

CROWDSOURCING STRUCTURED DATA

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Hyunjung Park
June 2014

© 2014 by Hyunjung Park. All Rights Reserved.

Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.

<http://creativecommons.org/licenses/by-nc/3.0/us/>

This dissertation is online at: <http://purl.stanford.edu/cn920wc7145>

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Neoklis Polyzotis

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Crowdsourcing can be used to incorporate human computation into a variety of data-intensive tasks that are difficult for computers alone to solve well. Crowd-powered algorithms treat humans as processing units, while crowdsourced data uses humans as a data source. We present two different approaches to the second problem: collecting data from the crowd.

We first present Deco, a system for “declarative crowdsourcing.” Given a declarative query posed over a relational database, Deco uses the microtask approach to ask specific questions to the crowd, augmenting existing data to produce the query result. After briefly describing Deco’s data model and query language, we focus on how Deco’s query execution engine and query optimizer work together to produce high-quality query results while minimizing monetary cost and reducing latency.

Second, we present CrowdFill, an alternative approach for collecting structured data from the crowd. Instead of posing specific questions as microtasks, CrowdFill shows a partially-filled table to all participating workers; these workers contribute by filling in empty cells, as well as upvoting and downvoting data entered by other workers. We describe how the system uses our primitive operations to guide data collection towards prespecified constraints while providing an intuitive data entry interface. We also describe CrowdFill’s compensation scheme that encourages useful work while adhering to a monetary budget.

Contents

Abstract	iv
1 Introduction	1
1.1 Approaches for Crowdsourcing Structured Data	2
1.1.1 Microtask-based Approach	2
1.1.2 Table-filling Approach	4
1.1.3 Comparison of the Two Approaches	5
1.2 Overview of Contributions	7
1.2.1 Deco	8
1.2.2 CrowdFill	10
1.3 Summary	11
2 Foundations of Deco	12
2.1 Data Model	12
2.1.1 Conceptual Relation	12
2.1.2 Raw Schema	13
2.1.3 Fetch Rules	14
2.1.4 Resolution Rules	14
2.1.5 Data Model Semantics	15
2.2 Query Language and Semantics	16
2.3 System Overview	17
2.4 Related Work	18

3	Query Execution in Deco	20
3.1	Challenges and Approach	21
3.1.1	Executing Queries in Two Phases	21
3.1.2	Enabling Parallelism Using Asynchronous Pull	21
3.1.3	Choosing Right Degree of Parallelism	22
3.1.4	Initiating Good Fetches	22
3.1.5	Changing Result Incrementally	23
3.2	Query Operators and Execution Plans	23
3.2.1	Query Operators	23
3.2.2	Query Plans	25
3.3	Query Execution with No Existing Data	30
3.3.1	Basic Query Plan	32
3.3.2	Reverse Query Plan	35
3.3.3	Combined Query Plan	36
3.3.4	Hybrid Query Plan	36
3.3.5	Join of Conceptual Relations	37
3.4	Query Execution with Existing Data	38
3.4.1	Materialization Phase	38
3.4.2	Accretion Phase	40
3.4.3	Meeting Parallelism Objectives	42
3.5	Fetch Prioritization	44
3.5.1	Formal Problem Definition	45
3.5.2	Heuristic Algorithm	49
3.5.3	Query Execution Engine Extension	50
3.5.4	Amazon Mechanical Turk Support	51
3.6	Experimental Evaluation	52
3.6.1	Performance of Different Query Plans	52
3.6.2	Parallelism, Cost, and Latency	54
3.6.3	Effectiveness of Fetch Prioritization	56
3.7	Related Work	60
3.8	Conclusion	61

4	Query Optimization in Deco	62
4.1	Challenges and Approach	63
4.1.1	Cost and Cardinality Estimation	64
4.1.2	Statistics	64
4.1.3	Plan Enumeration	65
4.2	Cost Estimation	65
4.2.1	Cardinality Estimation Algorithm	68
4.2.2	Cost Estimation Examples	74
4.3	Search Space and Plan Generation	77
4.3.1	Join Tree	78
4.3.2	Algebraic Representation	81
4.3.3	Fetch Rule Selection	81
4.3.4	Complete Query Plan	82
4.4	Enumeration Algorithm	83
4.4.1	Naive Enumeration	84
4.4.2	Efficient Enumeration	85
4.5	Experimental Evaluation	86
4.5.1	Accuracy of Cost Estimation	86
4.5.2	Efficiency of Plan Enumeration	94
4.6	Related Work	95
4.7	Conclusion	95
5	Design and Implementation of CrowdFill	97
5.1	Formal Model	98
5.1.1	Table Specification	98
5.1.2	Table State and Primitive Operations	100
5.1.3	Constraints	101
5.1.4	Concurrent Operations	104
5.2	System Overview	110
5.2.1	Architecture	110
5.2.2	Front-end Server	112
5.2.3	Back-end Server	113

5.2.4	Worker Client	113
5.3	Satisfying the Constraints	114
5.3.1	Probable Rows Invariant	115
5.3.2	Maintaining the Invariant	116
5.3.3	Probable Rows Invariant Maintenance Example	117
5.4	Compensating Workers	119
5.4.1	Challenges and Approach	119
5.4.2	Allocating Total Budget to Workers	120
5.4.3	Estimating Compensation	125
5.5	Experimental Evaluation	127
5.6	Related Work	130
5.7	Conclusion	131
6	Summary and Future Work	133
6.1	Summary of Contributions	133
6.2	Contrasting Our Two Approaches	134
6.3	Future Work	135
6.3.1	Deco	135
6.3.2	CrowdFill	136
6.3.3	Crowdsourcing Structured Data	137
	Bibliography	138

List of Tables

1.1	Comparison of the microtask-based and table-filling approaches . . .	6
3.1	Message exchanges during the accretion phase (empty raw tables) .	32
3.2	Message exchanges for materialization	39
3.3	Message exchanges to start accretion phase	41

List of Figures

1.1	An example partially-filled table	3
2.1	Deco architecture	17
2.2	Deco user interface: query execution	18
2.3	Deco user interface: query plan visualization	19
3.1	Basic query plan	26
3.2	Reverse query plan	27
3.3	Combined query plan	28
3.4	Hybrid query plan	29
3.5	Performance of different query plans	53
3.6	Interactions among parallelism, cost, and latency	55
3.7	Effectiveness of fetch prioritization (Experiment 3)	58
3.8	Effectiveness of fetch prioritization (Experiment 4)	60
4.1	Example query plans	67
4.2	Trace of EstimateCard	75
4.3	Example join trees	80
4.4	Accuracy of cost estimation (Experiment 1)	87
4.5	Accuracy of cost estimation (Experiment 2, part 1)	89
4.6	Accuracy of cost estimation (Experiment 2, part 2)	90
4.7	Accuracy of cost estimation (Experiment 3, part 1)	91
4.8	Accuracy of cost estimation (Experiment 3, part 2)	93
4.9	Efficiency of plan enumeration	94
5.1	CrowdFill architecture	110

5.2	Table schema editor	111
5.3	Data entry interface	112
5.4	Bipartite graph representation of the PRI maintenance	118
5.5	Accuracy of estimated compensation	129
5.6	Earning rates for uniform and weighted allocation, two workers . .	130

Chapter 1

Introduction

Many modern data-processing tasks are difficult for computer algorithms alone to solve well, as the tasks require full understanding of data presented in a variety of forms. *Crowdsourcing* [22] can help accomplish such tasks by providing a programmatic way of incorporating the use of human abilities into computer algorithms. In a typical crowdsourcing scenario, human *workers* are asked to perform computation as if they were processing units, or generate data as if they were a data source.

