRPQ). A descriptor regular path query (DRPQ) $Q^{DR} \leftarrow DR$ is a regular path query defined over the extended alphabet $\Sigma \bigcup \Gamma$, where Γ is a human-interpretable infinite alphabet of labels. Essentially, the *Descriptor* relations DR are free-text conditions that can be answered by human workers. Kleene star in descriptor regular path queries corresponds to iterative human computation.

Definition 4 (RRPQs). A resolver regular path query (RRPQ) $Q^{RR} \leftarrow RR$ is a regular path query over a predefined alphabet $P = produce \cup consume$, where the labels *produce* and *consume* correspond to dataflow producers and consumers respectively. The left operand of a *Resolver* relation RR always has to be a worker node supplied by an integrated crowdsourcing platform. Essentially, the *Resolver* relations are dataflow constructs between the data to be fetched and the workers working on the data. Note, Resolvers are not the only relations that can be specified between a worker and the task at hand, i.e., RPQs can be used to specify worker constraints. Kleene star and concatenation over the *produce* relation represent higher-order selection of workers, e.g., workers find workers who find workers who can fetch data.

4.3 Expressiveness

In this section we demonstrate the expressiveness of CRowdPQ by implementing the three use cases from the motivating scenario. For this purpose we employ a CRowdPQ-enhanced version of SPARQL 1.1: Descriptor (DRPQ) and Resolver (RRPQ) relations are denoted using triangle and square brackets respectively.

Synchronous Collaboration. Implement a web page, and create images for it. These two tasks require distinct skill sets. Also, to improve collaboration, workers assigned to the tasks should be socially related, or at least in close social vicinity (i.e., there exists a path between them of maximum length of 2).

```
SELECT ?webPage, ?pictures
WHERE
{
    ?webPage <‘‘Design a web page’’>.
    ?pictures <‘‘Draw pictures for the web page’’> ?webPage.
    ?webDesigner [produce] ?webPage.
    ?artist [produce] ?pictures.
    ?webDesigner friendOf[1, 2] ?artist.
    ?webDesigner [consume] ?pictures.
    ?artist [consume] ?webPage.
}
```

Worker Discovery. You have a recorded melody fragment and want to know what song it belongs to. You want only those workers to work on it who like rock music.

```
SELECT ?melodyName
WHERE
{
    ?melodyName <‘‘Is similar to’’> @file.
    ?melodyName <‘‘Can you recognize the melody?’’>.
    ?musicFan [produce] ?melodyName.
    ?musicFan <‘‘Find a person passionate about rock music.’’>.
    ?indexWorker [produce] ?musicFan.
}
```

Note, that the specification of workers with no constraints is optional, i.e., $?indexWorker$ can be omitted.

Workflow and Social Relations. You need a set of hand-drawn pictures. You ask crowd workers to draw and rate the pictures. To avoid biased ratings, workers rating the pictures should not be socially related to workers drawing the pictures.

```
SELECT ?picture, ?ranking
WHERE
{
    ?picture <‘‘Draw a funny sheep.’’>.
    ?talentedPainter [produce] ?picture.
    ?mercilessCritic [consume] ?picture.
    ?mercilessCritic [produce] ?ranking.
    ?ranking <‘‘How funny is this sheep?’’> ?picture.
    FILTER NOT EXISTS { ?talentedPainter friendOf ?mercilessCritic }
}
```

Note, the example above can be easily changed to a control group (i.e., one worker creates a picture and another one filters it) by replacing the `?rank` variable with the `?filteredPicture` variable and adjusting descriptor relations appropriately.

5 Database Engine

In this section we show how classical database components can be extended to be able to cope with human workers as schemaless, volatile and context-dependent data sources.

5.1 Synchronous Collaboration: Social Formations and Caching

In Examples 1 and 3 of Sect. 4.3 we have shown the expressivity of our query language with respect to specifying social formations.

In traditional RDBMS, the purpose of query caching is to speed up query evaluation by reusing results from previous queries. While classical caching mechanisms of preserving query results are also applicable in Crowdstore, here we consider a different kind of caching. Instead of caching results, we cache workers and social formations of workers (teams) in case of synchronous collaboration. If a worker has been answering recently a similar query to the query at hand, she might be a good fit to the task. For example, if a worker has been searching

recently through newspapers for information about charity events, she might be able to quickly answer a query of searching recent newspapers for road incidents.

The key element for efficiency of such caches is the ability to identify similarities between queries, which resorts to finding a similar subgraph in a list of cached query graphs (subgraph isomorphism). Matching descriptors can be achieved by finding similarity between texts (or extracting labels). Answering subgraph isomorphism is a NP-complete problem, so when using exact matching (isomorphism) the query cache will not scale. Moreover, for complex queries expressing dense social formations, like cliques, it might be difficult to find an exact match. To alleviate these two problems we can use approximate graph matching, which, however, might not return the most suitable workers. Depending on the importance of the relations between workers, we can choose either of the two heuristics: relax the input query graph by removing worker nodes or data nodes in order to focus on worker experience (i.e., who worked successfully on what) or maximize social similarity respectively (i.e., what teams were successful).

