# CrowdDB

## Answering Queries with Crowdsourcing

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Master's Thesis Nr. 2

Systems Group, Department of Computer Science, ETH Zurich

CrowdDB – Answering Queries with Crowdsourcing

by

Sukriti Ramesh

Supervised by

Prof. Dr. Donald Kossmann

September 2010–March 2011

inf | Informatik
Computer Science

# Abstract

Despite the advances in the areas of databases and information retrieval, there still remain certain types of queries that are difficult to answer using machines alone. Such queries require human interaction to either provide data that is not readily available to machines or to gain more information from existing electronic data.

CrowdDB is a database system that enables *difficult* queries to be answered by using crowdsourcing to integrate human knowledge with electronically available data. To a large extent, the concepts and capabilities of traditional database systems are leveraged in CrowdDB. Despite the commonalities, since CrowdDB deals with procuring and utilizing human input, several existing capabilities of traditional database systems require modifications and extensions. Much unlike electronically available data, human input provided by crowdsourcing is unbounded and virtually infinite. Accordingly, CrowdDB is a system based on an open-world assumption. An extension of SQL, termed as CrowdSQL, is used to model data and manipulate it. CrowdSQL is also used as the language to express complex queries on the integrated data sources. Furthermore, interaction with the crowd in CrowdDB requires an additional component that governs automatic user interface generation, based on available schemas and queries. Also, performance acquires a new meaning in the context of a system such as CrowdDB. Response time (efficiency), quality (effectiveness) and cost (in $) in CrowdDB are dependent on a number of different parameters including the availability of the crowd, financial rewards for tasks and state of the crowdsourcing platform. In this thesis, we propose the design, architecture and functioning of CrowdDB. In addition, we present the details of building such a system on an existing Java-based database, H2. The design and functionalities of CrowdDB have also been presented in [13].

# Acknowledgement

I would like to express my sincere thanks to Prof. Dr. Donald Kossmann for his constant support and guidance during the course of this thesis. I would also like to thank Prof. Michael Franklin, Dr. Tim Kraska and Reynold Xin from the University of California, Berkeley for the valuable discussions and useful advice through the period of study.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

CrowdDB is a novel database system that leverages the capabilities of the crowd to answer queries that are otherwise difficult for traditional database systems. This chapter provides an introduction to CrowdDB. Section 1.1 provides the background and motivation to build a system such as CrowdDB. Section 1.2 provides the detailed problem statement. A summary of the contribution of this thesis is provided in Section 1.3 along with an overview of the remaining chapters in Section 1.4.

## 1.1 Background and Motivation

**Role of machines** Machines are the prime component of information systems. Their ability to store, index and retrieve large amounts of data helps automate a large number of processes in the system. As machines have grown more powerful, technologies have also evolved to harness the capabilities of the newer, more powerful machines. Newer technologies, in turn, aid the development of richer applications that perform several advanced operations efficiently and effectively. Despite such development, there are some tasks in information systems that still require human intervention.

**Role of humans** Humans play a pivotal role in the functioning of information systems since there exist certain tasks that only they are capable of doing. Typically characterized by their subjective nature, examples of such tasks include defining mapping rules for data integration, tagging images, ranking entities, etc. Even if state-of-the-art technologies allow machines to deal with such tasks, their usage comes at prohibitively high costs. Additionally, humans are capable of providing information that is not readily available on the web. Consequently, they are indispensable for tasks that involve collecting, matching, ranking and aggregating such information.

**Difficult Queries** Traditional databases are built on the closed-world assumption, i.e., the data that they contain is presumed to be complete and correct. Given such an assumption, there are three classes of queries that are difficult to answer with traditional relational database technology.

The first class of difficult queries includes queries on incomplete or missing information. Querying for an entity that does not exist in the database will return an empty result set. For example, consider the following query.

```
SELECT * FROM movie WHERE title = 'The Godfather';
```

The query will return an empty result set if the `movie` table contains no entries pertaining to `'The Godfather'`, despite the fact that the entity exists in the real-world.

While it is possible that the database contains no entry corresponding to `'The Godfather'`, it may contain an entry corresponding to `'Godfather'`. Resolving `'The Godfather'` and `'Godfather'` to be the same real-world entity is representative of the second class of difficult queries that relational databases are incapable of handling.

The third class of difficult queries involves subjective comparisons and ranking of multiple entities. Since such comparisons are typically based on opinions, it is impossible for machines to provide the correct answer. A query aiming to find the *best* picture of the Golden Gate Bridge in a given collection of images is an example of such a query.

## 1.2 Problem Statement

Given the multitude of application scenarios and their complexity, neither machines nor humans alone can answer all possible queries. State-of-the art-techniques in databases and information retrieval still fall short while answering complex queries.

Machines and people have disjoint capabilities. Machines are capable of efficiently storing and retrieving data of various kinds using associated metadata. Humans, on the other hand, are capable of making subjective decisions. In order to be able to respond to complex queries, it is necessary to tap the combined knowledge of both machines and humans. To this end, CrowdDB primarily deals with integrating the inherent capabilities of humans into a database system, hence enabling it to answer otherwise difficult queries. Human input is obtained by using *crowdsourcing*. CrowdDB models crowd input as relations in a database. Once federated in CrowdDB, the data from electronic data sources and from people complement each other to make it possible to answer traditionally difficult queries.

To a large extent, CrowdDB taps the underlying design principles of traditional databases. Given the nature of CrowdDB though, there are certain glaring differences that need to be catered to. Traditional databases are based on a closed-world assumption. CrowdDB, on the other hand, is partially based on crowd input. Crowd input being virtually infinite invalidates the closed-world assumption. Owing to its open-world nature, CrowdDB requires the implementation of several extensions to traditional databases. Furthermore, since CrowdDB is essentially a database system it is possible to use a declarative programming language such as SQL to access the federated information. For this purpose though, traditional SQL needs to be extended to cater to interactions with the crowd. Operators need to be provided that permit interleaving of query processing and crowdsourcing. Additionally, CrowdDB needs to ensure performance of the system is not adversely impacted while interacting with humans. Metrics such as latency, cost and accuracy adopt new meanings in the context of such a database system and hence, need to be redefined.

## 1.3 Contribution

CrowdDB is a database system that uses crowdsourcing to answer queries that are difficult for traditional databases. This thesis provides a detailed description of the architecture of CrowdDB and the extended functionality of each of its components. Figure 1.1 gives an overview of CrowdDB. The specification of CrowdSQL, an extension of SQL that enables the integration of the crowd into a database system, is also presented. Special semantics are introduced that allow CrowdDB to tackle the open-world issue. The compile-time and run-time stages of query processing in CrowdDB are explained in great detail. New query operators are introduced that allow crowdsourcing to be included in query plans. Furthermore, since CrowdDB directly interacts with the crowd to answer queries, automatic user interface (UI) generation plays a prime role in such a system. This thesis includes various aspects of generating UIs automatically based on submitted queries and the schema of queried relations. Finally, a prototype of CrowdDB implemented by extending H2, an open-source Java database, is also presented. Preliminary experiments performed using Amazon Mechanical Turk, a crowdsourcing platform, demonstrate that CrowdDB is indeed able to answer queries that are difficult for traditional databases. Several aspects related to the design and functionality of CrowdDB are also presented in [13].

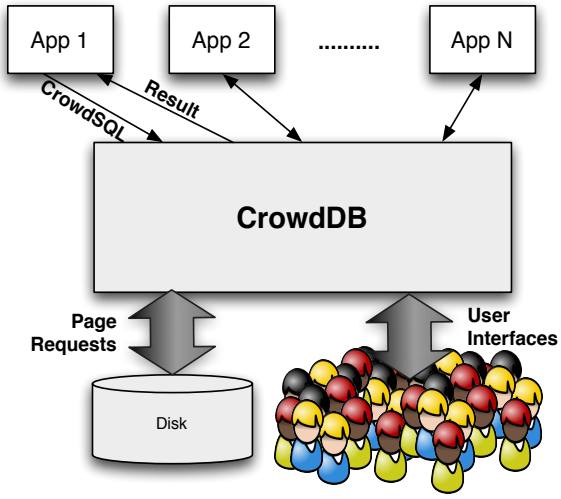


Figure 1.1: CrowdDB

## 1.4 Overview

The rest of the thesis is organized as follows: Chapter 2 gives an overview of crowdsourcing. Since CrowdDB uses Amazon Mechanical Turk as the underlying crowdsourcing platform, a description of its workflow and API is also provided. Chapter 3 explains the system architecture and components of CrowdDB. It also includes an overview of the features of H2, the open-source Java database used for the implementation of the CrowdDB prototype. Chapter 4 provides an overview of the data model used in CrowdDB. Chapter 5 presents the extensions introduced to SQL DDL, DML and the query language to enable automatic integration of crowdsourced data. Chapter 6 elaborates on the compile-time functioning of CrowdDB while Chapter 7 provides the details of the run-time system, including the processing model of CrowdDB. Chapter 8 presents the preliminary results of experiments performed on Amazon Mechanical Turk. Chapter 9 provides a summary of prior work related to the database and crowdsourcing communities. Finally, the conclusion and possible avenues for future work are covered in Chapter 10.

# Chapter 2

# Crowdsourcing

CrowdDB uses crowdsourcing to obtain human input. An overview of crowdsourcing is provided in Section 2.1. Section 2.2 explains the working of Amazon Mechanical Turk, the crowdsourcing platform employed in CrowdDB.

## 2.1 Overview

Crowdsourcing is comprised of publishing tasks that can be completed by a set of people, aptly referred to as the *crowd* to achieve a set of objectives known to the publisher. Crowdsourcing is a relatively new idea that provides opportunities to tap the collective intelligence of people across the globe, using Web 2.0 enabled technologies. Characterized by the on-demand and elastic workforce, crowdsourcing provides a scalable and cost-effective solution usable in diverse application scenarios.

Crowdsourcing tasks broadly fall into two categories. The first category of crowdsourcing allows the crowd to contribute collectively towards a single task. The task could range from developing a new technology to designing a system. The ESP game [38] is one such example. The second category of crowdsourcing allows the crowd to focus on a range of simple yet distinct tasks, specific to application scenarios or businesses. The tasks usually help in systematically collecting or analyzing data. Such crowdsourcing typically involves using an Internet marketplace, such as Amazon Mechanical Turk, where publishers can post tasks and the crowd can respond by solving the tasks and submitting answers.

## 2.2 Amazon Mechanical Turk (AMT)

Crowdsourcing platforms, such as Amazon Mechanical Turk (AMT) [1], commercialize the trend of using human input for the completion of various tasks, primarily those that computers are unable to do. AMT is one of the suites of Amazon Web Services. It is a crowdsourcing Internet marketplace that enables usage of human input to complete simple tasks, called microtasks. Owing to their nature, such microtasks are typically difficult or impossible for computers to solve. For humans, solving microtasks requires minimal training if any, and little time and effort investment. AMT is built on the principle of 'Artificial Artificial Intelligence' or AAI. The idea of AAI is to identify tasks for which AI is not yet adequate and outsource them to humans instead. CrowdDB is built using AMT and its API, the details of which are explained in the following sections.

### 2.2.1 AMT Terminology

AMT allows businesses to post a wide variety of tasks to be completed by a global, diverse and scalable worker community. Workers, on the other hand, have the opportunity to choose from a large number of tasks of varying nature. Terms used specifically in the context of AMT are explained in this section.

**Requester** A requester is typically the representative of a business and publishes tasks that are crowdsourced on AMT. The requester decides the nature of the task, the qualification (if any) that a worker must have to complete the task and the remuneration received for successfully completing it. After the crowdsourced results are received, accepting or rejecting the results is at the discretion of the requester.

**Worker** A worker is an individual that selects and performs one or more tasks on AMT. Upon successful submission and approval of results, the worker receives a cash reward, the value of which is defined by the requester.

**HIT** A HIT or a Human Intelligence Task is the smallest unit of work posted on AMT by a requester. Examples of HITs include tagging pictures, answering survey questions, finding new information or completing information about existing entities.

**Assignment** Every HIT can be replicated on AMT. Each replication is termed as an assignment. AMT ensures that a single worker can work on a maximum of one assignment in each HIT. Such plurality, enabling multiple workers to submit answers to the same HIT, is useful in measuring the quality of crowdsourced results. Rewards are given to each of the workers for successfully completed and approved assignments.

**HIT Type** Before posting a HIT to AMT, a requester is required to assign the HIT to a certain HIT Type. The HIT properties that define a HIT Type are title, description, keywords, reward, assignment duration, auto-approval time period and a set of zero or more qualifications.

**HIT Group** HITs of the same type are grouped together into a HIT Group. Such a grouping makes it easier for workers to identify the HITs that they would prefer to complete. Groups with higher number of HITs typically receive higher visibility on AMT.

**Qualification** Certain HITs are only available to workers who complete a qualification test, possibly time-bound. Requesters use qualifications to make sure that their HITs are performed by workers who have a specified minimum skill set. Qualifications are useful when HIT completion requires specific skills or knowledge, such as for language translation.

**Reward** The money earned by a worker for successfully completing an assignment is termed as reward. Once the requester approves submitted results, AMT automatically transfers the reward from the requester's prepaid HIT balance to the Amazon Payments account of the worker. The minimum reward for an assignment is $0.01.

### 2.2.2 AMT Workflow

AMT allows requesters to define HITs, the number of assignments for each HIT and the corresponding reward. Requesters may also qualify their workforce and pay exclusively for what they use. AMT collects a commission of 10% of the reward from the requester. The minimum payable commission is $0.005 per HIT.

AMT functions as follows: Requesters can post HITs to AMT either by using the Requester User Interface (RUI) and HIT templates or by using the AMT API. Section 2.2.3 presents a summary of relevant operations that can be performed using the AMT API. Requesters also specify the number of assignments per HIT, the reward (in $) per assignment and the qualifications, if any. AMT groups HITs by their type. Workers can search for HITs and accept assignments that they find suitable. Once they finish processing the assignments and submit the results, the requester can access the crowdsourced

results using RUI or the AMT APIs. Based on the submitted results, the requester may approve or reject the assignment. Approvals are typically given for complete and useful answers. Upon approval of an assignment, the worker gets paid the designated reward. Additionally, submitted assignments are automatically approved if no action is taken within the auto-approval time-out period specified while creating the HIT.

AMT provides an interface for workers to easily view, search and retrieve HITs that match specific search criteria. Upon accepting an assignment of a HIT, the worker views an interface provided by the requester of the particular HIT. The requester has complete control over the design of the HIT interface. Hence, it is in the best interest of the requester to provide a descriptive interface that is conducive to better productivity and meaningful results.

Over a period of time, crowdsourcing platforms such as AMT inadvertently build relationships between the requesters and the worker community. Workers may tend to favor requesters who provide certain type of tasks, provide good interfaces and have good payment practices. To steadily build a reputation in the worker community may serve as a long-term advantage to requesters. Additionally, based on results submitted and approved, workers can build or maintain their track records, usable as a proof of reliability.

### 2.2.3 AMT API

AMT allows invoking the web service using its SOAP interface or its REST interface. The SOAP interface can be used to invoke specific operations by referring to the corresponding WSDL file which describes the operations along with formats and data types of the requests and responses. If the REST interface is used, the response is in the form of an XML document that conforms to a specific schema. Based on the operation being invoked, it may be necessary to set certain parameters. The following are the operations from the AMT API used by CrowdDB.

#### CreateHIT

- **Description**
  The `CreateHIT` operation enables the creation of a HIT based on a set of request parameters. The operation can be invoked in two ways – either by using the HIT Type, as explained in Section 2.2.1, or by explicitly mentioning HIT property values.

- **Request parameters**
  `HITTypeId or (title, description, keywords and reward)` - To describe the HIT Type and accordingly, the HIT Group of the newly created HIT
  `Question` - An XML data structure to describe task content and UIs visible to the worker
  `Lifetime` - The duration of time in seconds after which the HIT is expired and becomes unavailable to workers
  `MaxAssignments` - The number of times that the HIT can be completed by distinct workers

- **Response**
  Once a HIT is successfully created, a response in the form of a `HIT` element is received. Among other attributes, the `HIT` element contains the `HITId` attribute which is used to uniquely identify the HIT for future processing.

#### GetAssignmentsForHIT

- **Description**
  The `GetAssignmentsForHIT` operation returns all completed assignments for a HIT. It is used to access the results of the HIT.

- **Request parameters**
  `HITId` - To identify the HIT whose assignments need to be accessed
  `SortProperty` - The field by which the assignments may optionally be sorted (e.g. accept time, submit time, etc.)
  `SortDirection` - Ascending or descending order of sort property

- **Response**
  The operation returns the `GetAssignmentsForHITResult` element which includes the set of assignments of the HIT. Each `Assignment` element includes the `AssignmentId`, the corresponding `WorkerId` and the submitted answers.

**ApproveAssignment**

- **Description**
  The `ApproveAssignment` operation approves the result of an assignment and pays the HIT reward to the worker. The operation also transfers the commission to AMT.

- **Request parameters**
  `AssignmentId` - To identify the assignment to be approved

- **Response**
  The operation returns an `ApproveAssignmentResult` element which contains the request element, if applicable.

**RejectAssignment**

- **Description**
  The `RejectAssignment` operation rejects the result of an assignment. An assignment may be rejected if its results do not match the expectations of the requester. The worker is not paid for the assignment.

- **Request parameters**
  `AssignmentId` - To identify the assignment to be rejected

- **Response**
  The operation returns a `RejectAssignmentResult` element which contains the request element, if applicable.

**ForceExpireHIT**

- **Description**
  Forcing a HIT to expire causes the HIT to be removed from the AMT marketplace with immediate effect. If workers have already accepted an assignment belonging to the HIT prior to forced expiration, they are allowed to complete it, return it or abandon it.

- **Request parameters**
  `HitId` - To identify the HIT to be disabled

- **Response**
  The operation returns a `ForceExpireHITResult` element which contains the request element, if applicable.

# Chapter 3

# CrowdDB

An overview of CrowdDB is presented in this chapter. Section 3.1 presents concrete use cases to motivate the need for a system like CrowdDB. The detailed architecture of CrowdDB, along with the functionalities of each of its components, is provided in Section 3.2. A prototype of CrowdDB has been implemented using the H2 database [28]. Section 3.3 provides a summary of the features of H2.