In data-intensive tasks, perhaps the most common usage of crowdsourcing to date has been annotating a set of data items with human inputs, e.g., video or image labeling [56, 59], search relevance judgements [10], and natural language annotation tasks [55]. More recently, “crowd-powered” algorithms have been developed to better process existing datasets by leveraging humans, e.g., filtering items [40], finding a maximum item [31, 58], sorting and joins [38, 61], and entity resolution [60, 62]. Another important usage of crowdsourcing is creating new data, or augmenting an existing dataset, based on human inputs, which is the topic of this thesis.

In collecting data from the crowd, as well as in other crowdsourcing tasks, there are three important considerations. First, human workers need to be paid, incurring *monetary cost*. Second, overall *latency* of a computer-driven task is much longer once humans are involved. Third, human workers make mistakes, affecting the *quality* of result. Overall, many new issues and tradeoffs involving cost,

latency, and quality make dealing with crowdsourced data quite challenging. We will see in later chapters that an underlying principle of the systems we have developed is to optimize for some combination of cost, latency, and quality.

1.1 Approaches for Crowdsourcing Structured Data

In this thesis we consider crowdsourcing *structured data*: data that adheres to an explicitly-specified, fixed schema. Structured data is well-understood and easy to deal with, letting us focus specifically on the new challenges of gathering data from the crowd, and integrating crowdsourced data with existing data and processing techniques. Structured data most commonly follows the *relational data model* [17, 27]. In this model, data is represented as a two-dimensional table called a *relation*, or simply a *table*. In a table, each row typically corresponds to a data element or a fact, while each column is a property or attribute of the element or fact. Figure 1.1 is an example of a table (with some empty rows and missing column values, explained shortly) with information about countries. All of our work in this thesis considers relational data.

In the thesis we explore two complementary approaches for crowdsourcing structured data: a *microtask-based* approach, and a *table-filling* approach. In the remainder of this section we provide a high-level description of each approach, including a running example for motivation, then we discuss the advantages and disadvantages of the two approaches.

1.1.1 Microtask-based Approach

Our first approach is based on *microtasks*: tasks that are designed to be completed by human workers in a relatively short duration (e.g., in a few minutes or less). For example, a microtask might ask a worker to label a given image or to rate a piece of text based on its sentiment. *Users* who compose microtasks can post them on crowdsourcing *marketplaces* such as Amazon Mechanical Turk [1] and specify a monetary compensation they are willing to offer for each microtask. Human workers can browse through available microtasks on a marketplace and complete

country	language	capital
Chile		Santiago
Peru	Spanish	
Austria		
Germany		
Korea	Korean	Seoul

Figure 1.1: An example partially-filled table

the ones they choose. Once a submitted answer to a microtask is approved by the user who posted it, the marketplace collects the specified compensation from the user and pays the worker on behalf of the user.

Since microtasks posted on a marketplace can be completed independently by different workers, crowdsourcing marketplaces give users on-demand, parallel access to many human workers. However, data-intensive tasks that users want to accomplish via crowdsourcing do not usually fit in a single microtask. Thus, to benefit from crowdsourcing marketplaces, special algorithms, or the users themselves, must decompose overall tasks into sets of microtasks.

Collecting structured data using a microtask-based approach thus begins with decomposing the data collection task into a set of microtasks. Once the microtasks are posted on a marketplace and completed by workers, submitted answers are assembled into table form. As a running example, let us consider the table shown in Figure 1.1. This example table is partially-filled because we might have some data already collected from humans or obtained by other means, and we would like to reuse the data instead of starting from scratch. Let us suppose our goal is to fill in all empty rows and cells in the example table with no duplicate values for the country column. In other words, we want complete language and capital information for eight countries.

There are many possible ways of decomposing our example task into a set of microtasks, corresponding to different partitionings of the table. Suppose we partition the table in the finest granularity, making each microtask responsible for

filling in a cell. To fill in the empty language cell in the first row, we may pose the following question as a microtask: “What is the language spoken in Chile?” A corresponding question can be posed to obtain language values for Austria and Germany. Likewise, we can collect the capital value for Peru using the microtask: “What is the capital city of Peru?” Notice that we can pose these questions only for countries already known to us. To fill in the three empty rows, we can first ask for a country name: “Provide a country name.” Once we obtain a new country name, we instantiate the previous microtasks to collect its language and capital values. As illustrated here, not all microtasks are posted at once; some may be created later based on the answers to the completed microtasks.

Alternatively, we may always want to ask for language and capital values at the same time, e.g., “What are the official language and capital city of Austria?” This type of question might be somewhat wasteful for those countries whose language or capital value is already present, such as Chile and Peru. On the other hand, workers might be able to provide both values with lower overall cost and/or latency than obtaining them separately, benefiting rows for Austria and Germany. In general, different partitioning yields different costs and latencies in data collection, and finding an optimal partitioning is a challenging problem.

As we mentioned earlier, in addition to cost and latency, the quality of crowd-sourced data is an important consideration. To improve the quality of collected data in the microtask-based approach, we can instantiate multiple identical microtasks for the same question and resolve inconsistencies of the submitted answers using voting, averaging, or some other means. This redundancy not only makes finding an optimal partitioning even more complicated, but it also makes assembling the final table more complex.

1.1.2 Table-filling Approach

In the microtask-based approach, table partitioning is the primary means to enable parallelism among workers in collecting data. Our table-filling approach enables parallelism in a more holistic fashion: We do not partition a table, but instead allow multiple workers to view and concurrently modify the entire table.

Specifically, we show a copy of an entire partially-filled table (such as the one in Figure 1.1) to each participating worker and ask workers to contribute what they know by filling in empty cells in the displayed table. We immediately propagate each data entry by a worker to all other copies of the table displayed elsewhere, so workers can collaboratively enter data while the displayed table is updated in real time.