Such quality caches can be pre-built by running pre-labeled queries over gold standard data (e.g., [5,7,11]) and caching workers and teams that have shown good quality.

5.2 Crowd Indexes

In Example 2 of Sect. 4.3 we have shown how the discovery of crowd workers can be crowdsourced itself. We call index workers those workers that select and search for workers for a query at hand. The distinction to regular workers should be driven by different reward mechanisms applied to index workers, i.e., index workers should be rewarded depending on the work quality of the workers they choose. The Crowdstore design incorporates two techniques for worker indexes:

- Routing indexes. In the most trivial case the system can ask an index worker to simply enter a list of workers she thinks satisfy the descriptor relation(s), or a list of index workers who can route further. Routing indexes represent directed graphs, and Crowdstore needs to detect cycles.
- Zonemap indexes. If there are other relations in addition to descriptors that are adjacent to a worker node in a query graph, then Crowdstore can efficiently filter worker candidates. In such case, index workers can be presented with a list of workers they can select from. However, such lists might be immerse, so the system needs to group workers by available tags (e.g., by country, or age), presenting several hierarchical lists to index workers. This approach enables index workers to quickly filter the list of workers.

5.3 Query Optimization

One of the central aspects of query optimization is join ordering. Consider the following example: “Where was this picture taken? - this query should be answered by workers living in London”. An extremely inefficient case of evaluating such

query is sending the question to all the workers, and then filtering responses by workers living in London. When joining descriptor and regular relations the join order is predefined, as automated filtering is always more efficient than filtering done by a crowd. However, join ordering for two crowd-produced relations can be highly error-prone and inefficient as the cost of relations is not known beforehand. Contrarily to existing crowd-powered approaches, in Crowdstore we take a different approach by assuming that a query writer can have better insight into predicate selectivity than a crowd. CRowdPQ, as shown in Example 3 in Sect. 4.3, provides a query writer with the ability to specify join ordering by using *consume* relations. If no *consume* relations are specified, then existing joining techniques can be applied.

Another approach CRowdPQ provides is "denormalized" ("collaborative") joins: instead of asking crowd workers to work independently on two separate relations, Crowdstore can ask workers to collaborate and produce already matched and joined results. The benefit of "collaborative" join is that the worker produced data can be ambiguous and, without direct contact with the data producer, difficult to match. Moreover, creative tasks require collaboration, as shown in Example 1 in Sect. 4.3. If a single worker node in a query is connected with a *produce* relation to multiple nodes, then "denormalization" will result in sending a single HIT to a crowd worker asking to provide data for the whole query graph. When working on "collaborative" joins, crowd workers will need to use synchronous collaboration software.

Joining two crowd-produced relations without predefined join ordering allows two approaches. The first approach consists of two sets of workers producing data for relations independently and in parallel, and then a third set of workers joins the two produced relations. The second approach is inherent to relational DBMS, i.e., data is produced for one relation and then is used to filter in-place data for another relation.

5.4 Crowd Pool

In [2,3] the authors show that paying workers a small wage to stay on call is enough to draw a crowd together two to three seconds later. The problem here is which workers to keep on payroll based on variable query patterns, e.g., what subset of workers satisfy most queries given the budget constraints. If a worker becomes less active, it is better to replace the worker with another one. Basically, an efficient system needs to maintain a limited set of useful/active workers and efficiently replace ineffective workers with new ones. This scenario resembles problems addressed by the buffer pool manager in traditional RDBMs, i.e., limited working set, replacement of least recently used database pages. Henceforth, we draw here correspondence between crowd workers and database pages: similarly as how a crowd worker can generate/provide data, a database page can provide table records. The central part of the buffer pool manager in RDBMs is the clock algorithm, which evicts least-recently-used pages (LRU). The Crowd pool component in Crowdstore similarly evicts least-recently-active (LRA) work-

ers. The Crowdstore adaptation of the clock algorithm, however, incorporates the following adjustments:

- Tracking slow-performing workers. The purpose of keeping a page in-memory in RDBMs is the ability to fetch results faster. Similarly, keeping a worker on payroll leads to the expectation of fast results. If a worker responds slower than other payroll (and non-payroll) workers, then Crowd pool can evict such worker.
- Delayed enrollment on payroll. In RDBMs when reading records from a database page it is necessary to fetch the page in memory (page-in). In Crowdstore, however, there is no such restriction, i.e., even if some query required workers with a skill set disjoint with skill sets in Crowd pool, such skill set might not be needed again. So, apart of counting how useful is a payroll worker, Crowd pool needs to count how useful a non-payroll worker is.