## 3.1 Use Cases

CrowdDB is a hybrid system that takes advantage of the disparate capabilities of humans and machines to answer difficult queries. Examples of some such queries are provided in this section. Consider the following schema representing `movie` entities.

```
 CREATE TABLE movie(
  title VARCHAR(255) PRIMARY KEY,
  category VARCHAR(255),
  year_of_release INTEGER,
  director_name VARCHAR(255),
  running_time INTEGER
);
```

### 3.1.1 Finding Missing Data

Relational databases presume that the data they hold is complete. Yet, data corresponding to an existing entity or tuple in a relation maybe missing and hence, stored as a `NULL` value. Furthermore, entire entities or tuples may be missing from the relation. Consider the following query executed against the `movie` relation.

```
SELECT * FROM movie
WHERE title = 'The Godfather';
```

In a traditional database if the relation contains a tuple with title `'The Godfather'`, the tuple is returned. The tuple may still contain `NULL` values representing missing data. If no tuple exists satisfying the `WHERE` clause, an empty result set is returned. In CrowdDB, crowdsourcing is used to complete incomplete tuples or to find missing tuples. By including crowdsourcing into query execution, CrowdDB makes sure that only complete data is returned to the application.

### 3.1.2 Entity Resolution

There may be multiple ways to refer to the same real-world entity, possibly bringing about a discrepancy in the way the data is stored and referred to. For instance, `'Peter Jackson'`, `'P. Jackson'` and `'Sir Jackson'` are all valid `director_name` values for the same director.

```
SELECT * FROM movie
WHERE director_name = 'Peter Jackson';
```

If an application submits the above query to a traditional database, tuples with `director_name` set to `'Peter Jackson'` are returned. Unfortunately, tuples that may have variants of `'Peter Jackson'` as `director_name` are ignored. In CrowdDB, it is possible to resolve `'Peter Jackson'`, `'P. Jackson'` and `'Sir Jackson'` to be the same entity by crowdsourcing entity resolution tasks as a part of query processing. Hence, the result set returned is more comprehensive.

### 3.1.3 Subjective Rankings

Queries that aim to find the *best* movie or rank the *top-k* movies based on a certain aspect (e.g. best screenplay) also classify as difficult queries for a traditional database. It is possible to enhance the schema and make it richer by introducing a new column that stores ratings of movie screenplays. Unfortunately, this is not a scalable solution since the number of possible aspects is not bounded. CrowdDB makes it possible to perform such subjective comparisons based on opinions from the crowd, as a part of query processing, making it fundamentally different from traditional databases.

## 3.2 System Architecture

CrowdDB taps the advantages of being based on traditional relational databases. To a large extent, the architecture of CrowdDB follows the textbook architecture of relational databases [32]. Figure 3.1 provides an overview of the CrowdDB architecture. The figure shows how data from the crowd is integrated into a traditional database system. This makes it possible to query both electronic and crowd data sources in a similar manner.

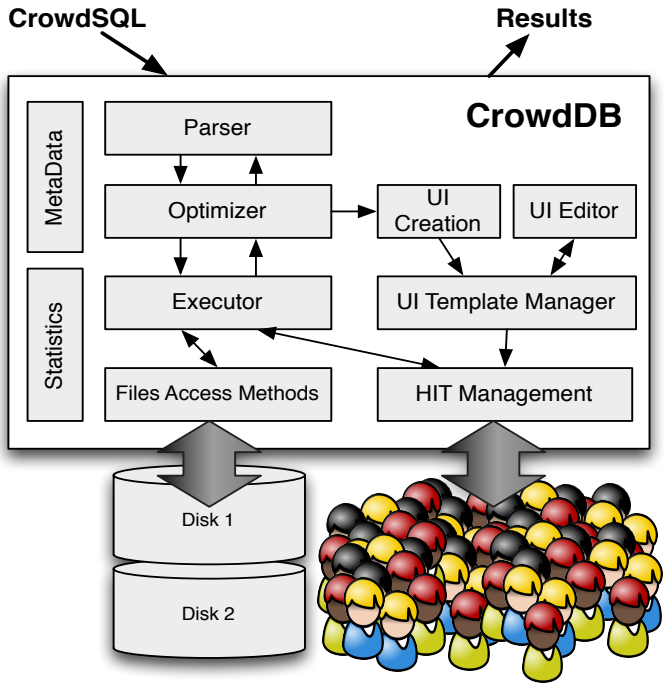


Figure 3.1: CrowdDB architecture

Figure 3.2 shows the components of the CrowdDB system and depicts the control flow of query execution. A detailed description of the role of each of the components is provided in this section.

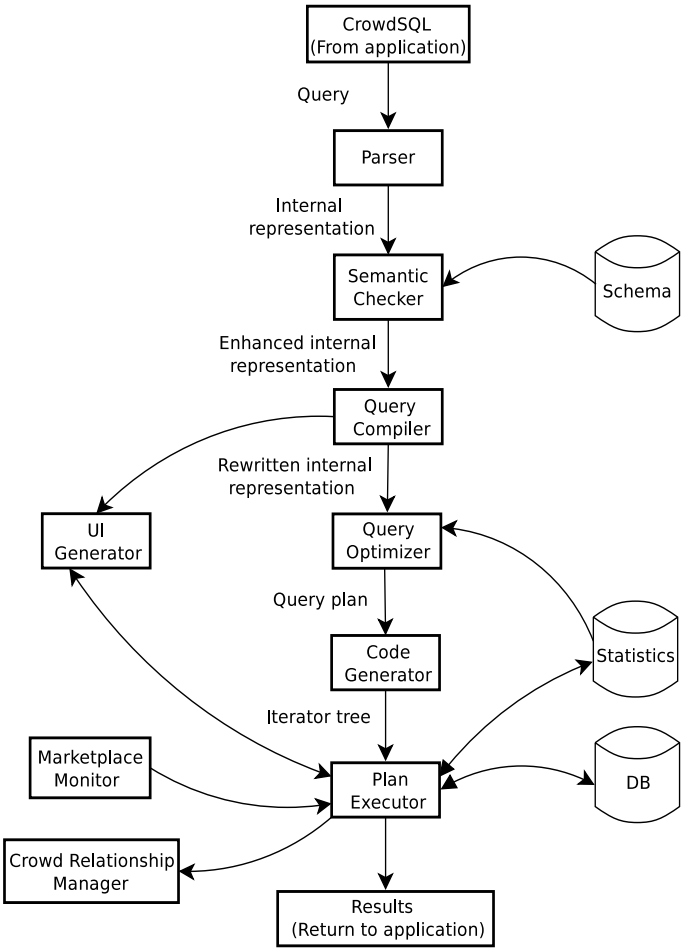


Figure 3.2: CrowdDB components and control flow

### 3.2.1 Parser

**Input** The parser, as in a traditional relational database system, takes the query from the application as input. The query is written in CrowdSQL, an extension of SQL described in Chapter 5.

**Function** The operation of the parser involves recognizing clauses in the query. For a typical Select-Project-Join query, the parser constructs the parse tree which contains details of the attributes to be projected, the names of one or more relations to be queried, along with the details of the conditional clauses, if any. During this process, the parser also validates syntax. If the syntax is invalid, the parser returns the appropriate error to the application.

**Extensions** Since CrowdSQL introduces new keywords in the DDL, DML and queries, the functionality of the traditional parser needs to be extended. Examples of the new keywords and conditions introduced in CrowdSQL include `CROWD`, `CROWDORDER` and $\sim=$. Details of the same are covered in Chapter 5.

**Output** The parser produces an internal representation of the query expressed as a parse tree. The parse tree is constructed based on a well-defined set of extended grammar rules for CrowdDB. The rules ensure the syntax of the query submitted to CrowdDB is correct.

### 3.2.2 Semantic Checker

**Input** The semantic checker takes, as input, the parse tree representation of the query.

**Function** The role of the semantic checker primarily involves ensuring that the query is semantically correct. Some examples of semantic checks that may be performed in a traditional database system include type checking, ensuring no inconsistencies exist in the query and ensuring that the projected attributes exist in the queried relations. In some cases, the semantic checker also adds details to the internal representation of the query. Additional details may include primary and foreign key constraints, correlations between subqueries, etc. If not semantically correct, the checker returns the appropriate error to the application and terminates query execution.

**Extensions** Apart from the functions listed above, the CrowdDB semantic checker performs additional checks on queries directed at crowdsourced relations. The additional conditions under which the semantic checker throws an error in CrowdDB are covered in Chapter 5.

**Output** As a result of semantic checking, a possibly enhanced parse tree representation is produced.

### 3.2.3 Query Compiler

**Input** The output of the semantic checker, essentially a parse tree, is fed as input to the query compiler.

**Function** The query compiler *rewrites* the query by applying a series of rules that transform the internal representation of the query to a form that is easier to optimize. Based on a set of algebraic laws, it replaces the nodes in the parse tree with relational algebra operators. Rewrite rules may consider predicate pushdown, join ordering, etc. Traditionally, the module also works to rewrite computationally intensive operations including aggregations, sorting, etc. It may also consider the kind of cache investment strategy required by the system.

**Extensions** In CrowdDB, the traditional query compiler is extended by adding crowd-specific rewrite rules. Querying crowdsourced relations requires special crowd operators which are used to express the output of the query compiler. Special crowd operators are explained as a part of the extended relational algebra of CrowdDB in Chapter 5. An important functionality of the CrowdDB query compiler is the initialization of UI generation process when one or more crowdsourced relations are queried. The UI generator is invoked by the query compiler to prepare a *UI template* based on the compiled query. The template is initialized with names of the queried relations and their attributes.

**Output** As an output, the query compiler produces a rewritten internal representation in the form a logical query plan. Additionally, if one or more crowdsourced relations are queried, the UI template pertaining to the query is produced.

### 3.2.4 Query Optimizer

**Input** The query optimizer in CrowdDB takes the logical query plan as input.

**Function** The query optimizer in a traditional database typically optimizes the logical query plan by using a cost-based optimization model or a rule-based optimization model. Compile-time optimizations compare different combinations of scans, joins, etc. to yield the best possible plan.

**Extensions** In addition to the traditional compile-time optimizations of the query optimizer, optimizations specific to crowd operators need to be included. The application developer is required to provide *configuration parameters* that include information about the number of HITs to be posted per group, the reward for each HIT, the time when the HIT needs to be posted, the size of the expected result set, the maximum permissible response time, the expected quality of the results, etc. Based on the configuration parameters and the metadata available, the query optimizer makes an estimation of the best query plan. The CrowdDB query optimizer also has an important run-time functionality. Query plans in CrowdDB need to be dynamically modified at run-time by including an adaptive query optimizer that considers *observable parameters* based on the state of the AMT marketplace. The Marketplace Monitor component in the CrowdDB architecture is responsible for measuring the observable parameters.

**Output** The query optimizer produces an optimized query plan.

### 3.2.5 Code Generator

**Input** The code generator takes a query plan as input.

**Function** In a traditional database management system, the code generator is responsible for generating executable code.

**Extensions** The CrowdDB code generator component extends the same functionality to cater to crowd-specific operations as well.

**Output** The code generator produces an iterator tree, which is then accessed by the run-time system.

### 3.2.6 Plan Executor

**Input** The plan executor or the run-time system accepts an iterator tree, as input.

**Function** The plan executor executes the iterator tree using the `open()`, `close()` and `next()` calls and produces query results.

**Extensions** The CrowdDB plan executor, in addition, executes iterators of new physical query plan operators that permit interaction with the crowd *during* query processing. A detailed explanation of the new operators is provided in Chapter 7. Plan execution with crowd operators includes invoking the run-time functionality of the UI generator to instantiate the UI templates as descriptively as possible. The instantiated UIs are HTML forms which are posted to AMT during query processing. Workers on AMT complete the HTML forms and submit their answers. Following this, the database is updated with the newly crowdsourced information. The run-time system updates metadata and statistics.

**Output** The plan executor retrieves relevant tuples, possibly from crowdsourced relations. The retrieved tuples comprise the result set which is returned to the application.

### 3.2.7 Crowd Relationship Manager

CrowdDB functions by repeatedly interacting with the crowd on a crowdsourcing platform such as AMT. Such interaction requires maintaining amicable relationships with workers. The crowd relationship manager is responsible for keeping track of all crowd interactions at a per-worker level. Obtaining details of all the assignments completed by a worker, providing meaningful feedback and

bonuses, where applicable, are some of the tasks of the crowd relationship manager. Malicious behavior and spamming are also concerns for CrowdDB. The relationship manager uses the AMT API to permit only workers with good track records, based on previous interactions, to work on posted HITs.

### 3.2.8 Marketplace Monitor

The marketplace monitor of CrowdDB is responsible for measuring observable parameters based on the state of AMT to enable query plans to be adaptively optimized at run-time. Observable parameters include sensing the number of workers available, the number of existing HITs belonging to a certain HIT type, etc. The marketplace monitor logs these details and periodically feeds them to the adaptive query optimizer. The adaptive query optimizer may, in turn, choose to dynamically modify the query plan to better suit the current state of AMT.

## 3.3 Prototype Implementation

The current prototype implementation of CrowdDB extends the code base of H2 [28], an open-source Java-based database. The choice of underlying database was largely driven by the language of its implementation. CrowdDB is required to generate UIs and interact with the AMT platform at run-time. Since Java, as a programming language, has maximum support for web-based interactions, it was preferred as the implementation language. This section provides an overview of the features of H2 which are applicable to the current prototype implementation of CrowdDB.

H2 supports standard SQL and JDBC connectivity. H2 can be used in the *embedded* mode where the database is started within the application. The database is opened from the same JVM that the application is running on. Multiple sessions can access the same database if they belong to the same process. H2 can also be used in the *server* mode where the database runs as a different process on the same or different JVM. Clients may connect to the database over TCP/IP and hence this mode is slower than the embedded mode. A browser-based console can be used to access the database in server mode. Both modes of operation support the creation of in-memory and persistent databases. H2 synchronizes database access when multiple connections to the same database are active, i.e., only a single thread representing a query is allowed to execute at a time. H2 allows using connections from a connection pool in order to improve performance if the application opens and closes a lot of connections.

SQL support in H2 includes primary and referential integrity constraints with cascade options, inner and outer joins, aggregate functions and built-in functions. Indexes are created automatically for primary key and uniqueness constraints. H2 also has a cost-based optimizer which uses a brute-force effort to calculate the cost for simple queries (up to seven joins). For others, the first few tables are evaluated based on cost and the rest are added to the plan using a greedy algorithm. Other optimizations include evaluating expressions only once if all parameters are constant, avoiding table access if the `WHERE` clause is false, etc. Additionally, H2 has features that support security (authentication, encryption, etc.) and multi-version concurrency control. Transactions are supported in H2 by employing read locks (shared) and write locks (exclusive) for each connection. Locks have a timeout and hence, deadlock states is avoided using the lock timeout exception.

H2 often performs better than other databases for a single connection benchmark run on one computer and PolePosition benchmark [29]. H2 is also compatible with IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle and PostgreSQL.

# Chapter 4

# CrowdDB Data Model

The CrowdDB data model is based on the relational model. Section 4.1 describes the open-world nature of CrowdDB. Section 4.2 details the particular importance that functional dependencies and normalization gain in CrowdDB.

## 4.1 Open-World Assumption

Despite the inherent similarities between CrowdDB and traditional relational databases, there exist certain fundamental differences. Traditional relational databases are built on the closed-world assumption, i.e., if the database does not contain the data, then it is assumed that the data does not exist. Accordingly, the data in a traditional database is assumed to be complete and correct. Contrary to that, CrowdDB is built on the open-world assumption, i.e., the truth value of a statement does not depend on whether it is known or unknown. Consequently, with the use of crowdsourcing, there is an unbounded amount of data that can satisfy the query. Such an open-world assumption has a significant impact on query semantics, query planning and processing. The following chapters delve into the details of the CrowdDB query language, its semantics and the different phases of query execution.

## 4.2 Functional Dependencies and Normalization

Much like traditional databases, functional dependencies play an important role in CrowdDB in designing good relation schemas. Normalization is needed to ensure that issues such as redundancy, update anomalies and deletion anomalies are eliminated. Good schema design in CrowdDB has an impact on both query processing and automatic UI generation.

Relations in CrowdDB are expected to be in BCNF. In the presence of normalization, all one-to-one relationships are included in the schema of a single relation. In case of a one-to-many relationship, two relations are created and linked by a referential integrity constraint. Finally, in case of a many-to-many relationship, an extra relation is created which links primary keys in both the participating relations.

# Chapter 5

# CrowdSQL

CrowdSQL is an extension of SQL that allows crowdsourced relations to be defined, manipulated and queried in CrowdDB. Section 5.1 provides a short overview of the SQL Standard. Section 5.2 explains the semantics of `CNULL`, a new keyword used in CrowdDB. Extensions made to the Data Definition Language (DDL), Data Manipulation Language (DML) and query language of SQL are presented in Section 5.3, Section 5.4 and Section 5.5 respectively.

## 5.1 SQL Standard

SQL or Structured Query Language is the most widely used language for managing data in a relational database management system. SQL was adopted as an ANSI-standard in 1986 and has since undergone a number of revisions. SQL is a set-based declarative query language. The basic language elements in SQL include clauses, expressions, predicates, queries and statements. Querying relations in a database is done using the `SELECT` statement which can optionally be coupled with different clauses, expressions and predicates. SQL supports a three-valued logic consisting of `TRUE`, `FALSE` and `NULL`. The `NULL` value is used to indicate unavailable information.

Using SQL as a starting point for enabling data definition and manipulation in CrowdDB has several advantages. Firstly, the declarative nature of SQL specifies *what* the query should accomplish and not *how* it should be accomplished. Accordingly, SQL allows for physical and logical data independence such that application programs remain resistant to changes that the APIs of crowdsourcing platforms may incorporate. Even a complete switch of the crowdsourcing platform employed would require minimal changes to the application source code. Secondly, by using SQL, it is possible to extend the concise semantics of all existing relational database operators including joins, aggregates and sorts. This also makes it possible to extend existing query processing and optimization techniques in relational databases, a well-researched domain.