Let us consider again the partially-filled table shown in Figure 1.1. If workers take turns in filling in empty cells, our table-filling approach operates in a straightforward manner: Each data entry simply takes the table one step closer to a completion. However, under this assumption, workers cannot enter data in a parallel fashion, and waste time waiting for their turns.

When workers are allowed to fill in empty cells concurrently, conflicts can occur when two workers fill in empty cells in the same row (either for the same cell or for different cells) at the same time. For example, suppose one worker fills in Mexico for country in one of the empty rows, and another worker fills in Athens for capital in the same empty row at the same time. Without proper resolution, the table ends up in a state no worker intended to reach. A more basic conflict occurs when two workers fill in the same cell with different values. Intuitive and seamless conflict resolution is a key component that makes the table-filling approach work in practice.

As in the microtask-based approach, data quality concerns make things much more complicated. Since new data entries are propagated and displayed immediately to all workers, our approach to ensuring quality of the collected data is to ask workers to upvote and downvote rows entered by other workers. Enough downvotes on a row suggest that the row has incorrect values, and the row is removed from the table. Additionally, when rows conflict (such as two different rows with the same key value), the one with the most upvotes is selected.

1.1.3 Comparison of the Two Approaches

Table 1.1 summarizes advantages and disadvantages of the two approaches considered in the thesis. We discuss each of the five comparisons listed in the table.

Microtask-based approach	Table-filling approach
+ More scalable	– Less scalable
– Nontransparent	+ Transparent
– Passive	+ Active
– Inflexible	+ Flexible
+ Suitable for combining humans and computers	– Targeted to humans only

Table 1.1: Comparison of the microtask-based and table-filling approaches

Scalability

The microtask-based approach is fairly scalable in number of workers as well as in data size. Since each microtask is completed independently by one worker, each worker has the same efficiency regardless of the total number of concurrent workers or the total number of microtasks. The table-filling approach is not as scalable as the microtask-based approach. As more workers fill in the table concurrently, more conflicts may occur, potentially wasting work and thus decreasing overall efficiency of each worker. Also, individual workers will tend to be more efficient at filling in a smaller table than a larger one, due to human attention and comprehension, and data-entry interface issues.

Transparency

From the perspective of workers, the microtask-based approach is nontransparent: Each microtask presents a self-contained question asking for a piece of data, so individual workers do not know what other workers are doing, nor what the user's overall intention is. In contrast, the table-filling approach is transparent: Workers can see data entered by other workers, as well as the user's overall intention. This transparency allows workers to learn from data entered by other workers, and to satisfy constraints specified by the user, e.g., not entering duplicate values.

Active participation

In the microtask-based approach, workers are relatively passive in selecting what work they do. In Amazon Mechanical Turk, for example, workers first choose a

group of microtasks, which include different instantiations of one question type. Within the chosen group, workers usually perform a series of preselected microtasks. (In principle workers could select specific microtasks out of the series, but doing so might significantly lower their overall efficiency and compensation, given current worker interfaces.) In the table-filling approach, workers actively identify those parts of the table they can contribute to best, and select which data they feel most qualified to endorse or refute.

Flexibility

In the microtask-based approach, workers do not have much flexibility in data entry. The order of data entry is predetermined by the types of questions available. Similarly the granularity of data entry is set beforehand based on the table partitioning. In contrast, the table-filling approach provides workers with a great deal of flexibility: Workers can fill in values in arbitrary order, and they can enter data at any granularity as long as there are empty cells.

Combining humans and computers

Although this thesis focuses primarily on collecting data from human workers, the microtask-based approach is also suitable for combining data from humans with computer-collected data: The microtasks we create conform to fixed input-output signatures based on table partitioning. For example, if we have a programmed function that takes as input a country name and produces as output its capital city, we can plug that function directly into our solution, in place of human workers. While in theory such a function could be incorporated into the table-filling approach as well, the integration would be far less seamless.

1.2 Overview of Contributions

The goal of our thesis work was to design and implement systems for crowdsourcing structured data, exploring both of the two complimentary approaches

described in the previous section. Here we motivate and outline the primary technical contributions of the thesis.

1.2.1 Deco

As discussed in Section 1.1.1, collecting data using the microtask-based approach involves several difficult decisions such as finding an optimal table partitioning and selecting question types to instantiate. To hide these complexities from users, we want a *declarative* system, where users express their needs at a high level, and our system determines the best strategy for collecting data from workers. Since our goal is to collect relational data, a declarative database query language is a natural choice for users to specify their needs. Moreover, we want our system to transparently manage existing data together with crowdsourced data, thus an extended database system is a natural approach here as well.

In Chapters 2-4, we present *Deco*, a database system for “declarative crowdsourcing” that collects structured data using the microtask-based approach. Developing a new type of database system involves first establishing its data model and query language, then implementing a system supporting the model and language. Typically, the goal is to reuse previous models, languages, and implementation techniques to the extent possible, making modifications and extensions as needed for the novel challenges and features. In *Deco*, the biggest challenges and novelty involved designing suitable query semantics, and implementing a query processor that answers a *Deco* query while minimizing the monetary cost and latency incurred by microtasks.

Data Model, Query Language, and System

To develop a new database system that supports crowdsourced data, we first need to design a data model and a query language. Designing an intuitive, expressive, and implementable data model and query language can involve significant effort and consideration of many alternatives [41]. Once the data model and query language have been defined completely, implementing a system to support them can also involve many nontrivial challenges.

In Chapter 2, we briefly describe Deco’s data model and query language, then describe the overall architecture of the Deco system.¹ As we will explain in Chapter 2, Deco’s data model is an extension to the relational model, designed to be general, flexible, and principled. Deco’s query language is a simple syntactic extension to SQL, incorporating certain constraints necessary for the crowdsourcing environment. Deco’s query semantics is based directly on SQL semantics, but with some important extensions for the new environment. Deco’s overall architecture is similar to a traditional database system, however there are many significant differences in its query execution engine and query optimizer, discussed next.

Query Execution

Database systems typically use an internal *query execution plan* to represent how data is accessed and processed to produce a query result. A query plan is composed of *query operators*, each of which corresponds to a particular data processing step, such as filtering or sorting. To use a similar approach in Deco, we first needed to design several new query operators, then we considered the overall structure of query plans. It turns out that the structure of Deco query plans looks similar to traditional query plans, but our objective of minimizing cost and reducing latency requires significant changes in the details of query execution.