6 Future Work, Discussion and Conclusion

In this paper we present the hybrid human-machine database Crowdstore, powered by the graph query extension CRowdPQ. Contrarily to existing crowdsourcing query languages, CRowdPQ can express social collaborations between crowd workers, sophisticated worker discovery and complex crowdsourcing workflow patterns. Incorporation of dataflow constructs makes CRPQs slightly less declarative, since a query writer can directly influence execution plans. However, mispredictions in query evaluation performed by the crowd possess considerable cost overhead, rendering explicit join ordering critical. Crowdstore serves as a holistic design concept of a new generation crowdsourcing database, featuring extended indexes, caches and buffer pool manager. A more detailed description and evaluation of each of these components will be provided in our future work.

References

1. Acosta, M., Simperl, E., Flöck, F., Norton, B.: A sparql engine for crowdsourcing query processing using microtasks. Institute AIFB, KIT, Karlsruhe (2012)
2. Bernstein, M.S., Brandt, J., Miller, R.C., Karger, D.R.: Crowds in two seconds: Enabling realtime crowd-powered interfaces. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST 2011, pp. 33–42, New York, NY, USA. ACM (2011)
3. Bernstein, M.S., Karger, D.R., Miller, R.C., Brandt, J.: Analytic methods for optimizing realtime crowdsourcing. Computing Research Repository (2012)
4. Bozzon, A., Brambilla, M., Ceri, S.: Answering search queries with crowdsearcher. In: Proceedings of the 21st International Conference on World Wide Web, WWW 2012, pp. 1009–1018, New York, NY, USA. ACM (2012)
5. C. Callison-Burch. Fast, cheap, and creative: Evaluating translation quality using amazon's mechanical turk. In Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1, EMNLP 2009, pp. 286–295, Stroudsburg, PA, USA. Association for Computational Linguistics (2009)

6. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Containment of conjunctive regular path queries with inverse. In: Proceedings of the 2000 International Conference on Knowledge Representation and Reasoning, KR 2000, pp. 176–185. Breckenridge, Colorado, USA (2000)
7. Downs, J.S., Holbrook, M.B., Sheng, S., Cranor, L.F., Are your participants gaming the system?: Screening mechanical turk workers. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2010, pp. 2399–2402, New York, NY, USA. ACM (2010)
8. Franklin, M.J., Kossmann, D., Kraska, T., Ramesh, S., Xin, R., Crowddb: answering queries with crowdsourcing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 61–72, New York, NY, USA, 2011. ACM (2011)
9. Ishii, H., Kobayashi, M., Clearboard: a seamless medium for shared drawing and conversation with eye contact. In: Proceedings of the SIGCHI Conference on Human factors in computing systems, pp. 525–532. ACM (1992)
10. Kittur, A., Nickerson, J.V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., Horton, J.: The future of crowd work. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, CSCW 2013, pp. 1301–1318, New York, NY, USA. ACM (2013)
11. Le, J., Edmonds, A., Hester, V., Biewald, L., Ensuring quality in crowdsourced search relevance evaluation: the effects of training question distribution. In: SIGIR Workshop on Crowdsourcing for Search Evaluation, pp. 21–26 (2010)
12. Liptchinsky, V., Satzger, B., Zabolotnyi, R., Dustdar, S.: Expressive languages for selecting groups from graph-structured data. In: Proceedings of the 22Nd International Conference on World Wide Web, WWW 2013, pp. 761–770, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee (2013)
13. Marcus, A., Karger, D., Madden, S., Miller, R., Oh, S.: Counting with the crowd. In: Proceedings of the 39th International Conference on Very Large Data Bases, PVLDB 2013, pp. 109–120. VLDB Endowment (2013)
14. Marcus, A., Wu, E., Karger, D., Madden, S., Miller, R.: Human-powered sorts and joins. Proc. VLDB Endow. **5**(1), 13–24 (2011)
15. Marcus, A., Wu, E., Karger, D.R., Madden, S., Miller, R.C., Crowdsourced databases: query processing with people. In: 5th Biennial Conference on Innovative Data Systems Research (2011)
16. Parameswaran, A., Polyzotis, N.: Answering queries using humans, algorithms and databases. In: Conference on Inovative Data Systems Research (CIDR 2011). Stanford InfoLab, January 2011
17. Parameswaran, A.G., Park, H., Garcia-Molina, H., Polyzotis, N., Widom, J., Deco: declarative crowdsourcing. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM 2012, pp. 1203–1212, New York, NY, USA. ACM.(2012)
18. Wang, J., Kraska, T., Franklin, M.J., Feng, J.: Crowder: crowdsourcing entity resolution. Proc. VLDB Endow. **5**(11), 1483–1494 (2012)
19. Wang, J., Li, G., Kraska, T., Franklin, M. J., Feng, J.: Leveraging transitive relations for crowdsourced joins. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 229–240, New York, NY, USA. ACM (2013)
20. Wood, P.T.: Query languages for graph databases. ACM SIGMOD Record **41**(1), 50–60 (2012)