## 5.2 Extended Keywords - CNULL

In CrowdDB, relations may be incomplete, i.e., one or more attribute values may be missing in existing tuples. It is also possible that entire tuples may be missing from the relation. CrowdDB allows the missing data to be completed using crowd input.

In order to indicate that data is missing, `CNULL` is used as a special placeholder. `CNULL` indicates that CrowdDB does not currently have the desired value but is capable of finding it using crowdsourcing. The inclusion of `CNULL` in CrowdSQL leads to a four-valued logic, unlike traditional relational databases. Similar to the behavior of `NULL`, `CNULL` is treated as a valid value in any *non-key* crowdsourced attribute of a relation. Hence, attributes with primary key constraints or uniqueness constraints are not permitted to hold `CNULL` values. Apart from this similarity, the semantics of `NULL` and `CNULL` are fundamentally different. While `NULL` represents a value that does not exist or is un-

available, CNULL indicates that the value can, in the future, be found out as a side-effect of query processing with the crowd. The semantics of query processing in CrowdDB are such that incomplete tuples, i.e., tuples with CNULL values, are never returned as a part of the result set. Once the incomplete tuples have been completed by the crowd, they are included in the result set. On the other hand, a tuple with one or more NULL values may still be part of the result set.

## 5.3 Extended DDL

CrowdDB is built upon the premise that data in the database may be obtained from electronic data sources (much like traditional databases) or from human input (using crowdsourcing platforms). In order to cater to the latter, DDL extensions permitting the definition of crowdsourced data need to be added to SQL. The traditional SQL DDL consists of CREATE, ALTER, DROP and TRUNCATE commands. CrowdSQL DDL allows specific attributes or entire relations to be marked as crowdsourced by using the CROWD keyword.

### 5.3.1 Crowd Column

The CROWD keyword can be used to mark one or more columns of a table as crowdsourced. Such columns contain crowd input. Values in crowdsourced columns can be set to CNULL if they have not yet been crowdsourced. If tuples contain CNULL values, they are typically replaced with crowdsourced values as a side-effect of query processing in CrowdDB. In particular, if a query accesses a crowdsourced attribute that has been set to CNULL, then CrowdDB will ask the crowd to complete it and will store it for future use.

The following SQL code creates a table movie with two crowd columns, year_of_release and director_name. Such a table classifies as a regular table with crowd columns. Given the schema, it is possible that year_of_release and director_name values for one or more tuples in the table may be set to CNULL. The other columns are regular columns and bear the same semantics as they do in relational databases, i.e., they are expected to contain complete information.

```
CREATE TABLE movie(
  title VARCHAR(255) PRIMARY KEY,
  year_of_release CROWD INTEGER,
  category VARCHAR(255),
  director_name CROWD VARCHAR(255),
  running_time INTEGER
);
```

The DEFAULT keyword can also be used to set the default value of a crowdsourced column to CNULL. The following code snippet provides an example of the usage of the DEFAULT keyword with crowdsourced columns.

```
CREATE TABLE movie(
  title VARCHAR(255) PRIMARY KEY,
  year_of_release CROWD INTEGER,
  category VARCHAR(255),
  director_name CROWD VARCHAR(255) DEFAULT CNULL,
  running_time INTEGER
);
```

### 5.3.2 Crowd Table

A crowd table can be created in CrowdDB when all the attributes of the relation are crowdsourced. Crowd tables are capable of dealing with two kinds of missing data. The first kind is when the tuple

exists in the relation but is incomplete. In such a case, CNULL is used to indicate missing values in the non-key columns. The second kind is when entire tuples are missing from the relation. By declaring a relation as crowdsourced, new tuples can be added to the relation by the crowd. The second kind of missing data makes crowd tables virtually infinite since the number of tuples that may be added by the crowd knows no bounds. Both kinds of missing data are completed as a side-effect of query processing in CrowdDB. Of interest to note is that regular tables with crowd columns do not have to deal with the open-world issue since the number of tuples is bounded by the regular columns.

In CrowdDB, the open-world issue in crowd tables is dealt with by enforcing certain restrictions. Firstly, to restrict crowd input, every crowd table *must* have a primary key constraint. The mandatory primary key serves as an intuitive way to identify entities that have already been crowdsourced and hence, avoids repeated crowdsourcing of the same entity. Secondly, to restrict the data returned upon query execution, queries on crowd tables are required to specify the size of the expected result set. For this purpose, such queries *must* have a LIMIT clause. The semantics of the LIMIT clause in the context of queries on crowd tables are explained in Section 5.5.

```
CREATE CROWD TABLE movie(
  title VARCHAR(255) PRIMARY KEY,
  year_of_release INTEGER,
  category VARCHAR(255),
  director_name VARCHAR(255),
  running_time INTEGER
);
```

### 5.3.3 Crowdsourced Integrity Constraints

CrowdDB allows integrity constraints to be enforced upon relations in the database. The semantics and implications of integrity constraints on crowd columns and crowd tables are explained in this section.

#### Primary Key Constraints

For primary keys defined on regular columns, the semantics remain the same as in relational databases. Such columns, by definition, implicitly hold a NOT NULL constraint. In CrowdDB, a crowdsourced column, whether in a regular table or a crowd table, can be bound by a primary key constraint. Since a primary key attribute is essential in *identifying* entities, allowing its value to be set to CNULL serves no purpose. Accordingly, in order to abide by the implicit uniqueness requirement of the primary key constraint, such columns are not allowed to hold CNULL values. As a result, for regular tables, specifying a crowd column as a primary key has no impact whatsoever. On the other hand, for crowd tables, it is possible for the crowd to insert new tuples into the table and hence add new primary keys that do not violate the primary key constraint.

#### Foreign Key Constraints

CrowdDB also allows referential integrity between relations to be specified using foreign key constraints. The referenced and referencing columns of the foreign key constraint may be regular or crowdsourced. If the referencing column is crowdsourced, it is possible that it holds CNULL values hence indicating a missing reference. The completion of missing references occurs as a side-effect of query processing. If the referenced column belongs to a crowd table, it is also possible to add new keys to the referenced table during query processing. Section 5.5.6 provides the detailed semantics of query processing in the presence of foreign key constraints.

## 5.4 Extended DML

The CrowdSQL DML includes the INSERT, UPDATE and DELETE commands.

### 5.4.1 Insert

The INSERT command is used to add new tuples to existing relations. Values inserted into crowd columns may be set to CNULL, except if the column also holds a primary key constraint. Given the schema of the movie relation in Section 5.3.2, the following INSERT commands can be used to insert data into it.

```
INSERT INTO MOVIE
VALUES('Pulp Fiction', 1994, 'Drama', 'Quentin Tarantino', 154);

INSERT INTO movie
VALUES('Schildlers List', Null, 'History', CNULL, CNULL);
```

INSERT commands on crowd tables that attempt to set all values in the tuple to CNULL are disallowed since the primary key of a crowdsourced tuple can never be set to CNULL. Hence, the following command would throw an exception upon execution.

```
INSERT INTO movie
VALUES(CNULL, CNULL, CNULL, CNULL, CNULL);
```

### 5.4.2 Update

The UPDATE command retains the semantics that it has in relational databases. In addition though, CrowdDB permits values in crowd columns to be updated to CNULL. It is disallowed to update the value of a non-crowdsourced column to CNULL. The following is an example of a valid UPDATE command.

```
UPDATE movie SET year_of_release = CNULL
WHERE director_name = 'Frank Darabont';
```

### 5.4.3 Delete

CrowdDB executes the DELETE command in the same way as relational databases. Deleting tuples that contain CNULL values is permitted. The following is an example of a valid DELETE command.

```
DELETE FROM movie
WHERE director_name = 'Frank Darabont';
```

## 5.5 Extended Query Language

CrowdSQL uses the SELECT command to query data from regular and crowdsourced relations. In this section, the semantics of using the SELECT command to query regular tables with crowd columns and crowd tables are explained.

The main difference in the behavior of the SELECT command stems from the possible presence of CNULL values in selected crowdsourced attributes. If a SELECT command encounters one or more missing values denoted by CNULL, it aims to complete the missing information using crowdsourcing. This causes the SELECT command to inflict changes upon the queried tables, hence making its semantics fundamentally different from those in traditional databases.

The basic syntax for the SELECT command in SQL is retained in CrowdSQL. The grammar that defines the valid SELECT commands is described in Section 6.1. The SELECT command typically consists of the *select-list* with attribute names from the queried relations, the *from-list* with the names of the relations to be queried and an optional *qualification* expressed as a boolean condition in the WHERE clause. Furthermore, the command may also contain optional ORDER BY and GROUP BY clauses. In addition, CrowdDB introduces new functions, CROWDEQUAL and CROWDORDER, to query the crowd for subjective comparisons and rankings respectively. The usage of the new functions is explained in Section 5.5.3 and Section 5.5.4.

Since crowd tables are unbounded, it is necessary to explicitly limit the size of the result set when they are queried. For this purpose, CrowdDB places an additional syntactic requirement which states that queries on crowd tables *must* contain a LIMIT clause. Using the LIMIT clause, the application developer specifies the *exact* size of the expected result set. The argument of the LIMIT clause is used during query processing to infer the number of tuples that need to be crowdsourced, based on the number of tuples already available in the crowd table. An exception to the syntactic requirement of needing a LIMIT clause is when the query contains an equality predicate on a primary key attribute. In this case, CrowdDB *infers* the size of the expected result set to be 1. During query processing, if the number of tuples in the result set is less than the number specified by the LIMIT clause, new tuples are crowdsourced. The final result set with *exactly* the LIMIT number of tuples is returned to the application. Consequently, the semantics of the LIMIT clause vary in the context of regular and crowd tables in CrowdDB. In the former case, the LIMIT clause specifies the upper limit for the number of tuples to be returned in the result set. In the latter case though, the LIMIT clause specifies the exact number of tuples expected in the result set.

### 5.5.1 Simple SELECT

This section contains examples of simple queries executed on a regular table with crowd columns and a crowd table.

**On Regular Tables with Crowd Columns** The following examples illustrate the valid syntax of the SELECT command in CrowdDB, when executed upon the movie relation created in Section 5.3.1. The syntactic requirement of needing a LIMIT clause does *not* apply to queries on regular tables with crowd columns, since the presence of one or more regular columns in the schema automatically limits the data to be returned. Upon execution of such queries, CrowdDB detects any missing information and crowdsources it before returning the result set to the application.

```
SELECT * FROM movie;

SELECT DISTINCT(director_name) FROM movie;
```

**On Crowd Tables** Consider the schema in Section 5.3.2. SELECT queries on such a schema are required to have a LIMIT clause. In the absence of the LIMIT clause, CrowdDB throws an exception. The following queries are syntactically valid queries, given the schema.

```
SELECT * FROM movie LIMIT 10;

SELECT DISTINCT(director_name) FROM movie LIMIT 5;
```

As expressed in the queries, the expected size of the result sets is 10 and 5 tuples respectively. Upon executing such queries, if CrowdDB detects that the result set has insufficient number of tuples, it crowdsources the remaining tuples. New tuples are added by the crowd to the movie relation. After crowdsourcing, CrowdDB returns the result set with exactly as many tuples as required by the application.

### 5.5.2 WHERE Clause

Executing a SELECT query with a WHERE clause on a regular table with crowd columns or on a crowd table is explained in this section.

**On Regular Tables with Crowd Columns** Consider the schema presented in Section 5.3.1. The following examples illustrate the effects of querying it using a SELECT command with a WHERE clause.

```
SELECT * FROM movie
WHERE title = 'The Shawshank Redemption';
```

Since title is the primary key of the movie table, there can, at most, be a single tuple that satisfies the above query. If it exists in the table, the tuple is returned. If the tuple contains CNULL values, the specific fields are crowdsourced before being returned to the application. If no tuple is found to satisfy the predicate, an empty result set is returned.

```
SELECT * FROM movie
WHERE director_name = 'Francis Ford Coppola';
```

The above query specifies a WHERE clause on the director_name column of the movie table. Since the director_name column is crowdsourced, it may contain CNULL values. Hence, as a part of the execution of the SELECT command, the CNULL values are crowdsourced and updated. Following this, all the tuples that satisfy the predicate compose the result set and are returned to the application. The following are other examples of valid queries on regular tables with crowd columns.

```
SELECT * FROM movie
WHERE running_time > 120;
```

```
SELECT * FROM movie
WHERE category IN ('Drama', 'Action', 'Thriller');
```

**On Crowd Tables** SELECT commands with WHERE clauses behave differently when the queried table is entirely crowdsourced. The difference in semantics allows crowd tables to tackle the open-world issue. Consider the table schema presented in Section 5.3.2. The following queries illustrate the behavior of CrowdDB when movie is a crowd table.

```
SELECT * FROM movie
WHERE title = 'The Shawshank Redemption';
```

In this case, since the WHERE clause consists of a predicate involving the primary key of the crowd table, the query processor *infers* a LIMIT value of 1. Accordingly, if a tuple satisfies the predicate, it is returned as part of the result set. As in the previous cases, all CNULL values are crowdsourced and updated before the results are returned to the application. If no tuple is found satisfying the predicate, CrowdDB asks the crowd to insert a single new tuple into the movie relation satisfying the predicate on the primary key.

```
SELECT * FROM movie
WHERE director_name = 'Francis Ford Coppola';
```

Such a query, as mentioned previously, violates CrowdDB requirements. Accordingly, predicates on non-key crowd columns in crowd tables *must* be followed by a LIMIT clause. The above query would throw an exception when executed with CrowdDB.

```
SELECT * FROM movie
WHERE director_name = 'Francis Ford Coppola' LIMIT 5;
```

The above query aims to find *exactly* 5 tuples that have the specified `director_name`. The CrowdDB query processor returns 5 tuples if they already exist in the `movie` table. If not, CrowdDB crowdsources the remaining tuples and returns the result set to the application. The following are other examples of valid queries on the crowd table `movie`.

```
SELECT * FROM movie
WHERE year_of_release < 1990 LIMIT 10;
```

```
SELECT * FROM movie
WHERE director_name LIKE 'Fran%' LIMIT 5;
```

### 5.5.3 CROWDEQUAL

CrowdDB introduces a new binary comparison function called `CROWDEQUAL`, symbolized as '~='. The `CROWDEQUAL` function is useful when a potentially subjective comparison of two values needs to be performed by the crowd. The function is useful when entity resolution is required to be performed by the crowd on noisy data. It takes, as input, the two values to be compared and returns a boolean value. The following is the general form of the `CROWDEQUAL` function.

```
attribute ~= '[a non-uniform value]'
```

Comparisons performed by the crowd as a result of the `CROWDEQUAL` function are cached for future use in auxiliary tables. The processing of the `CROWDEQUAL` function is explained in further detail in Section 7.1.2. `CROWDEQUAL` only functions on *existing* tuples for the purpose of subjective comparison or entity resolution. Its usage does not lead to the crowdsourcing of new tuples. Hence, the `LIMIT` clause is not mandatory, whether the table is regular or crowdsourced. Consider the following query directed at the schema specified in Section 5.3.2.

```
SELECT * FROM movie
WHERE director_name ~= 'Peter Jackson';
```

The above query aims to resolve the entities in the `movie` table based on their `director_name` and find any entities with `director_name` same or similar to `'Peter Jackson'`. Unless cached, the crowd is asked to compare all existing `director_name` values in the relation with `'Peter Jackson'`. The results are cached and tuples that are deemed as being the same entity as `'Peter Jackson'` are returned in the result set. In case any of the tuples in the relation have the `director_name` attribute set to `CNULL`, the value is crowdsourced and updated before being compared.

### 5.5.4 CROWDORDER

CrowdDB introduces another comparison function, `CROWDORDER`, which crowdsources the task of ranking or ordering multiple entities based on a certain aspect. It is used in conjunction with the SQL `ORDER BY` clause. Similar to `CROWDEQUAL`, the `CROWDORDER` function ranks *existing* tuples and never leads to crowdsourcing of new tuples, irrespective of whether it is used on a regular table or a crowd table. The following is the general form of the `CROWDORDER` function.

```
CROWDORDER(attribute, 'aspect')
```

Consider the schema in Section 5.3.2. The following example query illustrates the usage of the `CROWDORDER` function to rank `movie` entities based on their screenplays.

```
SELECT * FROM movie
ORDER BY CROWDORDER(title, 'Best screenplay');
```

The `title` attribute is provided to the crowd to distinguish between the multiple `movie` entities to be ranked. All incomplete tuples are completed using crowdsourcing before the result set is returned to the application.

### 5.5.5 Aggregates

CrowdSQL permits the usage of traditional SQL aggregate functions namely `AVG()`, `COUNT()`, `MAX()`, `MIN()` and `SUM()` on crowd columns. Since such columns may hold `CNULL` values, it is necessary that the aggregation be performed only after the `CNULL` values are replaced by crowdsourced input. The usage of aggregate functions never leads to crowdsourcing new tuples. The following query is an example of using the `AVG()` aggregate function to find the average running time of entities in the `movie` relation created in Section 5.3.2.

```
SELECT AVG(running_time) FROM movie;
```

If the `running_time` attribute value in any of the existing `movie` entities is set to `CNULL`, it is first crowdsourced and updated in the `movie` relation. Once all the values are available, the average is computed and returned to the application.

### 5.5.6 Joins

Much like SQL, in order to combine tuples from two or more relations, CrowdSQL supports joins. Cross joins, natural joins, theta joins and semi joins are the different types of joins that are supported.