In Chapter 3, we describe Deco’s query plans, new query plan operators, and query execution engine in detail. Deco’s new query operators implement data operations specific to the crowdsourcing environment, e.g., instantiating microtasks and resolving inconsistencies of answers. Deco’s query execution engine uses a new “hybrid” execution model, which respects Deco semantics while enabling our objective. In this execution model, query execution bears as much similarity to *incremental view maintenance* [12] as to a traditional *iterator* model [27]. Our objective also requires prioritizing accesses to crowdsourced data, which turns out to be an interesting NP-hard problem.

¹Deco’s data model and query language were the result of a group effort, while the Deco architecture and system were developed primarily by the author.

Query Optimization

In traditional database systems, for a given query there are usually many different possible query plans, all of which produce a correct query result. Executing the different plans may incur vastly different costs, so it is very important to find the best query plan, which is the problem of *query optimization*. A typical query optimizer consists of three components: a cost estimation algorithm (for a given query plan), a search space of valid query plans, and a plan enumeration algorithm. In Deco, our goal in query optimization is to find the best plan to answer a given Deco query in terms of estimated monetary cost.

In Chapter 4, we describe Deco’s query optimizer in detail. To accurately estimate the monetary cost of executing a given query plan, Deco’s cost model distinguishes between free existing data versus paid new data, and Deco’s cost estimation algorithm copes with changes to the database state as data is gathered from workers during query execution. When exploring the search space and applying the cost estimation algorithm, Deco’s plan enumeration algorithm maximizes reuse of common subplans, which turns out to be particularly challenging in our setting.

1.2.2 CrowdFill

In Chapter 5, we present *CrowdFill*, a separate system from Deco that implements the table-filling approach described in Section 1.1.2. In CrowdFill, one of our biggest initial challenges involved designing a formal model for table states and for primitive operations such as “fill” and “upvote”, to serve as the basis for our data-entry interface. The model is carefully designed to accommodate concurrent operations, with conflicts resolved in an intuitive and seamless fashion. We will see that our approach enables us to prove a strong result about table synchronization and consistency across multiple workers.

As described in Section 1.1, we often start data collection from a partially-filled table with some prespecified values, which CrowdFill treats as constraints on the final table. Ensuring that the final table meets the constraints is nontrivial: Enough downvotes may effectively remove rows with prespecified values from

the table; those values need to be restored to satisfy the constraints. To provide an intuitive data-entry interface while guiding the final table towards meeting the constraints, CrowdFill only allows new rows (with prespecified values) to be inserted into the table by a central monitoring client. With this approach, workers never need to add rows, and they need not be aware of the constraints, allowing them to simply fill in empty values in existing rows, and cast votes. We will see that the monitoring client populates new rows using an algorithm based on maximum bipartite matching.

Lastly, we needed to devise a compensation scheme that encourages useful work, provides compensation commensurate with a worker's efforts, yields high-quality data, and adheres to a fixed monetary budget. Instead of offering fixed compensation for each data entry or voting action, CrowdFill's compensation scheme is based on each worker's overall contribution to the final table, as well as variability in the difficulty of filling in cells. At the same time, CrowdFill displays estimated compensation for individual actions during table-filling, to keep workers engaged and focused on entering the needed data.

1.3 Summary

Crowdsourcing structured data is a new research area, with new challenges stemming from issues involving monetary cost, latency, and data quality. In this thesis we present two systems for collecting structured data from the crowd: Deco with a microtask-based approach, and CrowdFill with a table-filling approach. The two approaches are complimentary with different advantages and disadvantages, and each of them presents numerous technical challenges. After briefly describing Deco's data model, query language, and overall architecture in Chapter 2, we describe Deco's query execution engine and query optimizer in Chapters 3 and 4, respectively. Then, we switch to CrowdFill in Chapter 5, describing its formal model, system architecture, constraint satisfaction algorithm, and compensation scheme. (We discuss related work at the end of each chapter.) In Chapter 6, we conclude with future directions for Deco and CrowdFill, as well as the general area of crowdsourcing structured data.

Chapter 2

Foundations of Deco

In this chapter we lay foundations for *Deco* (for “declarative crowdsourcing”), a system that answers declarative queries posed over stored relational data together with data gathered on-demand from the crowd.¹ In Section 2.1, we define a data model for Deco that is general, flexible, and principled. In Section 2.2, we define a query language for Deco as a simple extension to SQL with constructs necessary for crowdsourcing. In Section 2.3, we describe Deco’s overall architecture. In Section 2.4, we discuss related work.

2.1 Data Model

We begin by describing each of the Deco data model components using a running example.

2.1.1 Conceptual Relation

Conceptual relations are the logical relations specified by the Deco schema designer and queried by end-users and applications. The schema designer also

¹The material presented in this chapter first appeared in [42, 43, 44, 45]. Deco’s data model and query language (Sections 2.1-2.2) were the result of a group effort, but are presented here as necessary background for the remainder of the thesis. The Deco architecture and system (Section 2.3) were developed primarily by the author.

partitions the attributes in each conceptual relation into *anchor attributes* and *dependent attribute-groups*. Informally, anchor attributes typically identify “entities” while dependent attribute-groups specify properties of the entities.

As a running example, suppose our users are interested in querying two conceptual relations with information about countries and cities:

```
Country(country, [language], [capital])
City(city, country, [population])
```

Each dependent attribute-group (single attributes in this case) is enclosed within square brackets. In the Country relation, the anchor attribute country is a country name, while language and capital are properties of the country. In the City relation, the pair of city and country names identifies a city, while population is a property of the city.

2.1.2 Raw Schema

Deco is designed to use a conventional RDBMS as its back-end. The *raw schema*—the schemas for the data tables actually stored in the underlying RDBMS—is derived automatically from the conceptual schema, and is invisible to both the schema designer and end-users. Raw tables contain existing data obtained by past queries or otherwise present in the database, and are extended as new data is obtained from the crowd, enabling seamless integration of conventional data and crowdsourced data. For each relation R in the conceptual schema, there is one *anchor table* containing the anchor attributes, and one *dependent table* for each dependent attribute-group; dependent tables also contain anchor attributes. (In general, both anchor and dependent attributes can be a group of attributes.)

In our example, we have the raw schema:

```
CountryA(country)
CountryD1(country, language)
CountryD2(country, capital)
CityA(city, country)
CityD1(city, country, population)
```


2.1.3 Fetch Rules

Fetch rules allow the schema designer to specify how data can be obtained from humans. A fetch rule takes the form $A_1 \Rightarrow A_2 : P$, where A_1 and A_2 are sets of attributes from one conceptual relation (with $A_1 = \emptyset$ permitted), and P is a *fetch procedure* that implements access to human workers. (P might generate HITs (Human Intelligence Tasks) to Amazon Mechanical Turk [1], for example.) When invoked, the fetch rule $A_1 \Rightarrow A_2$ obtains new values for A_2 given values for A_1 , and populates raw tables using those values for attributes $A_1 \cup A_2$. The schema designer also specifies a fixed monetary cost for each fetch rule, to be paid to human workers once they complete the fetch rule.