Extending the semantics of querying a single crowd table to cater to joins, it is necessary to have a `LIMIT` clause if one or more of the relations participating in the join are crowdsourced. The purpose of enforcing a `LIMIT` clause is to deal with the open-world issue by bounding the size of the expected result set. As a side-effect of join processing, it is possible to add new tuples to one or more of the queried crowd tables, as summarized in Table 5.1.

| | | **Table 1** | | |
|---|---|---|---|---|
| | | **Regular table** | **Regular table with Crowd columns** | **Crowd table** |
| **Table 2** | **Regular table** | Traditional join | Incomplete tuples in Table 1 may be crowdsourced | Incomplete tuples and new tuples in Table 1 may be crowdsourced |
| | **Regular table with Crowd columns** | Incomplete tuples in Table 2 may be crowdsourced | Incomplete tuples in both tables may be crowdsourced | Incomplete tuples in both tables may be crowdsourced,<br>New tuples in Table 1 may be crowdsourced |
| | **Crowd table** | Incomplete tuples and new tuples in Table 2 may be crowdsourced | Incomplete tuples in both tables may be crowdsourced,<br>New tuples in Table 2 may be crowdsourced | Incomplete tuples and new tuples in both relations may be crowdsourced |

Table 5.1: Two-way joins in CrowdDB

In the presence of referential integrity constraints, CrowdDB executes a functional join query. In addition to completing missing information using the semantics specified in Table 5.1, CrowdDB

takes special care to ensure that the missing references that are crowdsourced do not violate any referential integrity constraints. If the referencing column is crowdsourced, it is possible that it contains `CNULL` values. This denotes a missing reference which can be completed as a side-effect of query processing. The options available to the crowd for completing missing references depend on whether the referenced table is regular or crowdsourced. If the referenced table is regular, the crowd completes the missing reference by ensuring it refers to an existing key. Alternately, if the referenced table is crowdsourced, the crowd can complete the missing reference by making it refer to an existing key or by adding a new key to the referenced table. Table 5.2 summarizes the additional restrictions on the completion of missing references during functional join execution based on the nature of the referenced and referencing columns.

| | | **Referencing Table** | | |
|---|---|---|---|---|
| | | **Regular table** | **Regular table with Crowd columns** | **Crowd table** |
| **Referenced Table** | **Regular table** | Traditional join | Incomplete foreign keys must refer to *existing* primary keys | Incomplete foreign keys must refer to *existing* primary keys |
| | **Regular table with Crowd columns** | Traditional join | Incomplete foreign keys must refer to *existing* primary keys | Incomplete foreign keys must refer to *existing* primary keys |
| | **Crowd table** | Traditional join | Incomplete foreign keys can refer to *existing or new* primary keys | Incomplete foreign keys can refer to *existing or new* primary keys |

Table 5.2: Additional restrictions on two-way functional joins in CrowdDB

Consider the following example. The following SQL code creates a crowd table `director` with `name` as a primary key. Another crowd table `movie` is created with a foreign key constraint that ensures `director_name` references values in the `name` column of the `director` table.

```
CREATE CROWD TABLE director(
  name VARCHAR(255) PRIMARY KEY,
  place_of_birth VARCHAR(255),
  year_of_birth INTEGER
);

CREATE CROWD TABLE movie(
  title VARCHAR(255) PRIMARY KEY,
  year_of_release INTEGER,
  category VARCHAR(255),
  director_name VARCHAR(255) REFERENCES director(name),
  running_time INTEGER
);
```

Consider the following functional join query executed on the above crowd tables.

```
SELECT * FROM movie JOIN director
WHERE movie.director_name = director.name;
```

As a side-effect of processing the above query, it is possible that existing `CNULL` values in the crowd tables are replaced by crowd input. If the `director_name` attribute value is missing for one or more tuples, the crowd can complete it by referencing it to an existing `director` entity or by adding a new tuple to the `director` relation and then referencing it. The details of processing such a query are presented in Chapter 7.

# Chapter 6

# CrowdDB Query Compiler

The compile-time functionality of the CrowdDB query processor extends that of a traditional database management system [32] by allowing it to appropriately handle queries written in CrowdSQL and build query plans. The functioning of the CrowdDB query processor is broadly divided into four stages. The first stage is parsing, covered in Section 6.1, where the CrowdSQL query is checked for valid semantics and a parse tree is constructed. Crowd parse trees provide a structured way of representing valid queries written in CrowdSQL. The CrowdDB compiler then invokes the UI generator, presented in Section 6.2, to initialize the process of UI template generation. The UI generator is an additional component in CrowdDB that is required to enable integration of the crowd into a traditional database system. The next step in the compile-time activities of CrowdDB is the generation of logical query plans as explained in Section 6.4. Crowd relational algebra is used to represent the logical query plan as an expression tree and is presented in Section 6.3. The final step of query compilation involves converting the logical query plans into physical query plans, which include details of ordering the operations during query execution and the algorithms to be used for each of the operations. An overview of physical query plan generation is provided in Section 6.5 and its details are covered in Chapter 7.

## 6.1 Parser

Parsing is the first step of query compilation. The CrowdDB parser builds a parse tree representation of the CrowdSQL query by representing the lexical elements in the query as nodes of the tree. The leaf nodes of the parse tree represent keywords, attribute names, relation names, etc. while the interior nodes represent query clauses or subparts such as the list of attributes to select, the list of tables to query or if present, the condition to be evaluated. The parser ensures syntactic correctness of the CrowdSQL query by checking that keywords, operators and identifiers are used in the appropriate position, with respect to the grammar rules.

### 6.1.1 Extended Grammar

A detailed specification of SQL grammar is presented in [15]. The following is the grammar for a simple subset of CrowdSQL expressed in the Backus-Naur Form.

```
<query> ::= SELECT <select_list> FROM <from_list>
            [WHERE <condition>] [ORDER BY <order_by_clause>]
            [LIMIT NUMBER]

<select_list> ::= "*" | <column_name>["," <column_name>]
<column_name> ::= IDENTIFIER | <crowd_column_name>
<crowd_column_name> ::= IDENTIFIER
```

```
<from_list> ::= <table_name>["," <table_name>][JOIN <table_name>]
<table_name> ::= IDENTIFIER | <crowd_table_name>
<crowd_table_name> ::= IDENTIFIER

<condition> ::= <condition> [(AND|OR) <condition>]
<condition> ::= <column_name> <op> (<column_name> | IDENTIFIER)
<op> ::= IN | NOT IN | LIKE | = | ~= | < | <= | > | =>

<order_by_clause> ::= <column_name>["," <column_name>]
                    | CROWDORDER (<column_name>, TEXT)
```

A query in CrowdSQL consists of the `SELECT` keyword followed by a list of crowd or regular columns to be selected. Following that, is the `FROM` keyword and a list of one or more crowd or regular tables upon which the query is to be executed. The query may optionally include a `WHERE` predicate followed by a condition to further filter tuples from the selected result set. Another optional clause is the `ORDER BY` clause. The result set may be ordered by a certain column or by the `CROWDORDER` function. Owing to the open-world nature of CrowdDB, the `LIMIT` clause gains particular importance in the context of crowd tables, as elaborated in Chapter 5.

### 6.1.2 Parse Trees

To represent the syntactic structure of the CrowdSQL query, the parser constructs parse trees using the set of grammar rules specified in Section 6.1.1. If the CrowdSQL query does not abide by the grammar rules, the parser aborts query execution and returns an exception. Consider the following example of a CrowdSQL query based on the schema specified in Section 5.3.2.

```
SELECT director_name FROM movie
WHERE title = 'The Godfather';
```

The parse tree for the query is represented as in Figure 6.1. The query is split into its constituent parts based on the grammar rules. In this case, the `from_list` consists of a single crowd table and the `select_list` consists of a single crowd column.

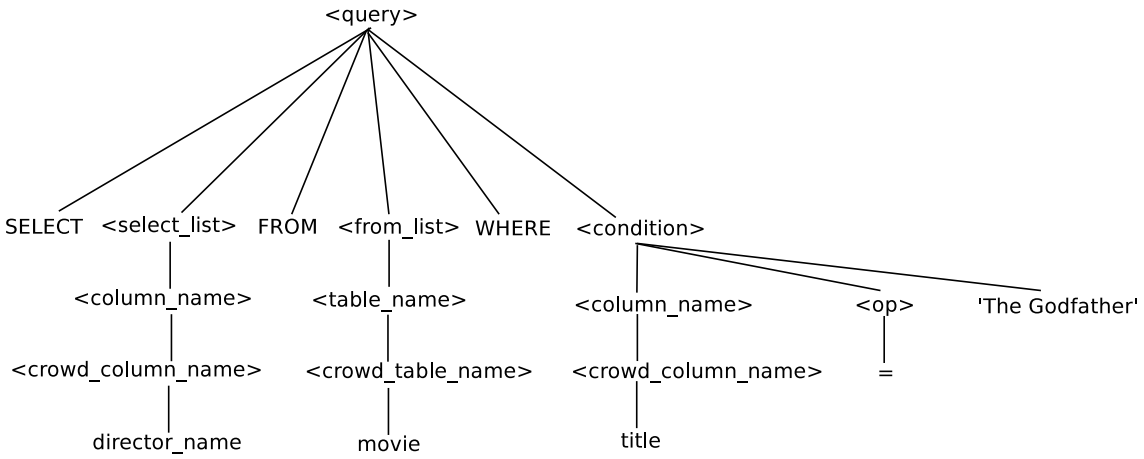


Figure 6.1: Parse tree for a query on a single relation

A CrowdSQL join query directed at the schema specified in Section 5.5.6 is given below. The corresponding parse tree is shown in Figure 6.2. The `from_list` consists of two crowd tables, `movie` and `director`. The `select_list` includes all the attributes from the two crowd tables and the join condition checks for the same director name in the two tables.

```
SELECT * FROM movie JOIN director
WHERE movie.director_name = director.name;
```

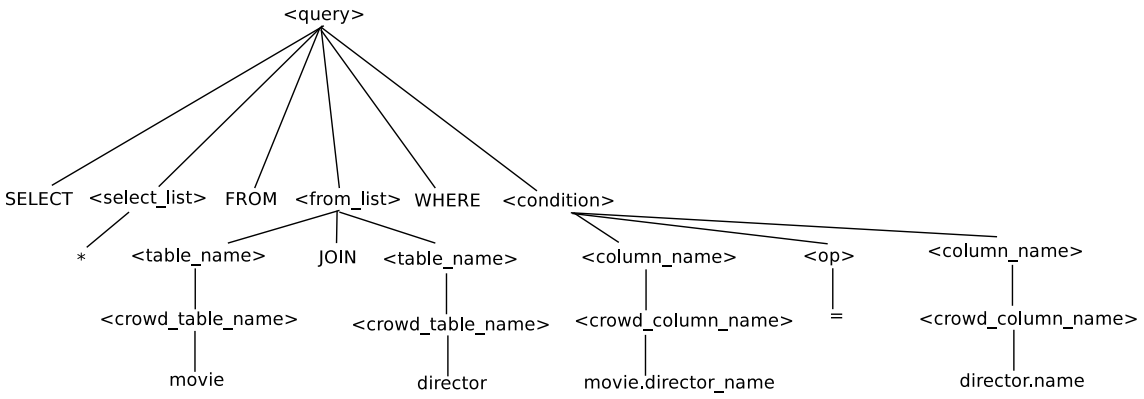


Figure 6.2: Parse tree for a join query

### 6.1.3 Semantic Checking

Apart from validating the syntax of the CrowdSQL query, the parser also performs basic semantic checking. It checks to ensure that the tables queried in the `from_list` are legitimate tables in the schema (either regular tables or crowd tables). If the queried relation is a crowd table and a predicate exists on a non-key attribute, the parser checks for the presence of a `LIMIT` clause. The `LIMIT` clause is needed to circumvent the open-world assumption of data as explained in Section 5.5. The absence of a `LIMIT` clause causes the parser to throw an exception. Another aspect of semantic checking involves resolving the attribute names in the `select_list` and ensuring that each attribute belongs to a single relation in the `from_list`. The parser also checks for ambiguity by ensuring that each attribute is *not* in the scope of more than one relation. Type checking is another important aspect of semantic checking. An example of type checking is to ensure that operators, such as equality, are only applied to compatible types. Also, semantic checking in CrowdDB ensures that `CNULL` can only be inserted or updated in crowd columns. It is a reserved keyword in CrowdDB and hence is not allowed to be used in regular columns.

## 6.2 User Interface Generation

Interaction with the crowd in CrowdDB makes automatic UI generation a vital component of its processing model. CrowdDB is capable of automatically creating UIs for regular tables with crowd columns or crowd tables. The UI generator creates *templates* at compile-time, which are instantiated at run-time to either crowdsource new tuples or aid in the completion of existing incomplete tuples. Instances of the UI templates are also used to gather input for the special crowd comparison functions, namely `CROWDEQUAL` and `CROWDORDER`. This section focuses on the compile-time activity of UI generator, i.e., the generation of UI templates based on the query and the schemas of the queried relations involved. The instantiation of the UI templates at run-time is discussed in Chapter 7.

### 6.2.1 User Interfaces for Incomplete Data

Once the parse tree is created, if one or more of the queried relations is crowdsourced, the UI generator is invoked. UI templates are essentially HTML forms that require instantiation at run-time before they can be used to interact with the crowd. The UI generator extracts the names of the queried relations and sets them as the title of the UI template. The template also carries instructions set by the application developer that are intended to assist workers to complete the form meaningfully.

If the query involves a single crowdsourced relation, each attribute in the schema of the relation becomes the label of a field in the UI template. At run-time, such a template is instantiated and updated. Upon posting it to AMT, the crowd completes the form and submits it. JavaScript code is used to ensure that crowdsourced values obtained from user input are of the correct type. If successful, such a form leads to a single update or insert into a crowdsourced relation.

If the query involves multiple crowdsourced relations, the schema of each of the queried relations is accessed. Attributes of each of the relations become labels of fields in the UI template. After instantiation at run-time, such a HTML form is capable of gathering input for multiple crowdsourced relations, potentially leading to multiple updates or inserts on the queried relations.

If any of the attributes in the queried relations is bound by a referential integrity constraint to a crowdsourced relation, the UI template is required to reflect this constraint. This case is special since the relation being referenced is crowdsourced, making it possible to crowdsource new tuples as a side-effect of query processing, as explained in Section 5.5.6. Accordingly, the UI generator uses the schema of the referenced crowdsourced relation and adds all the attributes of the relation as labels of fields in the UI template.

Finally, if the query includes a condition in the `WHERE` clause, the condition is included as a heading in the UI template to assist the worker in entering the data relevant to the query. `Submit` and `Cancel` buttons are provided to allow form submission.

### 6.2.2 User Interfaces for Subjective Comparisons

The UI generator follows a different procedure to create UI templates to support crowdsourced comparisons. This section covers UI template generation for the `CROWDEQUAL` and `CROWDORDER` functions.

**CROWDEQUAL** The UI template for the `CROWDEQUAL` function is a HTML form with predefined instructions requesting the workers to complete the entity resolution task. The title of the template contains the name of the queried relation. The `CROWDEQUAL` function has the following form.

```
attribute ~= '[a non-uniform value]'
```

The `attribute` is extracted from the `CROWDEQUAL` function and included as a heading in the UI template. The `'non-uniform value'` is added to the UI template as well. Once crowdsourced, the `'non-uniform value'` is used for comparison purposes by the crowd. A radio button with `Yes` and `No` options is provided to indicate the choice of the worker. `Submit` and `Cancel` buttons are provided to allow form submission. The UI template is instantiated at run-time and then used for crowdsourcing.

**CROWDORDER** The UI template generated for a `CROWDORDER` function is a HTML form that contains generic instructions requesting the crowd to rank a set of distinct values. The name of the queried relation forms the title of the UI template. The `CROWDORDER` clause has the following form.

```
CROWDORDER(attribute, 'aspect')
```

The `attribute` is used as a heading in the UI template along with the `'aspect'` to indicate the nature of the subjective comparison to the crowd. `Submit` and `Cancel` buttons are provided to allow form submission at run-time. Further changes to the `CROWDORDER` UI template are made at run-time.

## 6.3 Relational Algebra

Parse trees are converted to logical query plans expressed using relational algebra to describe the sequence of steps involved in query execution. CrowdDB extends traditional relational algebra to include crowd operators that are capable of dealing with crowd columns and tables. The semantics of crowd operators are different from their traditional counterparts, primarily owing to their ability to cater to the special requirements of `CNULL` values in CrowdDB. A detailed explanation of the underlying algorithms of crowd operators is provided in Chapter 7.

Table 6.1 provides an overview of the crowd operators used in CrowdDB. Crowd operators are used in addition to the traditional relational algebra to express logical query plans in CrowdDB. In the table, $\varphi$ represents an optional condition in the query.

$$\varphi = a\ \theta\ b$$

where $\theta \in \{\leq, <, =, >, \geq \sim=\}$, $a$ is an attribute name, $b$ is an attribute name or a value. Of interest to note is the inclusion of `CROWDEQUAL` ($\sim=$), the special binary comparison function in CrowdDB.

| Operator | Type | Description |
|---|---|---|
| $\pi^{crowd}_{c_1,\ldots,c_n}(R)$ | Unary operator | Crowd Projection |
| $\sigma^{crowd}_{\varphi}(R)$ | Unary operator | Crowd Selection where $\varphi$ is a condition |
| $R \times^{crowd} S$ | Binary operator | Crowd Cross Join |
| $R \bowtie^{crowd} S$ | Binary operator | Crowd Natural Join |
| $R \bowtie^{crowd}_{\varphi} S$ | Binary operator | Crowd Theta Join where $\varphi$ is a condition |
| $\tau^{crowd}_{(c,aspect)}(R)$ | Unary operator | CrowdOrder |

Table 6.1: Crowd relational algebra

**Crowd Projection** The crowd projection operator is a unary operator represented as $\pi^{crowd}_{c_1,\ldots,c_n}(R)$. Crowd projections occur in logical query plans owing to attributes specified in the `select_list` or owing to attributes specified in the `condition`. It produces a relation where one or more of the attributes are crowdsourced. The projected columns that are crowd columns may contain `CNULL` instances. The crowd projection operator does *not* include `CNULL` values in the the projected *set*. During query execution, upon encountering a `CNULL` value, the UI generator is invoked to update the UI template instead.

**Crowd Selection** The crowd selection operator is a unary operator represented as $\sigma^{crowd}_{\varphi}(R)$. It produces a relation with a subset of the tuples in a crowd table or a regular table with crowd columns that satisfy a `condition`. The crowd selection operator is implemented such that at run-time if the `condition` evaluates to `CNULL` or if one or more of the selected attribute values is `CNULL`, the specific tuple is not included in the *selected* set. Instead, the run-time functionality of the UI generator is invoked.

**Crowd Cross Join** The crowd cross join is a binary operator represented as $R \times^{crowd} S$ where either or both $R$ and $S$ are crowd tables or regular tables with crowd columns. The crowd cross join combines each tuple of $R$ with all the tuples of $S$. During query execution, if a selected attribute value is found to be `CNULL` in either of the participating relations, the UI generator is invoked.

**Crowd Natural Join** The crowd natural join is a binary operator represented as $R \bowtie^{crowd} S$ where either or both $R$ and $S$ are crowd tables or regular tables with crowd columns. Tuples in the relations are combined by comparing attributes that have the same name in both participating relations. The operator returns all combinations of tuples from $R$ and $S$ that satisfy the join condition. At run-time, if the join condition evaluates to `CNULL` or if either of the tuples from the participating relations have one or more selected attribute values set to `CNULL`, the UI generator is invoked.

**Crowd Theta Join** The crowd theta join is a binary operator represented as $R \bowtie_{\varphi}^{crowd} S$ where either or both $R$ and $S$ are crowd tables or regular tables with crowd columns. Typically, the functionality of the operator is equivalent to that of a crowd cross join followed by a selection. During query execution, if a `CNULL` is encountered in the theta join condition or in any of the selected attribute values, the UI template for the query is updated.

**CrowdOrder** The `CROWDORDER` is a unary operator represented as $\tau_{(c,aspect)}^{crowd}(R)$. It is used in conjunction with `ORDER BY` clause and allows the crowd to subjectively compare a set of tuples based on an attribute `c` along a certain `aspect`. The operator interacts with the UI generator to instantiate the UI template at run-time. It is typically the final operator in the logical query plan of the query.

## 6.4 Logical Query Plans

Logical query plans are used to describe the steps needed to transform the queried data sources into the format required by the result set. In traditional databases, they are represented as expression trees with nodes containing relational algebra operators. In CrowdDB, since the data sources may be crowdsourced, logical query plans are expressed using both traditional and crowd relational algebra.

Logical query plans in CrowdDB are constructed in a manner similar to that in traditional databases. If one or more underlying relations are crowdsourced, the query plan is built using the appropriate crowd operators. Some examples of queries executed on the `movie` schema and the corresponding query plans are explained below.

**Query 1 - Query with a predicate on primary key attribute**
Consider the schema provided in Section 5.3.2. The following query is used to find the name of the director of the movie with title `'The Godfather'`. In this case, the `LIMIT` of 1 is inferred. The corresponding query plan is provided in Figure 6.3.

```
SELECT m.director_name
FROM movie m
WHERE m.title = 'The Godfather';
```

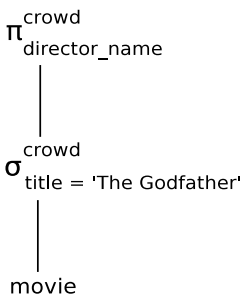


Figure 6.3: Logical query plan for Query 1

**Query 2 - Query with a predicate on non-key attribute**
Consider the schema provided in Section 5.3.2. The following query is used to find the titles and names of directors of ten movies released in `1994` that have a running time greater than `120` minutes. The query plan for the query is shown in Figure 6.4.

```
SELECT m.title, m.director_name
FROM movie m
WHERE m.year_of_release = 1994
AND m.running_time > 120 LIMIT 10;
```

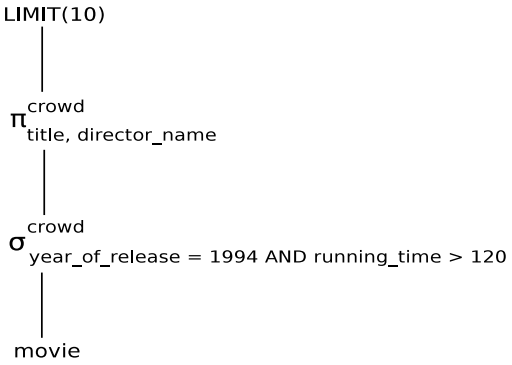


Figure 6.4: Logical query plan for Query 2

**Query 3 - Query with a predicate involving theta join condition**
Consider the schema provided in Section 5.5.6. The following query is used to find ten directors whose birthdate is before the year of release of the movie with title `'The Godfather'`. The query plan is shown in Figure 6.5.

```
SELECT d.name, d.place_of_birth, d.year_of_birth
FROM movie m, director d
WHERE m.title = 'The Godfather'
AND m.year_of_release > d.year_of_birth LIMIT 10;
```

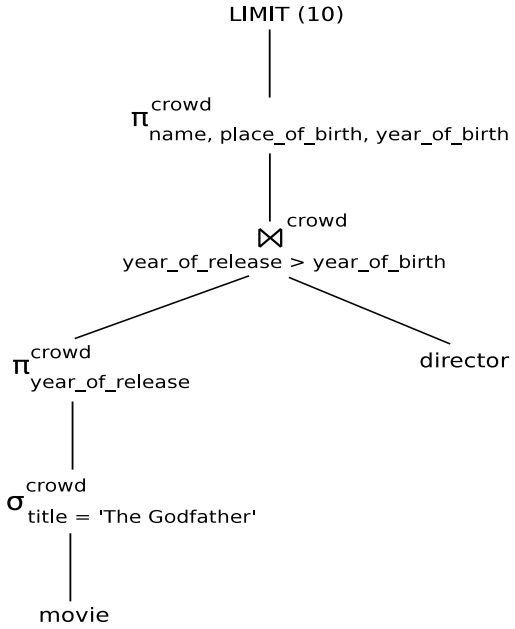


Figure 6.5: Logical query plan for Query 3

**Query 4 - Query with a predicate involving equi join condition**
Consider the schema provided in Section 5.5.6. The following query is used to find the details of ten `'Action'` movies and their directors. The corresponding query plan is expressed in Figure 6.6.

```
SELECT m.title, d.name, d.year_of_birth
FROM movie m JOIN director d
WHERE m.director_name = d.name
AND m.category = 'Action' LIMIT 10;
```

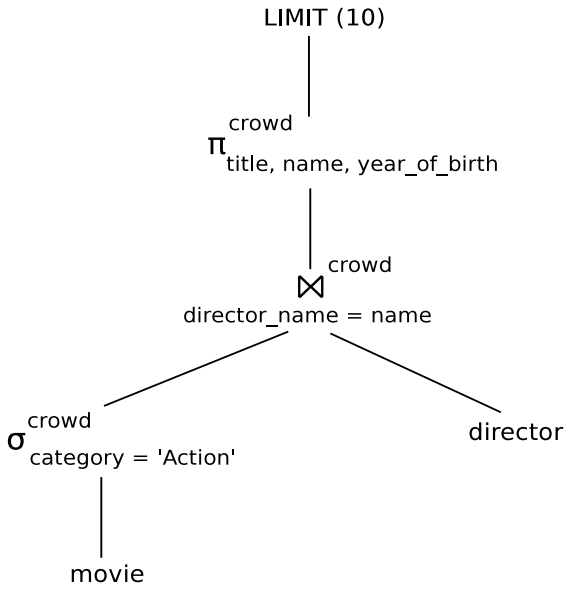


Figure 6.6: Logical query plan for Query 4

**Query 5 - Query with predicate involving `CROWDEQUAL` function**
Consider the schema provided in Section 5.3.2. The following query is used to find all existing tuples whose titles resolve to the same entity as `'LOTR'`. The query plan is shown in Figure 6.7.

```
SELECT m.title, m.year_of_release
FROM movie m
WHERE m.title ~= 'LOTR';
```

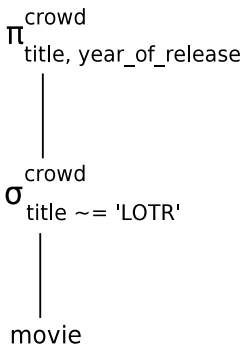


Figure 6.7: Logical query plan for Query 5

**Query 6 - Query with `CROWDORDER` function**
Consider the schema provided in Section 5.3.2. The following query aims to rank movies released after `1994` on the basis of their screenplays. The corresponding query plan is shown in Figure 6.8.

```
SELECT m.title, m.director_name
FROM movie m
WHERE m.year_of_release > 1990
ORDER BY CROWDORDER(m.title, "Best Screenplay");
```

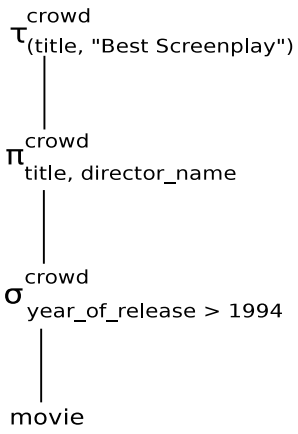


Figure 6.8: Logical query plan for Query 6

## 6.5 Physical Query Plan Generation

The last stage of query compilation converts the logical query plans into physical query plans. This is done by selecting the algorithm to be implemented by each operator in the logical query plan and deciding their sequence of execution. In case of CrowdDB, the logical query plans are expressed in terms of traditional and crowd operators. Hence, algorithms to implement the crowd operators are required such that crowd interaction using AMT can be interleaved with traditional query processing.

For a single logical query plan, it is possible to generate multiple physical query plans. Alternative physical query plans are conventionally generated by re-ordering operations, choosing different scan methods, choosing different sequences of join operations, etc. In order to compare multiple physical query plans, a *cost* is associated with each physical plan operator. A cost-based optimizer is then used to quantitatively find the best physical query plan. The cost-based optimizer in CrowdDB is also required to factor in costs associated with employing crowd operators. Configurable parameters involved in deciding the cost of a crowd operator include maximum permissible response time, maximum cost ($) that the application developer is willing to spend on a task and minimum expected quality of results. The cost of employing crowd operators is also impacted by observable parameters based on the state of the AMT workplace, estimated size of the available workforce, time of day, etc. Accordingly, adaptive optimization of query plans based on run-time parameters is an important area of future work.

CrowdDB uses three new physical plan operators for crowd operations – AMTProbe, AMTJoin and AMTFunction. These operators are responsible for using the AMT API to collect crowd input during query processing. Tuples in crowdsourced relations are accessed with the AMTProbe operator. If the query includes a join to be executed on crowdsourced relations, the AMTJoin operator is used. The AMTFunction operator is used in the query plan when either of the crowd comparison functions, `CROWDEQUAL` or `CROWDORDER`, is used. The implementation of the AMTProbe, AMTJoin and AMTFunction operators along with the processing model is explained in Chapter 7.

# Chapter 7

# CrowdDB Query Execution

CrowdDB query execution interleaves traditional query processing with crowdsourcing. Accordingly, new physical plan operators that support crowdsourcing and their algorithms are explained in Section 7.1. The processing model that CrowdDB uses is based on the iterator model in combination with a batch model for crowdsourcing as explained in Section 7.2.

## 7.1 Physical Query Plans

CrowdDB introduces new physical plan operators to allow interaction with AMT during query processing. The new operators define the algorithms used by the extended relational algebra described in Section 6.3. If queried relations are not crowdsourced, traditional physical plan operators are used. A brief overview of the traditional physical plan operators is provided in Section 7.1.1. A detailed description of the new CrowdDB physical plan operators is provided in Section 7.1.2.

### 7.1.1 Traditional Operators

The physical plan operators of traditional relational databases [32] remain the same in CrowdDB and are used when no crowdsourcing is involved. The underlying algorithms for the traditional operators may be sort-based, hash-based or index-based. Depending on the size of the main memory and the relations, the algorithms may also be one-pass, two-pass or multi-pass algorithms.

**Unary Operations** Selection and projection operations are categorized as unary operations. Blocks holding tuples are read one at a time into the input buffer, the selection or projection operation is performed on each tuple and the result is written to the output buffer. This classifies as a one-pass algorithm. If the selection operator has a condition of the form $a\ \theta\ b$ where $a$ is an attribute that has an index and $b$ is a constant value associated with it, then a two-pass index-based selection algorithm can be employed.

**Binary Operations** Joins are classified as binary operations. Consider two relations $R$ and $S$ as the larger and smaller relation respectively. The cross join is computed by reading $S$ into $(M-1)$ blocks of main memory and reading each block of $R$ into the $M^{th}$ block, one at a time. Each tuple of $R$ is then concatenated with each tuple from $S$. The natural join is performed by reading all of the blocks of $S$ into $(M-1)$ blocks of main memory and building a search structure (a hash table or a balanced tree) on the join attribute. Each block of $R$ is read into the $M^{th}$ block. For every tuple in $R$, the search structure is probed to find tuples in $S$ that match it. Matching tuples are concatenated and written to the output buffer. Other types of joins can also be implemented as cross joins followed by selections.

Several alternatives exist for join implementations. The *nested loop join* is implemented either at the level of individual tuples or at the level of blocks of the participating relations. The tuple-based nested loop join fits well with the iterator model of implementation. In case of the *sort join*

implementation each relation is sorted independently on the join attribute and an optimized one-pass join of the sorted relations is then performed. The *hash join* is employed when the join attribute is used to hash tuples of the participating relations into buckets. A one-pass join algorithm is then used on all pairs of tuples in each bucket. The *index join* is used when an index exists on the join attribute. Executing the join involves looking up the index and finding all tuples that have matching join attribute values.

### 7.1.2 Extended Operators

The new operators in CrowdDB allow crowdsourcing to be performed during query processing. Three new operators are introduced in CrowdDB – AMTProbe, AMTJoin and AMTFunction. Since the new operators enable interaction with the crowd, they are required to utilize the UI templates generated at compile-time. At run-time, based on the data being queried, the new operators instantiate the UI templates to create HTML forms, which are then posted to AMT. The new operators are also responsible for collecting and processing submitted data from AMT. All operations involving AMT including creating new HITs, accessing submitted results and approving or rejecting assignments, are performed using the AMT API.

#### AMTProbe

AMTProbe is the primary physical plan operator that allows missing data to be completed by the crowd. Incomplete data in CrowdDB is of two types. The first type is when the tuple exists in the relation but one or more of its attributes are missing. In this case, the crowd is required to complete the missing information for the *specific* tuple. The second type of missing data is when the entire tuple is missing from the crowd table and needs to be added by the crowd.

- **Incomplete Tuples**
  During query processing, if a crowd selection or a crowd projection operator encounters `CNULL` in a specific tuple, the tuple is *not* added to the result set. Instead, AMTProbe invokes the UI generator indicating the need for crowdsourcing. The UI generator instantiates the UI template into a HTML form that reflects data in the specific tuple.

  The UI template generated at compile-time includes all the attributes names of the queried relation. Upon instantiation, all complete attribute values, i.e., non-`CNULL` attribute values, are added to the HTML form against the appropriate attribute names. If the attribute value is found to be `CNULL`, the UI generator needs to provide an input field to allow the crowd to complete the attribute value.

  The *type* of the input field is decided as follows:

  - If the attribute is not bound by any constraints, a text input field is included in the HTML form against the particular attribute name.
  - If the attribute holds a `CHECK` constraint restricting its domain, a drop-down menu is included with all valid values in the domain.
  - Lastly, if the attribute is bound by a foreign key constraint and its value is found to be `CNULL`, the domain of permissible key references is added to the HTML form. The referenced attribute is accessed and all permissible key references are included in a drop-down menu. The inclusion of drop-down menus ensures no constraint violation upon form submission.

  Figure 7.1 shows two examples of automatically generated UIs for incomplete tuples. The UIs are generated based on the schema in Section 5.3.2, in response to the following query.


Figure 7.1: Automatically generated UIs for completion of incomplete tuples

```
SELECT * FROM movie
WHERE running_time < 200 LIMIT 10;
```

Once the form is ready, AMTProbe posts it to AMT using the `CreateHIT` operation. Relevant HIT parameters such as reward and number of assignments are set by the application developer. AMTProbe uses the parameters during HIT creation on AMT. Once posted, AMTProbe repeatedly polls AMT using the `GetAssignmentsForHIT` operation to check for submitted results. As soon as results are submitted, AMTProbe performs an *update* on the queried relation with the newly crowdsourced data. The update, hence, replaces any `CNULL` values in the specific tuple with crowdsourced input. Upon a successful update, AMTProbe approves the assignment and, in turn, transfers the reward amount to the worker.

- **New Tuples**
  CrowdDB makes it possible to add new tuples to crowd tables during query processing. For this purpose, the UI template generated at compile-time is instantiated such that the crowd can enter *all* attribute values pertaining to the new tuple. The *type* of input field corresponding to each attribute is decided using the same rules as that of incomplete tuples explained above. If the query involves multiple crowdsourced relations that share no referential integrity constraint, AMTProbe repeats the same procedure for each relation in the query.

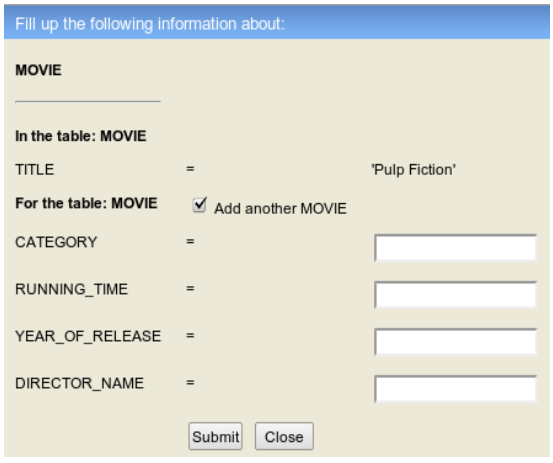


Figure 7.2: Automatically generated UI for crowdsourcing new tuples

Figure 7.2 shows an example of the UI generated to crowdsource a new `movie` entity with `title` as `'Pulp Fiction'`. Based on the schema provided in Section 5.3.2, the following query is capable of generating the shown UI.

```
SELECT * FROM movie
WHERE title = 'Pulp Fiction';
```

AMTProbe posts the form to AMT using the `CreateHIT` operation. It polls the AMT platform using the AMT API to find submitted results. Once results are submitted, AMTProbe collects all attribute values and executes an *insert* on the crowd table. The insert leads to a newly crowdsourced tuple to be added to the queried relation. If the insert is successfully executed, AMTProbe approves the assignment and transfers the reward amount to the worker. Since it is possible to crowdsource tuples for multiple crowdsourced relations in the same HTML form, it is also possible that multiple inserts are performed upon form submission.