Here are some example fetch rules and their interpretations for our example.

- [Country] $\emptyset \Rightarrow$ country: Ask for a country name, inserting the obtained value into raw table CountryA.
- [Country] country \Rightarrow capital: Ask for a capital given a country name, inserting the resulting pair into table CountryD2.
- [Country] language \Rightarrow country: Ask for a country name given a language, inserting the resulting country name into CountryA, and inserting the country-language pair into CountryD1.

Note there are many more possible fetch rules for our example. The only restriction on fetch rules is that $A_1 \cup A_2$ must include all anchor attributes.

The schema designer may add or remove fetch rules at any time during the lifetime of a database—they are more akin to “access methods” than to part of the permanent schema.

2.1.4 Resolution Rules

Suppose we have obtained values for our raw tables, but we have inconsistencies in the collected data. We use *resolution rules* to cleanse the raw tables—to get values for conceptual relations that are free of inconsistencies. For each conceptual relation, the schema designer can specify a resolution rule $\emptyset \rightarrow A : f$ for the anchor attributes A treated as a group, and one resolution rule $A \rightarrow D : f$ for each dependent attribute-group D . Resolution function f is a black-box that

adheres to a simple API, taking as input a set of values for the right-hand side attributes (corresponding to a specific value for the left-hand side) and returning a “cleaned” set of values. If the empty set is returned, more input values are needed to produce an output. In addition, the schema designer should specify the minimum and average number of input values needed for f to produce an output value. These numbers are needed for plan execution and cost estimation, as we will see in Chapters 3 and 4.

In our example, we might have the following resolution rules:

- [Country] $\emptyset \rightarrow$ country : *dupElim*
- [Country] country \rightarrow language : *majority-of-3*
- [Country] country \rightarrow capital : *majority-of-3*
- [City] $\emptyset \rightarrow$ city,country : *dupElim*
- [City] city,country \rightarrow population : *average-of-2*

Resolution function *dupElim* produces distinct country values for Country and distinct city-country pairs for City. Resolution function *majority-of-3* produces the majority of three or more language (or capital) answers for a given country. We assume a “shortcutting” version that can produce an answer with only two values, if the values agree. On the other hand, sometimes more than three values are input to the function, in which case a majority is needed to produce an output. Resolution function *average-of-2* produces the average of two or more population answers for a given city. Note any resolution functions are permitted, not just the types used here for illustration.

2.1.5 Data Model Semantics

The semantics of a Deco database is defined as a potentially infinite set of *valid instances* for the conceptual relations. A valid instance is obtained by a *Fetch-Resolve-Join* sequence: (1) *Fetching* additional data for the raw tables using fetch rules; this step may be skipped. (2) *Resolving* inconsistencies using resolution rules for each of the raw tables. (3) *Outerjoining* the resolved raw tables to produce the conceptual relations.

It is critical to understand that the Fetch-Resolve-Join sequence is a *logical concept only*. As we will see in later chapters, when Deco queries are executed, not only may these steps be interleaved, but only those portions of the conceptual data needed to produce the query result are actually materialized.

2.2 Query Language and Semantics

A Deco query Q is simply a SQL query over the conceptual relations. Deco's query semantics dictate that the answer to Q must represent the result of evaluating Q over some (logical) valid instance of the database. Because our Fetch-Resolve-Join database semantics is based on outerjoins, conceptual relations may logically include numerous NULL values. Thus, we restrict query results to include only those conceptual tuples whose attribute values are non-NULL, although this restriction could be lifted if desired.

One valid instance of the database can be obtained by resolving and joining the current contents of the raw tables, without invoking any further fetch rules. Thus, it appears a query Q can always be answered correctly without consulting the crowd at all. The problem is that this "correct" query result may be very small, or even empty. To retain our straightforward semantics over valid instances, while still forcing answers to contain some amount of data, we add one of the following *constraints* to a Deco query:

- "MaxCost c " specifies to spend up to c dollars (or other cost unit) to answer the query, while attempting to maximize the number of tuples in the result.
- "MaxTime t " specifies to spend up to t seconds (or other time unit) to answer the query, while attempting to maximize the number of tuples in the result.
- "MinTuples n " specifies that at least n tuples should be in the result, while attempting to minimize cost (or time).

In this thesis we focus on the third type of constraint: producing a minimum number of (non-NULL) result tuples while minimizing cost. We will discuss the other two types briefly in Section 6.3.

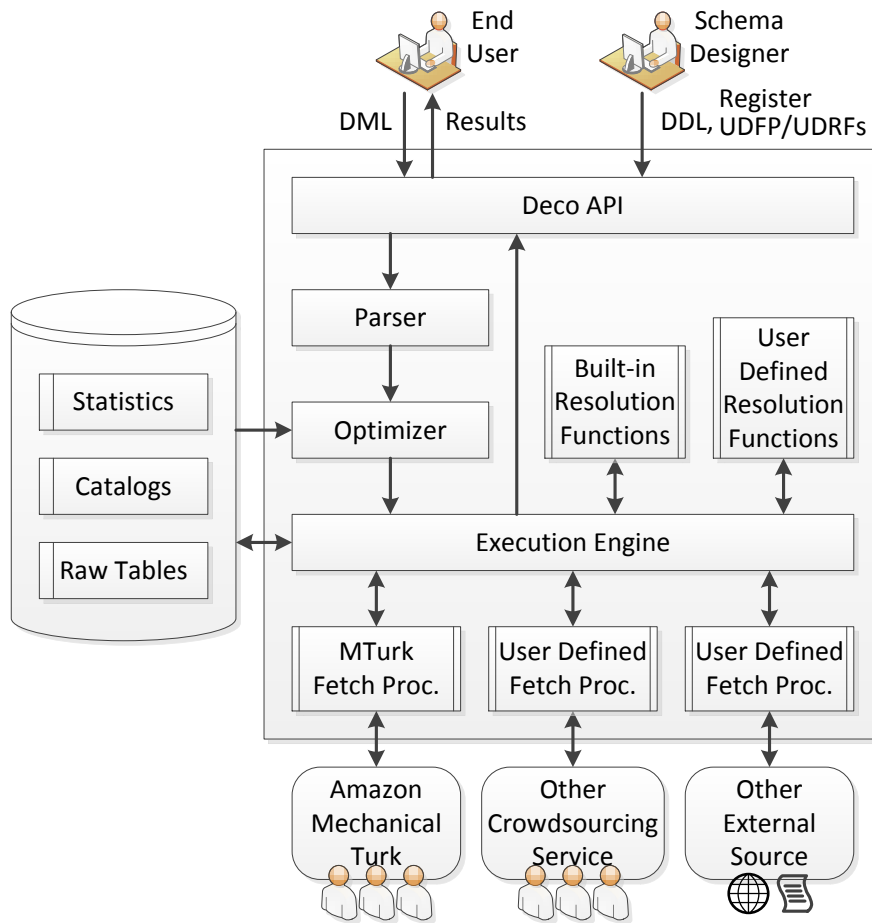


Figure 2.1: Deco architecture

2.3 System Overview

We implemented our Deco prototype in Python with a PostgreSQL back-end. The system supports DDL commands to create tables, resolution functions, and fetch rules, as well as a DML command that executes queries. Deco’s overall architecture is shown in Figure 2.1.