### AMTJoin

AMTJoin is the physical plan operator that crowdsources completion of functional joins. AMTJoin is used when a referential integrity constraint exists between the crowdsourced relations participating in the join. In case of joins in the *absence* of referential integrity constraints, AMTProbe is used on multiple relations as explained in Section 7.1.2.

AMTJoin is implemented as an index nested loop join if an index is available on the join attribute, or else as a nested loop join. Conceptually, AMTJoin performs an AMTProbe and then completes the join. Consider a query with a two-way join. Given that the participating relations may be regular or crowdsourced, Table 7.1 summarizes the activity of AMTProbe and AMTJoin under these conditions.

| | | **Referencing relation** | | |
|---|---|---|---|---|
| | | **Regular table** | **Regular table with Crowd columns** | **Crowd table** |
| **Referenced relation** | **Regular table** | Traditional join, AMTProbe and AMTJoin not applicable | AMTProbe for incomplete tuples in referencing relation, AMTJoin applicable | AMTProbe for incomplete tuples and new tuples in referencing relation, AMTJoin applicable |
| | **Regular table with Crowd columns** | AMTProbe for incomplete tuples in referenced relation, AMTJoin not applicable | AMTProbe for incomplete tuples in both relations, AMTJoin applicable | AMTProbe for incomplete tuples and new tuples in referencing relation, AMTProbe for incomplete tuples in referenced relation, AMTJoin applicable |
| | **Crowd table** | AMTProbe for incomplete tuples and new tuples in referenced relation, AMTJoin not applicable | AMTProbe for incomplete tuples in referencing relation, AMTProbe for incomplete and new tuples in referenced relation, AMTJoin applicable | AMTProbe for incomplete and new tuples in both relations, AMTJoin applicable |

Table 7.1: AMTJoin applicability

AMTJoin is applicable when the referencing table is crowdsourced. Upon processing of the join, if a tuple with an *incomplete* foreign key attribute value is accessed, AMTJoin performs the task of instantiating the UI template. The referenced table is accessed to find the set of keys that *can* be referenced. These are included in a drop-down menu and added to the HTML form against the attribute name bound by the foreign key constraint. This prevents integrity constraint violations from occurring upon form submission. It is also possible that during join processing, a tuple in the referencing table is accessed that has a complete foreign key value but one or more of the other attribute values are set

to CNULL. In such a case, the functionality of AMTProbe would suffice for completion of the missing information. Finally, if the referencing table is a crowd table, it is possible to add new tuples to it as a side-effect of processing the join.

As summarized in Table 7.1, when AMTJoin is applicable, the referenced table may be a regular table, a regular table with crowd columns or a crowd table. During join processing, if a tuple in the referenced table is accessed and is found to be incomplete, AMTProbe invokes the UI generator to instantiate the UI template and crowdsource completion of the tuple. Also, as a side-effect of join processing, adding new tuples to a referenced crowd table is possible.


Figure 7.3: Automatically generated UI for join completion with existing tuples

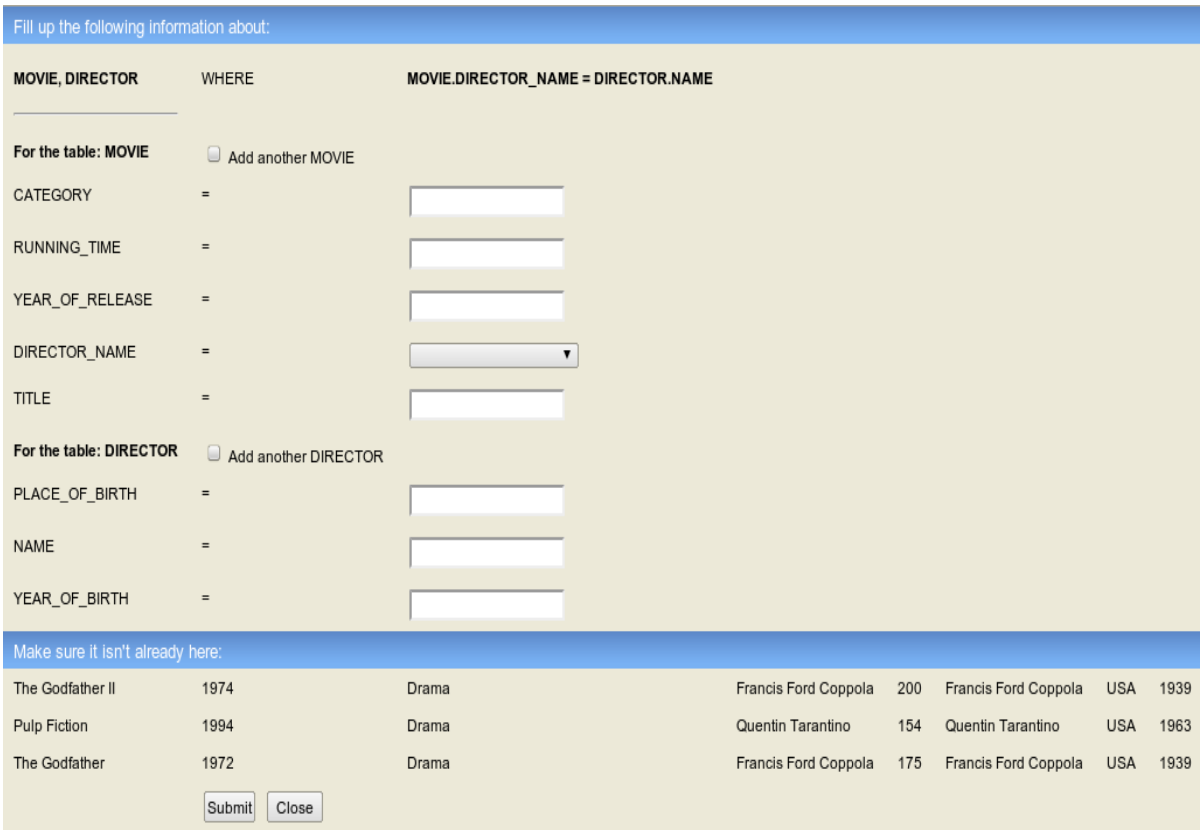


Figure 7.4: Automatically generated UI for join completion with new tuples

Figure 7.3 shows the UI generated for the completion of a foreign key reference of an existing tuple with `title` set to `'Pulp Fiction'`. A drop-down helps the worker select an existing `director`. Since `director` is a crowd table, the worker can also add a new director and hence, complete the missing reference. Figure 7.4 shows the UI generated when insufficient tuples exist in the result set. In such a case, tuples can be added by the crowd to either or both of the participating crowd tables.

Once the UI template is instantiated, AMTJoin invokes the `CreateHIT` operation on AMT to post the HTML form and crowdsource the join. Workers view the HITs on AMT, accept them and submit results. Once submitted, AMTJoin performs an *update* or an *insert* on the referencing table. In case the worker chooses to add a new key to the referenced table, it is essential that the *insert* on the referenced table is performed before the *update* on the referencing table. This prevents integrity constraint violations. Upon successful completion, AMTJoin approves the assignment and transfers the reward to the worker.

### AMTFunction

AMTFunction is a generalization of the operators corresponding to the crowd comparison functions in CrowdDB.

- **CROWDEQUAL**
  As described in Section 5.5.3, the `CROWDEQUAL` function used in the following form.

  ```
  attribute ~= '[a non-uniform value]'
  ```

  AMTFunction for `CROWDEQUAL` is used to compare each attribute value to the non-uniform value provided in the query. For each comparison, the `CROWDEQUAL` cache is first checked. AMTFunction maintains a cache for `CROWDEQUAL` results in the form of auxiliary tables.

  The auxiliary table is created specific to the attribute in the predicate. When the `CROWDEQUAL` function is used for the *first* time on a particular attribute, CrowdDB automatically creates the auxiliary table. The auxiliary table has three fields that hold the attribute value to be compared, the non-uniform value, and a boolean value that indicates if the two values resolve to the same entity. The primary key of the auxiliary table is a combination of the first two fields, i.e., the fields being compared. All `CROWDEQUAL` results gathered for the particular attribute are, henceforth, cached in this auxiliary table. Negative results are also cached to avoid repeated crowdsourcing.

  If AMTFunction finds the combination of values in the cache, no crowdsourcing is required. On the other hand, if the combination of values is not found in the cache, crowdsourcing is needed to check if the values resolve to the same entity. In this case, AMTFunction instantiates the UI template. The UI template already contains the non-uniform value along with a radio button with `Yes` and `No` options, added at compile-time. Instantiation, in this case, merely involves adding the specific attribute value to be compared to the HTML form.

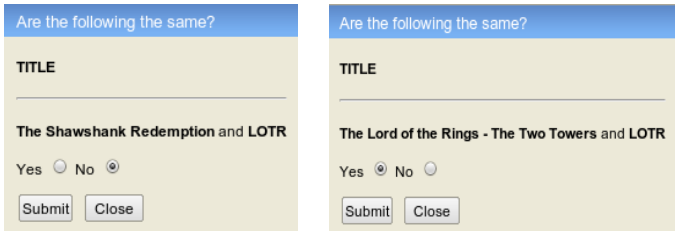


Figure 7.5: Automatically generated UIs for the `CROWDEQUAL` function

Figure 7.5 shows two examples of UIs generated for the following query based on the schema described in Section 5.3.2.

```
SELECT * FROM movie
WHERE title ~= 'LOTR';
```

AMTFunction posts the form to AMT using the `CreateHIT` operation. Using polling with `GetAssignmentsForHIT`, AMTFunction checks for the presence of submitted results. Once results are available, AMTFunction updates the auxiliary table. If a worker submits `Yes`, a new entry is added to the auxiliary table with the compared values in the first two columns and `TRUE` in the third column. If the submitted value is `No`, the third column is set to `FALSE`. Upon successfully updating the auxiliary table, the assignment is approved and the worker receives the reward.

- **CROWDORDER**
  The `CROWDORDER` function is employed when subjective rankings are required.

```
CROWDORDER(attribute, 'aspect')
```

AMTFunction for `CROWDORDER` accesses the queried table to retrieve all values of the attribute to be ranked. The UI template for a `CROWDORDER` function already carries the `'aspect'` information. AMTFunction instantiates the UI template by adding each attribute value to the form. Corresponding to each attribute value, a drop-down menu is provided on the HTML form with the permissible set of ranks that the value can be assigned.

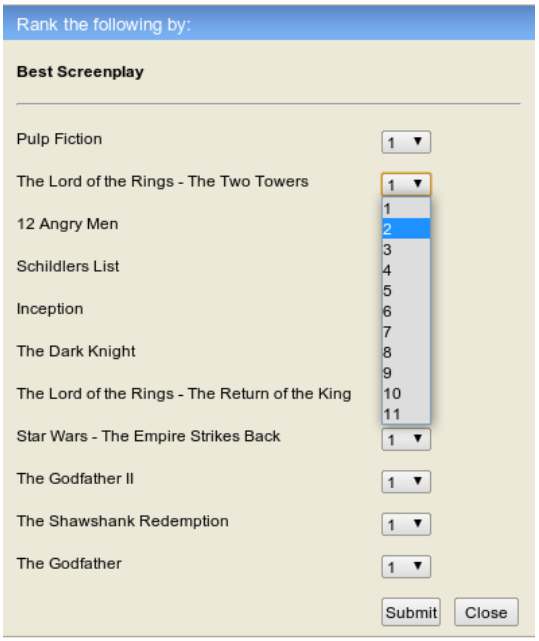


Figure 7.6: Automatically generated UI for the `CROWDORDER` function

Figure 7.6 shows an example of the UI generated for the `CROWDORDER` function. The schema for the `movie` relation has been defined in Section 5.3.2. The UI is generated when the following query is submitted to CrowdDB.

```
SELECT * FROM movie
ORDER BY CROWDORDER(title, 'Best screenplay');
```

Each generated UI is posted as a separate HIT on AMT. AMTFunction posts the form to AMT using the `CreateHIT` operation. Once the workers submit their results, AMTFunction uses the submitted ranks to reorder the selected tuples based on their crowdsourced rank. Multiple tuples are permitted to have the same rank. In such a case, all tuples within a single rank are randomly ordered. Upon completion, AMTFunction approves the assignment and pays the specified reward to the worker. Of interest to note is that it is possible for independent queries to order the same attribute along different aspects. Hence, caching `CROWDORDER` results requires storing the aspect as an additional dimension. To study different caching strategies for `CROWDORDER` results is an important avenue for future work in CrowdDB. Once cached, using the transitive property of partial orders is a possible optimization to minimize the required crowdsourcing.

**StopAfter**

In CrowdDB, the StopAfter operator [7] is used to tackle the open-world issue. Its main functionality is to limit the amount of crowdsourcing needed. Hence, in order to restrict the cardinality of the result set, it uses the argument of the `LIMIT` clause specified in the query.

StopAfter is responsible for detecting when enough tuples have been crowdsourced based on the required cardinality of the result set. Upon detecting such a situation, it expires all outstanding HITs on AMT by using the `ForceExpireHIT` operation of AMT. In CrowdDB, it is also possible to define the behavior of the StopAfter operator based on response time and cost ($) constraints. The implementation of such extensions of the StopAfter operator are part of future work.

## 7.2 Processing Model

Given the nature of the new physical plan operators in CrowdDB, it is required to extend the iterator-based processing model of traditional databases. The extended processing model permits crowdsourcing to be performed in *batches*.

Several traditional databases use the iterator model to process queries. Iterators support a pipelining approach to query execution, where tuples are passed between operators one at a time and hence, many operations are active at once. An iterator is characterized by the `open()` function which starts the process of getting tuples, the `next()` function which returns the next accessed tuple and the `close()` function which ends the iteration after all the tuples have been accessed.

CrowdDB uses the iterator model in combination with a batch processing model of crowdsourcing. Batch processing is implemented using multiple queues that are populated by the CrowdDB operators. Each queue holds a batch of crowdsourcing tasks of the same *priority*. The priority of a batch is defined by the impact that the set of crowdsourcing tasks will have on the expected query result set, upon completion. When query execution begins, tuples in the queried relation are accessed one at a time using the iterator model. The rules used to process each tuple depend on whether the submitted query has a `WHERE` predicate.

In case the query has no `WHERE` predicate, the following set of rules apply.

- If a tuple is complete, it is added to the result set.
- If one or more attribute values of a tuple are `CNULL`, AMTProbe creates the HIT and inserts it into the *incomplete-tuple queue*.
- If the query contains a `CROWDEQUAL` function, AMTFunction creates the entity resolution HIT corresponding to the tuple, and inserts it into the *entity-resolution queue*.
- Finally, if the query contains a `CROWDORDER` function, AMTFunction creates a HIT to order multiple tuples and inserts it into the *subjective-ranking queue*.

The following set of rules apply when the submitted query has a `WHERE` predicate. These rules are richer owing to the four-valued logic in CrowdDB which permits a predicate to be evaluated to `CNULL`.

- If a tuple satisfies the `WHERE` predicate and is complete, it is added to the result set.
- If a tuple satisfies the `WHERE` predicate but one or more of its other attribute values are `CNULL`, AMTProbe creates the HIT and inserts it into the *incomplete-tuple queue*.
- If the `WHERE` predicate evaluates to `CNULL` for a tuple, i.e., the attribute value required for predicate evaluation is `CNULL`, then AMTProbe or AMTJoin create the HIT and insert it into the *unknown-predicate queue*. In case the submitted query has no `WHERE` predicate, the *unknown-predicate queue* remains empty.
- If the query contains a `CROWDEQUAL` function, AMTFunction creates an entity resolution HIT corresponding to the specific tuple. The HIT is inserted into the *entity-resolution queue*.
- Finally, if the query contains a `CROWDORDER` function, AMTFunction creates a HIT to order multiple tuples and inserts it into the *subjective-ranking queue*.

Once all the tuples have been accessed by the iterators, the *first phase* of query execution in CrowdDB is complete. The result set contains all complete tuples that satisfy the query. If the size of the result set is greater than or equal to that expected by the query, the StopAfter operator detects that no crowdsourcing needs to be done. The result set is returned to the application and query execution is completed.

**Incomplete-tuple Queue** If the size of the result set is smaller than that required by the query, crowdsourcing needs to be performed. For this purpose, first, the *incomplete-tuple queue* is processed. The queue has top priority since it holds crowdsourcing tasks that upon completion will definitely impact the cardinality of the result set. Processing the queue involves posting the tasks to AMT, collecting the results and updating the queried relations by the relevant operators, as explained in Section 7.1.2.

**Unknown-predicate Queue** When the tasks from the *incomplete-tuple queue* have been processed, the *second phase* of query execution begins. In this phase, the query is executed on the updated relations. The size of the result set is now compared to that expected by the query. If satisfied, the StopAfter operator detects that no more crowdsourcing needs to be performed and any pending crowdsourcing tasks are expired. If the result set does not contain enough tuples, the *unknown-predicate queue* is processed. This queue holds the set of tasks to crowdsource that, upon completion, *may* have an impact on the result set size. The relevant operators posts tasks to AMT and collects the results. The queried relations are then updated.

**New-tuple Queue** Following the processing of the *unknown-predicate queue*, the *third phase* of query execution begins. The query is executed on the updated relations and the size of the result set is checked. If the table being queried is a regular table with crowd columns and/or if sufficient tuples exist in the result set, CrowdDB returns the result set and completes query execution.

In the event of insufficient tuples in the result set and the queried table being a crowd table, further crowdsourcing is possible. This is done by crowdsourcing the *remaining* required number of tuples as new tuples. The appropriate number of crowdsourcing tasks for adding new tuples are created using AMTProbe and inserted into the *new-tuple queue*. The operator collects results and performs inserts on the queried relations. Once enough new tuples have been crowdsourced, the query is executed again and the result set is returned to the application. In this case, the size of the result set will satisfy the requirement of the StopAfter operator with certainty.

**Entity-resolution Queue** If the submitted query involves the `CROWDEQUAL` function, all unknown values in the attribute of the predicate are first crowdsourced and completed as above. The *entity-resolution queue* is then processed by using AMTFunction to post and retrieve HIT results. Auxiliary tables are updated as explained in Section 7.1.2. The query is then executed on the updated tables and results are returned to the application.

**Subjective-ranking Queue** Finally, once all the accessed tuples are complete, if the submitted query involves the `CROWDORDER` function, the *subjective-ranking queue* is processed. AMTFunction posts the tasks to AMT and retrieves results. It reorders the tuples in the result set according to the submitted results and returns the result set to the application.

**Advantage and Disadvantage** Using batches to segregate the tasks and prioritize them, allows CrowdDB to minimize the amount of crowdsourcing needed and hence, save on cost ($). Furthermore, the use of batches permits multiple tasks to be posted to AMT concurrently, yielding savings in terms of time. Despite this, since each batch is processed *entirely* before the next phase of query execution begins, it is possible that CrowdDB crowdsources marginally more tasks than required.

# Chapter 8

# Experiments and Results

Preliminary experiments conducted on AMT and the results are presented in this chapter. The experiments have also been presented in [13]. Section 8.1 presents the results of a set of micro benchmarks evaluated on AMT. Section 8.2 details the execution of traditionally difficult queries. Observations based on the experiments are summarized in Section 8.3.

Experiments were conducted by posting over 25,000 HITs to AMT while varying certain factors such as reward ($) and size of HIT Group. The measured metrics include response time and quality of answers received. An important consideration for experiments conducted on AMT is the current state of the marketplace since the available worker community has a significant impact on the metrics. AMT, as an Internet marketplace, is still expected to evolve and analyzing its exact behavior is part of future work.

## 8.1 Micro Benchmarks

The set of experiments conducted as micro benchmarks revealed the behavior of AMT in terms of responsiveness and quality of results. The size of the HIT group and the reward for each task were the factors varied. Workers participating on AMT are not evenly distributed across time zones [33]. Since time of day impacts the availability of workers, all experiments were conducted between 0600 and 1500 hours to minimize the variance [19]. Each experiment was repeated four times and the average values measured. HITs were posted in groups of 100, with 5 assignments per HIT. The default reward was $0.01. The following schema was used for the micro benchmark experiments.

```
CREATE TABLE businesses (
  name VARCHAR PRIMARY KEY,
  phone_number CROWD VARCHAR(32),
  address CROWD VARCHAR(256)
);
```

After populating the table with the names of 3607 businesses across 40 cities in the USA, the following query was used to complete missing phone numbers and addresses.

```
SELECT phone_number, address FROM businesses;
```

### 8.1.1 Responsiveness based on Size of HIT Group

AMT groups the HITs based on their HIT Type, as mentioned in Section 2.2.1. Larger HIT Groups typically receive more visibility on AMT. The first experiment measured the responsiveness of workers on AMT, based on the size of the HIT Group. In this case, each HIT had a single assignment and required the worker to complete the details of one business. All HITs posted were of the same HIT Type, since they required completion of phone numbers and addresses of businesses. Hence, by

varying the number of HITs posted, the size of the HIT Group and hence, visibility on AMT could be controlled.

Figure 8.1 shows the response time in minutes for various HIT Group sizes. As the size of the HIT Group grows, less time is needed to receive the first response from the crowd. The same also holds for the first 10 and 25 HITs. The result of this experiment reiterates the observation that as the size of the HIT Group increases, the visibility on AMT is greater and hence, the response time is shorter.

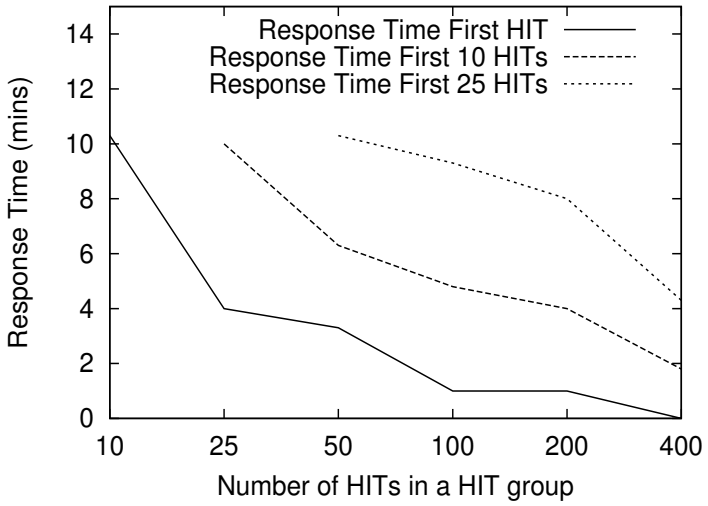


Figure 8.1: Response time based on size of HIT Group

Figure 8.2 shows the percentage and the absolute number of HITs completed in the first 30 minutes. The absolute numbers show that more HITs were completed in the larger HIT Groups than in the smaller ones. In terms of percentages, it can be seen that a group size of approximately 100 HITs is optimal. For such a group size, almost all the HITs, i.e., 97.3% of them, were completed within the first 30 minutes. With group sizes larger than 100, despite the higher visibility on AMT, several HITs remain incomplete owing to the adverse impact of a relatively small worker community. Such a result further motivates the need for the CrowdDB optimizer to consider the state of the AMT marketplace to decide the number of HITs to post and their granularity in terms of the number of assignments.

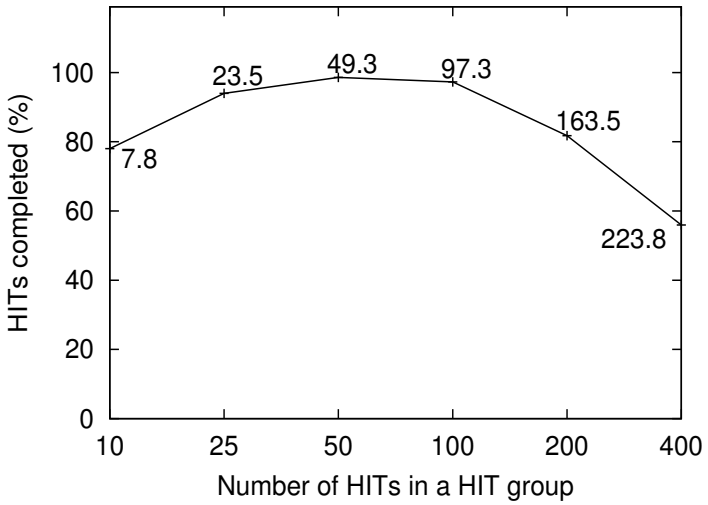


Figure 8.2: Percentage of HITs completed based on size of HIT group

### 8.1.2 Responsiveness based on Varying Reward

The second set of experiments was performed by varying the reward associated with each HIT. In this case, HITs were posted in groups of 100 with 5 assignments per HIT to study the responsiveness of workers on AMT.

Figure 8.3 shows the percentage of HITs that were completed as a function of time elapsed since the HIT was posted. A HIT is deemed completed when *all* its assignments receive responses. The reward for the HITs was varied between \$0.01 and \$0.04. As expected, since \$0.04 was the highest reward offered, the percentage of HITs completed in that reward category was the highest while that of \$0.01 was the lowest. Also, it can be seen that there is only a marginal difference between the HITs completed in the \$0.02 and \$0.03 reward categories. As is evident, the higher the reward, the quicker the response of the workers on AMT.

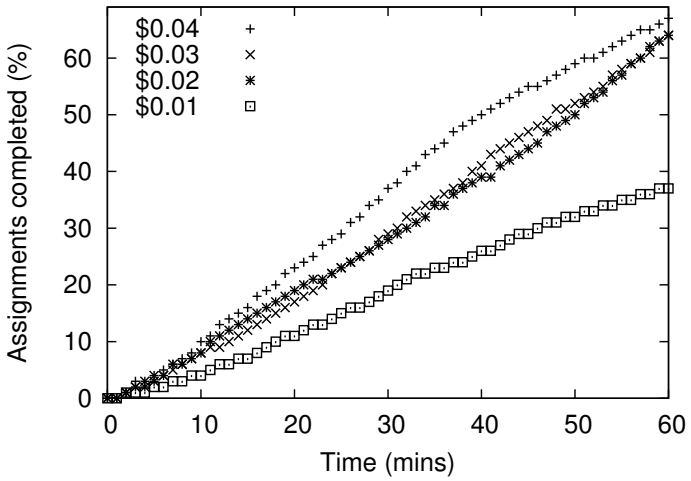


Figure 8.3: Percentage of HITs completed with varying rewards

Figure 8.4 shows the percentage of HITs that received a response for *at least* one assignment as a function of time. The results show that there is little or no difference in the time taken to receive at least one response for a HIT, whether the reward is \$0.02, \$0.03 or \$0.04. Also observable is the fact that within 60 minutes, nearly all the HITs in each of these reward categories (except the \$0.01 reward category) received a response. Comparing it with Figure 8.3, it can be seen that even with a reward of \$0.04, only 65% of the HITs received *all* responses in 60 minutes.

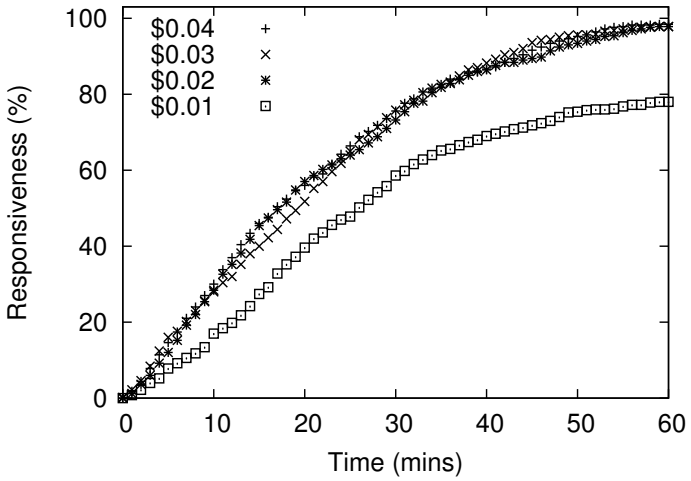


Figure 8.4: Percentage of HITs that received at least one assignment response with varying rewards

### 8.1.3 Responsiveness based on Interference between HIT Groups

Since AMT is a marketplace, the responsiveness to a certain HIT Group is impacted by the presence of other HIT Groups. The third experiment involved studying the interference between similar HIT Groups, when posted concurrently to AMT. In order to do so, initially a single HIT Group with 100 HITs was posted to AMT. After 30 minutes, another HIT Group with 100 HITs was posted to AMT. The second HIT Group contained similar tasks to those in the first group. The reward for HITs in both groups was set to be $0.01. Figure 8.5 shows the percentage of HITs completed in the first HIT Group as a function of time. From the results, it is clear that more HITs were completed during the first 30 minutes than in the next 30 minutes. Accordingly, it can be inferred that the presence of a similar concurrent HIT Group during the latter half of the experiment interfered with the completion of HITs in the first HIT group.

In a similar experiment to study interference, the reward associated with the two groups was set to different values. The first group consisted of HITs with a reward of $0.02 while the second group had HITs with a reward of $0.01. In this case, the percentage of HITs completed in the first group was *not* impacted by the second group, indicating that reward plays an important role in the selection of a HIT by a worker.

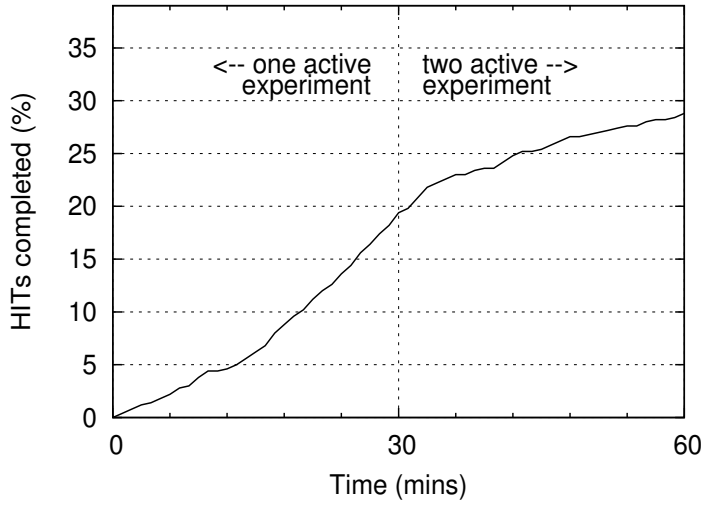


Figure 8.5: Responsiveness based on interference between HIT groups

### 8.1.4 Quality of Results

The purpose of the next experiment was to study the quality of the results obtained from AMT. Figure 8.6 shows the distribution of the number of HITs completed by participating workers. The curve follows a Zipf distribution which indicates that a small set of all AMT workers tend to gain a specialty in solving HITs of a particular kind. They may also have preferences based on the requester. This is in agreement with previous studies such as [21], which state that requesters tend to build communities of workers over the course of time based on the type of HITs that they post.

Figure 8.6 also shows the curve indicating the number of errors made by each worker. Each HIT in this experiment was replicated into 5 assignments. A majority vote was performed on the 5 assignments of each HIT to find the *correct* answer. If the answer provided by a worker differed from the majority vote, it was categorized as an error. Since the HITs involved finding phone numbers, the responses obtained from the workers were normalized such that hyphens, brackets, etc. were ignored. No particular trend could be found in the number of errors made by workers. As expected though, workers that complete more HITs tend make more errors, in absolute numbers. It is encouraging to note that no workers seemed to spam merely for the sake of the reward. This may be viewed as an

indicator that AMT is a relatively reliable crowdsourcing platform to build a system upon, as long as the rewards are not too high. It is possible that the probability of workers spamming increases as the reward associated with HITs increases.

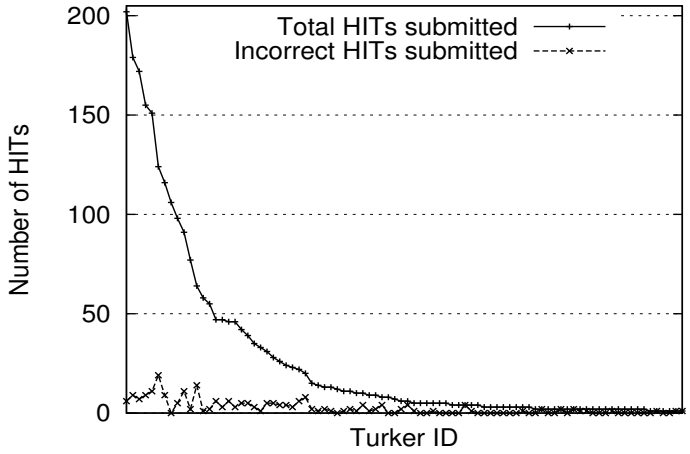


Figure 8.6: Distribution of quality of results by worker

## 8.2 Difficult Queries

The main motivation to build a system such as CrowdDB is its ability to answer traditionally difficult queries. This section presents preliminary experiments that study the completion of missing information, entity resolution and subjective rankings, using crowdsourcing.

### 8.2.1 Joins

In order to evaluate alternate query plans for joins that use crowdsourcing, the following schemas were used.

```
CREATE TABLE department (
  university VARCHAR(255),
  name VARCHAR(255),
  url  CROWD VARCHAR(255),
  phone  VARCHAR(32),
  PRIMARY KEY (university, name)
);

CREATE CROWD TABLE professor (
  name VARCHAR(255) PRIMARY KEY,
  email VARCHAR(255) UNIQUE,
  university VARCHAR(255),
  department VARCHAR(255),
  FOREIGN KEY (university, department)
   REFERENCES Department(university, name)
);
```

The relations were populated with 8 departments and 25 professors respectively. The following query was used to find information specific to a professor and the associated department.

```
SELECT p.name, p.email, d.name, d.phone
FROM   professor p, department d
WHERE  p.department = d.name AND
      p.university = d.university AND
      p.name = '[name of a professor]'
```

The first plan crowdsourced the `professor` information, including the `department` as an initial step. In the next step, the remaining details including `phone` attribute of the `department`, were crowdsourced. The second plan that was evaluated was *denormalized*, i.e., it crowdsourced information about the `professor` and the associated `department` in a single step. In order to uniformly compare the plans, each HIT crowdsourced information about a single `professor` entity and was replicated into 3 assignments. The reward was set to $0.01.

In terms of response time, the first plan took 206 minutes to complete while the second plan took 173 minutes. The first plan cost $0.99 to execute, while the second plan cost $0.75. Finally, in terms of quality, only a single incorrect `phone` attribute value was encountered while using the first plan. In contrast, the second *denormalized* plan produced wrong results for all `phone` attribute values, since workers entered the professors' phone numbers instead of phone numbers of the departments. As can be inferred from this result, good UI design is essential in order to gather meaningful results from the crowd. Since the query plan plays a crucial role in determining the response time, cost and quality of results obtained, the CrowdDB optimizer needs to cater to such varied requirements.

### 8.2.2 Entity Resolution

For the purpose of resolving entities using the crowd, the following schema was employed.

```
CREATE TABLE company(
    name VARCHAR(255),
    address VARCHAR(255)
);
```

Once the relation was populated with Fortune 100 companies from Wikipedia, the following query was used to resolve entities.

```
SELECT name FROM company
WHERE name ~='[a non-uniform name of the company]'
```

The values of the non-uniform names used are presented in Table 8.1. Each value in the set of non-uniform names was used to create an instance of the query. A HIT, in this case, involved comparing each non-uniform value to 10 company names. Each HIT was replicated thrice and a majority vote was performed on the answers submitted by workers. All four queries returned the correct answer based on the majority vote, as presented in Table 8.1. The total of 40 HITs, corresponding to the 4 non-uniform values, were completed in 39 minutes.

| Non-uniform name | Query result | Votes |
|---|---|---|
| Bayerische Motoren Werke | BMW | 3 |
| International Business Machines | IBM | 2 |
| Company of Gillette | P&G | 2 |
| Big Blue | IBM | 2 |

Table 8.1: Entity resolution of company names

### 8.2.3 Subjective Ranking of Pictures

In order to use the subjective judgement of humans to rank entities, the following schema was queried.

```
CREATE TABLE picture (
  p IMAGE,
  subject STRING
);
```

In total, 30 subject areas were chosen and 8 pictures from each subject area were ranked. Each HIT involved comparing 4 pairs of pictures, with 3 assignments each. The experiment lasted 68 minutes in total. The following is an example of a query with subjective ranking where the subject area is `'Golden Gate Bridge'`.

```
SELECT p FROM picture
WHERE subject = 'Golden Gate Bridge'
ORDER BY CROWDORDER(p, "Which picture visualizes better %subject");
```

The ranking of the top 8 pictures for `'Golden Gate Bridge'` are presented in Figure 8.7. The figure also carries information about the number of votes that the particular picture acquired, the rank assigned to it by the crowd and the rank assigned to it by a group of six experts. It is clear that the crowd provides a reasonable ranking of the pictures hence, establishing that CrowdDB is capable of using the crowd for subjective rankings.