Client applications interact with the Deco system using the Deco API, which implements the standard Python Database API v2.0: connecting to a database, executing a query, and fetching results. The Deco API also provides an interface for registering and configuring user defined fetch procedures and resolution functions. Using the Deco API, we built a command-line interface as well as a

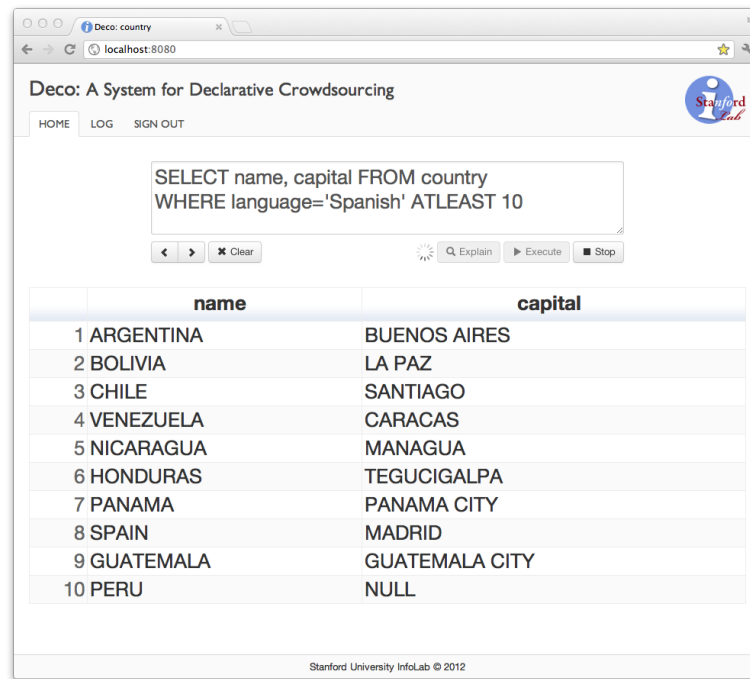


Figure 2.2: Deco user interface: query execution

web-based graphical user interface. The GUI executes queries (Figure 2.2), visualizes query plans (Figure 2.3), and shows log messages in real-time.

When the Deco API receives a query, at a very high level the overall process of parsing the query, choosing the best query plan, and executing the chosen plan is similar to a traditional database system. However, there are many significant differences in the details. The next two chapters describe details of query plan execution (Chapter 3), and how the system constructs and selects a plan (Chapter 4).

2.4 Related Work

This section discusses related work for Deco in general. More specific comparisons regarding Deco's query execution and optimization are given in Sections 3.7 and 4.6, respectively.

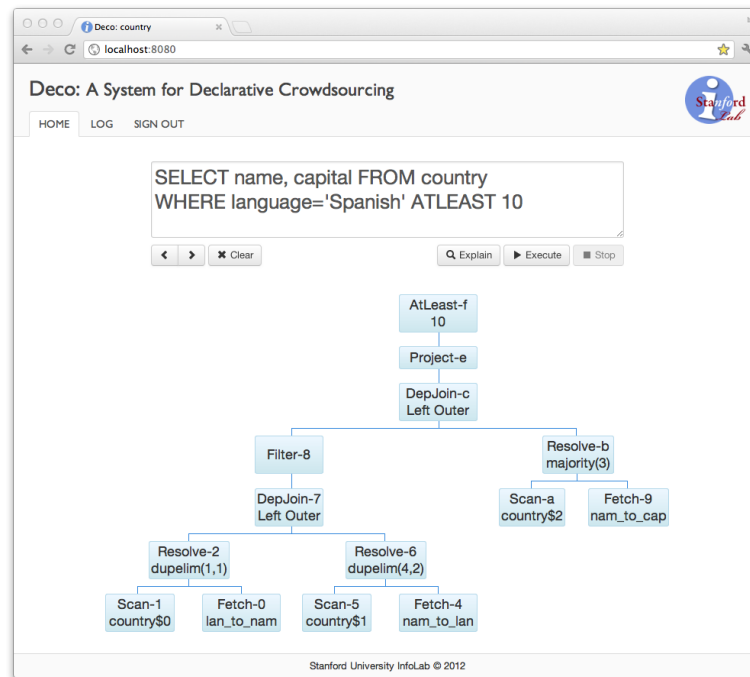


Figure 2.3: Deco user interface: query plan visualization

Several recent data-oriented systems have proposed a declarative approach to incorporate crowdsourced data [13, 20, 26, 36, 38]. Among those systems, CrowdDB [26] bears the closest similarity to Deco in terms of the data model and query language; however, Deco opts for more generality and flexibility, thus requiring the novel query processing techniques described in Chapters 3 and 4. (A detailed comparison between the two systems can be found in [41].) Quirk [38] is a workflow system that uses crowdsourcing primarily as part of its operations on existing structured data, and reference [38] studied how to reduce the monetary cost of its crowd-powered sort and join operators by improving worker interfaces. Since Deco makes no assumption about worker interfaces, their work is complimentary to Deco and can be incorporated into Deco to improve fetch procedures. To the best of our knowledge, CrowdDB and the other systems in this category do not yet have a cost-based query optimizer.

Chapter 3

Query Execution in Deco

We specified the Deco data model, query language, and system architecture in Chapter 2. In this chapter and the next one we consider how Deco’s query optimizer and query execution engine together answer a given Deco query. Our primary goal in query processing is to produce at least the required number of result tuples, while minimizing monetary cost. Our secondary goal is to reduce latency while maintaining the minimum cost. Note that Deco’s resolution functions effectively specify minimum thresholds for the third axis of interest, data quality. In our design of Deco, the query optimizer selects the plan with the least estimated monetary cost, without explicitly considering latency. Then, when the plan is executed by the execution engine, both cost and latency are considered.

In this chapter, we consider details of the query execution engine, specifically addressing how to implement the defined semantics while executing queries efficiently, assuming that the query optimizer has already picked a query execution plan.¹ We will describe how the query optimizer chooses query plans in Chapter 4. In Section 3.1, we discuss several unique challenges in Deco’s query execution engine, along with our approach to tackling them. In Section 3.2, we describe Deco’s query operators and execution plans. In Section 3.3, we provide details of query execution for the case where there are no existing tuples in any of the raw tables. In Section 3.4, we extend query execution for the general case where there may be existing tuples in raw tables. In Section 3.5, we describe prioritization of

¹The material presented in this chapter first appeared in [41, 44].

accesses to crowdsourced data. Section 3.6 describes our experimental evaluation, Section 3.7 covers related work, and we conclude in Section 3.8.

3.1 Challenges and Approach

Deco’s data model and query semantics along with the primary and secondary goals described above—minimizing cost, then reducing latency—give us several unique challenges to address. We motivate those challenges and how Deco’s execution engine addresses them. As in traditional database systems [27], a *query plan* is a rooted DAG of *query operators*; each operator corresponds to a data processing step and produces “output tuples” based on “input tuples” that are the outputs of its children.

3.1.1 Executing Queries in Two Phases

Consider a query with “MinTuples n ” constraint. To minimize monetary cost, Deco’s query execution engine must ensure that fetches are issued to the crowd only if the raw tables do not have sufficient data to produce n result tuples. To do so, Deco executes queries in two phases. In the *materialization* phase, the “current” result is materialized using the existing contents of the raw tables without invoking additional fetches. If this result does not meet the MinTuples constraint, the *accretion* phase invokes fetch rules to obtain more results. This second phase extends the result incrementally as fetch rules complete, and invokes more fetches as needed until the constraint is met.

3.1.2 Enabling Parallelism Using Asynchronous Pull

As a data source, the crowd has unique characteristics that are not found in conventional sources (like disks): very high latency and abundant parallelism. Thus, to reduce overall latency, it is important for Deco’s query execution engine to exploit parallelism when accessing the crowd. However, the traditional *iterator* model [27] for executing query plans is ill suited to this kind of parallelism, because *getNext* calls on query operators do not return until data is provided. (The

solution used in parallel database systems does not apply in our setting; see discussion in Section 3.7.) In Deco’s *hybrid execution model*, query operators do not expect an immediate response to a “pull” (*getNext*) request; the child operator will respond whenever a new output tuple becomes available. This built-in asynchrony in query execution ultimately allows Deco to ask multiple questions to the crowd in parallel, without having to wait for individual crowd answers. The concept of *asynchronous iteration* introduced in [28] also aims to enable parallelism when accessing outside sources during query execution. However, our specific setting led us to a somewhat different solution; see Section 3.7 for further comparison.

3.1.3 Choosing Right Degree of Parallelism

To reduce latency while still minimizing monetary cost, it is critical to choose the right degree of parallelism when accessing the crowd: too much parallelism might waste work, while too little increases latency. Our approach is to exploit as much parallelism as possible (to reduce latency), but only when the parallelism will not waste work (to minimize cost). Specifically we set the following two fundamental objectives for Deco’s query execution engine:

- **Objective #1:** Never parallelize accesses to the crowd if it might increase the monetary cost.
- **Objective #2:** Always parallelize accesses to the crowd if it cannot increase the monetary cost.

3.1.4 Initiating Good Fetches

In certain cases where existing data must be combined with new data to produce the result, minimizing monetary cost is especially difficult because existing data can make some fetches more “profitable” than others: choosing these fetches allows us to meet the *MinTuples* constraint with less cost. For example, it is better to invoke fetches that complete nearly-complete result tuples, or that may eventually contribute to many result tuples. Thus, Deco’s query execution engine has to choose not only the right degree of parallelism, but also the specific “good”

fetches to invoke in parallel. Individual query operators do not always have enough information to choose the good fetches, so our approach is to invoke more fetches than needed, but prioritize them so the better fetches are more likely to complete first (thus minimizing monetary cost). The query engine cancels any outstanding fetches once the MinTuples constraint is met, which for our model we assume incurs no extra cost. (Note this assumption largely holds in, e.g., Amazon Mechanical Turk [1].)

3.1.5 Changing Result Incrementally

Due to the flexibility of Deco's fetch rules and resolution rules, implementing the Deco semantics defined in Chapter 2 requires query operators to sometimes remove or modify intermediate output tuples that were passed to their parents previously. For instance, considering our example of Chapter 2, fetch rules such as $\text{country} \Rightarrow \text{language, capital}$ can provide tuples to multiple raw tables. Even if this fetch rule is invoked based on the need for a language value, data may as a side effect be inserted into the raw table for capital. This insertion can change the resolved capital value already produced because resolution functions are not necessarily monotonic. The process of propagating updates becomes similar to *incremental view maintenance* [12].

3.2 Query Operators and Execution Plans

In this section, we first describe Deco's query operators. Then we explain how those operators are assembled into query plans. Note the current Deco system processes Select-Project-Join queries only.

3.2.1 Query Operators

Deco includes standard Filter and Project operators that we do not describe further [27]. The following query operators are specific to Deco:

- Each Fetch operator corresponds to a fetch rule $A_1 \Rightarrow A_2 : P$. It invokes

procedure P using values for A_1 either received from its parent operator or bound based on the query. It does not wait for answers, so that many instances of the fetch procedure can be invoked in parallel. When values for $A_1 \cup A_2$ are returned by P , they are sent to one or more Scan operators that work with the Fetch operator. Scan operators insert the new tuples into raw tables and also pass them up to their parent Resolve operators. Note Fetch and Scan operators do not have any child operators, i.e., they are always leaves in the plan.

- Each Resolve operator corresponds to a resolution rule $A_1 \rightarrow A_2 : f$. It buffers all tuples in the raw table associated with its child Scan operator, grouped by their A_1 values. For each group, it maintains the result of function f on the set of A_2 values in the group. When f produces a non-empty set of resolved A_2 values, each of them is passed to the parent, along with the value for A_1 . When a new tuple is needed, the Resolve operator asks its child Fetch operator for the minimum number of additional input A_2 values needed for f to produce a resolved A_2 value for the group (recall Section 2.1.4).
- Each DepJoin (for dependent join [25, 28, 34]) operator is used to join conceptual relations in the From clause. It performs an inner join [27] based on equijoin predicates. Its behavior is similar to a relational indexed nested-loop join: It receives attribute values from its outer child, which it passes to its inner child to obtain additional attributes that constitute join result tuples.
- Each DLOJoin (for Dependent Left Outerjoin) operator is used to join raw tables constituting one conceptual relation. It performs a left outerjoin to implement Deco's data model semantics (recall Section 2.1.5): It always obtains anchor attributes from its outer child and dependent attributes from its inner, and produces a result even if there are no matching inner tuples. As in DepJoin its behavior is similar to a relational indexed nested-loop join.
- The MinTuples operator buffers the *partial result* of the query at the root of query plans: It contains the answer to the query computed so far, but may include tuples with NULL values. Based on the buffered partial result, the operator determines when the MinTuples constraint is satisfied.