(a) 15, 1, 1 (b) 15, 1, 2 (c) 14, 3, 4 (d) 13, 4, 5

(e) 10, 5, 6 (f) 9, 6, 3 (g) 4, 7, 7 (h) 4, 7, 8

Figure 8.7: Pictures of the Golden Gate Bridge [12] ordered by workers on AMT

## 8.3 Observations

The experiments prove that it is indeed possible to extend relational databases with crowdsourcing to enable them to answer traditionally difficult queries. The role of the CrowdDB optimizer is emphasized since it needs to make important decisions regarding the granularity of HITs, the number of HITs to post, the time of day to post, etc. In addition, automatic UI generation is very important in gathering input from the crowd. Generating a descriptive UI with precise instructions is conducive to workers submitting correct answers. Such operations are required to be performed by CrowdDB without needing intervention from the application developer. While interacting with workers on AMT, the crowd relationship manager component of CrowdDB gains particular importance. It is essential to keep the worker community satisfied in order to promote their participation in future HITs. For this purpose, a good strategy for the requester is to be reasonably lenient with rewards to workers.

# Chapter 9

# Related Work

CrowdDB is built by integrating the capabilities of relational databases and crowdsourcing. Section 9.1 covers related work in the area of relational databases. Crowdsourcing trends and studies, especially pertaining to AMT, are provided in Section 9.2. Recent related work in the area of integrating human input into query processing, much like CrowdDB, is covered in Section 9.3.

## 9.1 Relational Databases

To a large extent, the design and architecture of CrowdDB has been driven by that of traditional relational databases [32]. For CrowdDB to answer traditionally difficult queries with human input, functionalities have been added at the component level of traditional database architecture. The current prototype implementation of CrowdDB uses the code base of H2, a Java-based database [28].

Since CrowdDB integrates data from electronic data sources and human input, it bears resemblance with the systems that deal with information integration. The crowd can be viewed as an independent data source that is integrated into CrowdDB using automatically generated UIs. The challenges posed in integrating information from heterogeneous data sources and techniques to optimize queries that access them have been described in [8] and [16] respectively. In [17], techniques to store, share and access information from a federation of independent databases have been described.

The open-world issue causes the need for an explicit limit to be enforced on the amount of data that a query is expected to return. In order to address the open-world issue, CrowdDB extends the semantics of the StopAfter operator [7]. In [4], the authors propose PIQL, a Performance Insightful Query Language, which allows accurate performance prediction by providing strict bounds on the number of I/O operations. Such a solution also finds application in CrowdDB to deal with the open-world issue. Given that CrowdDB interacts with humans, it needs special means to gauge answer quality. In the information retrieval domain , several techniques to measure quality have been proposed [26]. Currently, CrowdDB uses a majority vote over the submitted assignments of each HIT.

Query optimization in CrowdDB needs to cater to configurable parameters set at compile-time and observable parameters gathered at run-time. Consequently, dynamic and adaptive query optimization techniques [5, 23, 37] may prove beneficial in the context of CrowdDB. Using the choose-plan operator to build dynamic query evaluation plans just prior to run-time has been proposed in [11, 14]. Furthermore, by dividing the range of values that the observable parameters can take into subsets, it is possible to employ parametric query evaluation plans as suggested in [20]. The approach functions by producing distinct plans based on the values of a selected subset of run-time parameters. Eddies [5] is an approach suggested to continuously adapt query processing at run-time. Scrambling query plans at run-time to cope with unexpected delays is another possible approach to dynamically alter query plans [3, 37]. In this case, an analogy can be drawn between network delay, considered in the original work, and crowd input delay in CrowdDB. Finally, CrowdDB supports special crowd comparison functions such as CrowdEqual which can be treated as expensive predicates whose optimization has been studied in [18].

## 9.2 Crowdsourcing

In recent times, crowdsourcing has become a popular means of achieving mass collaboration enabled by using Web 2.0 technologies. Harnessing the knowledge of the crowd in building smarter systems has been recognized as an important development in information technology [2].

Crowdsourcing platforms have recently been employed to build systems such as CrowdSearch and Soylent. Soylent [6] is a word processor which uses the crowd to achieve complex ends such as editing, shortening and proof-reading documents by posting tasks to Amazon Mechanical Turk. CrowdSearch [39] is another such system that uses crowdsourcing to improve the quality of image search on mobile phones. Local processing on mobile phones along with remote processing on servers is used to automate the image search. The accuracy of results returned is improved by human validation, performed by workers on Amazon Mechanical Turk.

Luis von Ahn and Laura Dabbish proposed the ESP game – a system that makes use of the crowd to label images [38]. According to their estimates, 5000 people continuously playing the game could assign a label to all images indexed by Google in 31 days. [34] is a vision paper that proposes a programming paradigm to allow creation of human-provided services within service-oriented environments. TurKit [25] is a toolkit that posts iterative tasks to AMT while using the crowd in the imperative paradigm, analogous to subroutines.

CrowdDB uses Amazon Mechanical Turk (AMT) [1] and its APIs to gather crowd input. Several studies have been performed regarding usage patterns and trends on AMT and other microtask platforms. Statistics about AMT HITs, requesters, rewards, HIT completion rates, etc. have been presented in [21]. The same report states that microtasks on AMT do not require special education and typically take less than one minute to complete. In extreme cases, tasks may need up to one hour to finish. It has also been found that the size of the AMT marketplace has been growing [33]. Despite the large and growing number of crowdsourcing workers, the number of workers available in reality to any one requester is small [24]. It has also been found that gradually workers may tend to favor certain types of tasks over others, based on their implicit learning from previous experiences [22]. Location is an important factor in the availability of workers. Workers are not evenly distributed across timezones and hence, the time of day can strongly impact the availability of workers [33]. Demographics of workers on Mechanical Turk including aspects such as age, gender and education have been studied in [33].

CrowdDB relies on gathering information from the crowd hence making the UI generator a prime component. The layout of the forms along with various GUI elements is decided based on the available data, metadata and SQL schema of the relations in the query. The approach followed by CrowdDB is similar to Oracle Forms and Microsoft Access. Oracle Forms is a software product used to create screens to access, view and edit data in Oracle databases. Microsoft Access has a similar UI feature that allows display and entry of data. Users can mix both macros and Microsoft VBA (Visual Basic for Applications) to program forms and logic for data manipulation. USHER is a research project that studies the design of data entry forms for crowdsourcing tasks with the specific intent of improving the quality of data entered using dynamic forms [9, 10].

Workers on AMT and other microtask platforms form a well-connected community. Requesters with unclear tasks or poor payment practices quickly gain bad reputations. This tends to have a long-term impact where workers are cautious about accepting tasks from the specific requester. Turker Nation [35] is a forum where active discussions regarding AMT, HITs, requesters, etc. take place. Turkopticon [36] adds functionality to AMT by providing workers with reviews of requesters.

## 9.3 Relational Databases and Crowdsourcing

Crowdsourcing has recently generated significant interest in the database community. [30] presents the design of a declarative language involving human-computable functions, standard relational operators and algorithmic computation. The query model for the same has been described using a Datalog-like

formalism. In [27], the authors propose Qurk, a query system for handling workflows that involve the crowd. The system employs user-defined functions in SQL to specify crowdworker-based expressions. Query execution is performed by asynchronous communication between the components of Qurk. Finally, in [31], the authors consider the problem of human-assisted graph search. Given a directed acyclic graph with a set of unknown target nodes as input, the goal of the search is to identify the target nodes solely by asking questions to humans.

# Chapter 10

# Conclusion and Future Work

The conclusion of this thesis is presented in Section 10.1. In Section 10.2, the status and current features of the CrowdDB prototype implementation are presented while Section 10.3 provides an summary of the possible avenues for future work in CrowdDB.

## 10.1 Conclusion

Using crowdsourcing as a means to tap the collective intelligence of humans to achieve a variety of ends is a relatively young yet promising research area. This thesis presents the design, architecture and working of CrowdDB, a database system that models crowdsourced input as relations hence permitting it to answer traditionally difficult queries. CrowdDB uses Amazon Mechanical Turk (AMT), a microtask platform, to gain access to the crowd. CrowdDB utilizes the crowd to complete missing information and perform subjective comparisons. Defining, manipulating and querying relations in CrowdDB is performed by using CrowdSQL, an extension of traditional SQL. Special semantics to deal with the open-world issue are also covered in the description of CrowdSQL. CrowdDB introduces the UI generator component in its architecture enabling UIs to be automatically generated. The functionality of the UI generator is utilized by the compile-time system and the run-time system of CrowdDB on an on-demand basis.

The current CrowdDB prototype incorporates the proposed extensions into a traditional database system, proving that it is indeed possible to build such a system with the required functionalities. Furthermore, the experiments performed demonstrate the usage of CrowdDB to answer difficult queries. They also highlight special optimization issues that CrowdDB is required to deal with. The CrowdDB optimizer is required to cater to both configurable parameters set by the application developer and observable parameters based on the state of the marketplace. Initial experiments convey the importance of sustained relationships with workers on crowdsourcing platforms. CrowdDB uses a dedicated crowd relationship manager component that ensures prompt reward payment and appropriate feedback provisions, if necessary. This helps in gradually building a community of dedicated workers.

## 10.2 Status of the Implementation

The current prototype of CrowdDB implements several features presented in this thesis. Creation of crowdsourced relations and querying them with different kinds of predicates is supported. User interfaces are automatically generated for all such cases. Support for auto-complete features and type checking is planned in future implementation work. Currently, CrowdDB supports two-way joins. In addition, primary and foreign key constraints may only be specified on single attributes. Furthermore, `AND` and `OR` conditions require special support in CrowdDB since traditional lazy evaluation rules may not apply directly. This stems from the fact that each individual predicate may be specified on a regular or crowdsourced attribute, potentially leading to crowdsourcing being interleaved with the

lazy evaluation of predicates. The current implementation of the StopAfter operator is based on the size of the result set. To extend its functionality to cater to maximum permissible response time and cost (in $) is part of future implementation work.

## 10.3 Future Work

While CrowdDB in its current state provides a means to effectively utilize crowdsourced information, several avenues for future work exist. One such avenue is to provide adaptive query optimization techniques that alter query plans at run-time, based on the state of AMT. The current CrowdDB prototype employs a simple prefetching strategy that crowdsources all incomplete attribute values in a tuple, even if just a subset of them is needed by the current query. Future work in this area includes exploring more elaborate prefetching strategies that gather extra crowd input at a marginally higher cost thereby providing significant savings for future queries. An analysis of alternative processing models in the context of CrowdDB may provide interesting results as well. Devising sophisticated ways to cache results of crowd operations, such as `CROWDORDER`, is another area of future work. Furthermore, enabling CrowdDB to process queries and generate UIs correctly for *denormalized* schemas is a possible extension. In such a case, functional dependencies would need to be explicitly defined at compile-time. Inference rules would then be used to deduce the nature of UI generation and query processing. Another aspect of future work is to evaluate methods, apart from majority voting, to measure quality of results obtained from the crowd. Exploring strategies to better interact with workers, including improved UI design and feedback mechanisms, is an ongoing process in the development of CrowdDB.

# Bibliography


[1] Amazon Mechanical Turk. `http://www.mturk.com`.

[2] Sihem Amer-Yahia, AnHai Doan, Jon M. Kleinberg, Nick Koudas, and Michael J. Franklin. Crowds, Clouds, and Algorithms: Exploring the Human Side of "Big Data" Applications. In *SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1259–1260, 2010.

[3] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling Query Plans to Cope with Unexpected Delays. In *PDIS '96: Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems*, pages 208–219, 1996.

[4] Michael Armbrust, Stephen Tu, Armando Fox, Michael J. Franklin, David A. Patterson, Nick Lanham, Beth Trushkowsky, and Jesse Trutna. PIQL: A Performance Insightful Query Language. In *SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1207–1210, 2010.

[5] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[6] Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. Soylent: A Word Processor with a Crowd Inside. In *UIST '10: Proceedings of the 2010 ACM Symposium on User Interface Software and Technology*, pages 313–322, New York, NY, USA, 2010. ACM.

[7] Michael J. Carey and Donald Kossmann. On Saying "Enough Already!" in SQL. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 219–230, 1997.

[8] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ '94: Proceedings of the 1994 Meeting of the Information Processing Society of Japan*, pages 7–18, 1994.

[9] Kuang Chen, Harr Chen, Neil Conway, Joseph M. Hellerstein, and Tapan S. Parikh. USHER: Improving Data Quality with Dynamic Forms. In *ICDE '10: Proceedings of the 2010 IEEE International Conference on Data Engineering*, pages 321–332, 2010.

[10] Kuang Chen, Joseph M. Hellerstein, and Tapan S. Parikh. Designing Adaptive Feedback for Improving Data Entry Accuracy. In *UIST '10: Proceedings of the 2010 ACM Symposium on User Interface Software and Technology*, 2010.

[11] Richard L. Cole and Goetz Graefe. Optimization of Dynamic Query Evaluation Plans. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 150–160, 1994.

[12] Flickr. Golden Gate Bridge. Pictures of the Golden Gate Bridge retrieved from Flickr by akaporn, Dawn Endico, devinleedrew, di_the_huntress, Geoff Livingston, kevincole, Marc_Smith, and superstrikertwo under the Creative Commons Attribution 2.0 Generic license.

[13] Michael Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. CrowdDB: Answering Queries with Crowdsourcing. In *SIGMOD '11: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[14] Goetz Graefe and Karen Ward. Dynamic Query Evaluation Plans. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 358–366, 1989.

[15] SQL-92 BNF Grammar. `http://savage.net.au/SQL/sql-92.bnf.html`.

[16] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *VLDB '97: Proceedings of the 1997 International Conference on Very Large Databases*, pages 276–285, 1997.

[17] Dennis Heimbigner and Dennis McLeod. A Federated Architecture for Information Management. *TOIS '85: Proceedings of 1985 ACM Transactions on Office Information Systems*, 3(3):253–278, 1985.

[18] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 267–276, 1993.

[19] Eric Huang, Haoqi Zhang, David C. Parkes, Krzysztof Z. Gajos, and Yiling Chen. Toward Automatic Task Design: A Progress Report. In *HCOMP '10: Proceedings of the 2010 ACM SIGKDD Workshop on Human Computation*, 2010.

[20] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. In *VLDB '92: Proceedings of the 1992 International Conference on Very Large Databases*, pages 103–114, 1992.

[21] Panagiotis G. Ipeirotis. Analyzing the Amazon Mechanical Turk Marketplace. *XRDS: The ACM Magazine for Students*, 17:16–21, December 2010.

[22] Panagiotis G. Ipeirotis. Mechanical Turk, Low Wages, and the Market for Lemons. `http://behind-the-enemy-lines.blogspot.com/2010/07/mechanical-turk-low-wages-and-market.html`, 2010.

[23] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 106–117, 1998.

[24] Greg Little. How Many Turkers Are There? `http://groups.csail.mit.edu/uid/deneme/?p=502`, 2009.

[25] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. TurKit: Tools for Iterative Tasks on Mechanical Turk. In *HCOMP '09: Proceedings of the 2009 ACM SIGKDD Workshop on Human Computation*, pages 29–30, New York, NY, USA, 2009. ACM.

[26] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[27] Adam Marcus, Eugene Wu, Sam Madden, and Rober Miller. Crowdsourced Databases: Query Processing with People. In *CIDR '11: Proceedings of the 2011 Conference on Innovative Data Systems Research*, 2011.

[28] Thomas Mueller. H2 Database. http://www.h2database.com.

[29] Thomas Mueller. H2 Database Performance. http://www.h2database.com/html/performance.html.

[30] Aditya Parameswaran and Neoklis Polyzotis. Answering Queries using Humans, Algorithms and Databases. In *CIDR '11: Proceedings of the 2011 Conference on Innovative Data Systems Research*, 2011.

[31] Aditya Parameswaran, Anish Das Sarma, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Human-Assisted Graph Search: It's Okay to Ask Questions. In *VLDB '11: Proceedings of the 2011 International Conference on Very Large Databases*, volume 4, pages 267–278. VLDB Endowment, February 2011.

[32] Raghu Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[33] Joel Ross, Lilly Irani, M. Six Silberman, Andrew Zaldivar, and Bill Tomlinson. Who are the Crowdworkers? Shifting Demographics in Mechanical Turk. In *CHI '10: Proceedings of the 2010 ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2863–2872, 2010.

[34] Daniel Schall, Schahram Dustdar, and M. Brian Blake. Programming Human and Software-Based Web Services. *IEEE Computer Society*, 43(7):82–85, 2010.

[35] Turker Nation. http://www.turkernation.com/.

[36] Turkopticon. http://turkopticon.differenceengines.com/.

[37] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based Query Scrambling for Initial Delays. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 130–141, 1998.

[38] Luis von Ahn and Laura Dabbish. Labeling Images with a Computer Game. In *CHI '04: Proceedings of the 2004 ACM Conference on Human Factors in Computing Systems*, pages 319–326, 2004.

[39] Tingxin Yan, Vikas Kumar, and Deepak Ganesan. CrowdSearch: Exploiting Crowds for Accurate Real-time Image Search on Mobile Phones. In *MobiSys '10: Proceedings of the 2010 International Conference on Mobile Systems, Applications, and Services*, pages 77–90, 2010.