We will see in the next subsection how query operators are assembled into a rooted DAG to make up a query plan.

3.2.2 Query Plans

In this section we introduce Deco’s query plans by explaining four possible query plans for an example query. Sections 3.3 and 3.4 will go into far more detail of plan execution. We continue with the example database from Chapter 2 (Section 2.1). Our example query finds eight Spanish-speaking countries and their capitals:

```
SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES 8
```

Basic Query Plan

Figure 3.1 shows a basic query plan for the example query. This plan uses three fetch rules: $\emptyset \Rightarrow$ country (operator 8), country \Rightarrow language (operator 11), and country \Rightarrow capital (operator 14). Recall from Section 2.1 that the raw schema includes three raw tables CountryA(country) (operator 7), CountryD1(country, language) (operator 10), and CountryD2(country, capital) (operator 13). Also recall that resolution functions for country, language, capital are *dupElim*, *majority-of-3*, and *majority-of-3*, respectively. Abbreviations in the query plan should be self-explanatory.

At a high level, the plan performs an outerjoin (operator 5) of a resolved version of CountryA (operator 6) and a resolved version of CountryD1 (operator 9), followed by a filter on language (operator 4). This result is outerjoined (operator 3) with a resolved version of CountryD2 (operator 12). Lastly, country and capital attributes are projected (operator 2).

For now let us assume there are no existing tuples in any of the raw tables. When we provide more details in later sections, we will consider the case of empty as well as non-empty raw tables. First, the root operator sends eight “pull” requests to its child operator (based on MinTuples). These requests propagate down the left (outer) side of the joins, and eventually invoke fetch rule $\emptyset \Rightarrow$ country

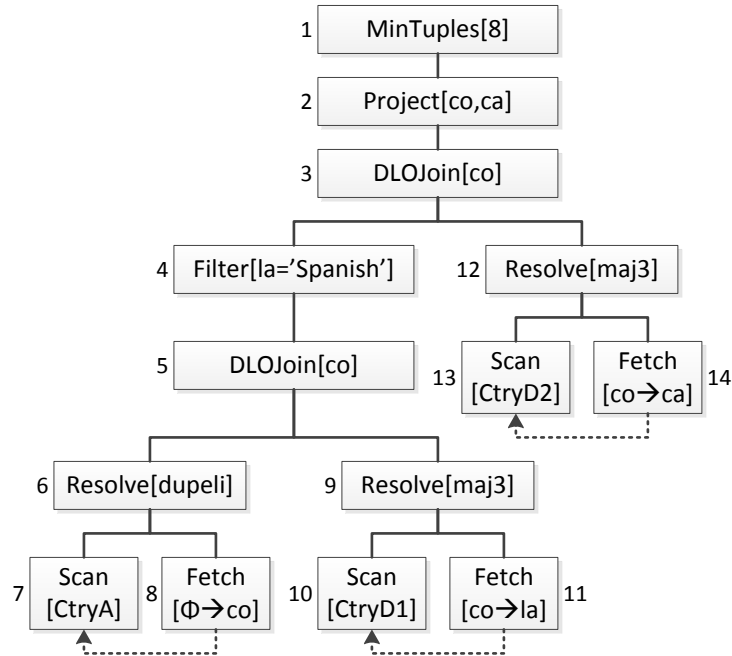


Figure 3.1: Basic query plan

eight times, without waiting for answers. At this point, there are eight outstanding fetches in parallel.

As these outstanding fetches complete, the new country values are inserted into raw table CountryA and passed up the plan by the Scan operator. Through the DLOJoin, new countries trigger invocations of fetch rule $\text{country} \Rightarrow \text{language}$. For each country value, two instances of this fetch rule are invoked in parallel because the resolution function *majority-of-3* requires at least two language values as input. At this point, we may have many fetches going on in parallel: some to fetch more countries, and some to fetch languages for given countries.

Until the MinTuples constraint is met, the query plan invokes additional fetches as needed. For example, if the two instances of fetch rule $\text{country} \Rightarrow \text{language}$ for a given country return two different language values, the plan invokes another instance of the same fetch rule to obtain the third language value. Likewise, as soon as a resolved language value for a certain country turns out to not be Spanish, the plan invokes a new instance of fetch rule $\emptyset \Rightarrow \text{country}$. For countries whose resolved language value is Spanish, the plan obtains capital values for the country,

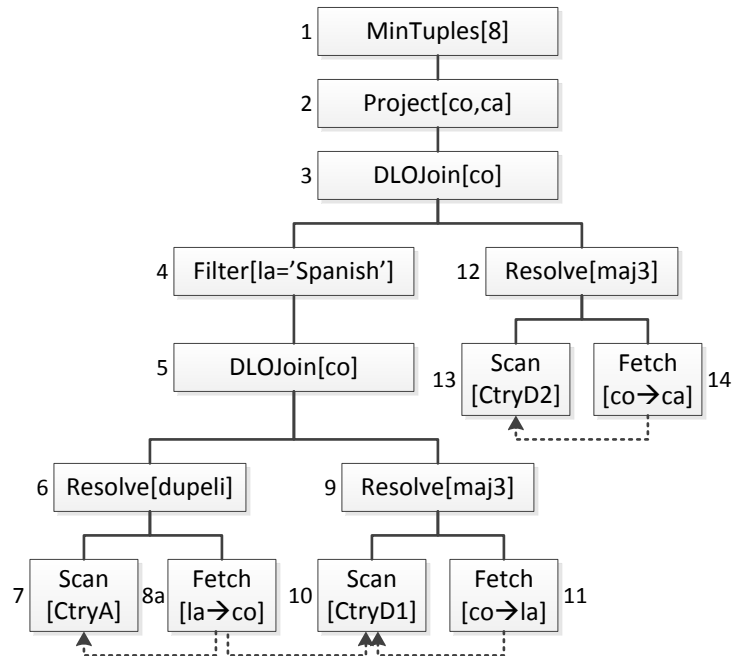


Figure 3.2: Reverse query plan

in parallel with other fetches similarly to how language values were obtained. Once the MinTuples constraint is met, the result tuples are returned to the client.

Reverse Query Plan

Suppose the predicate language='Spanish' is very selective, that is, most tuples have a different language value. If we use the basic query plan in Figure 3.1, even obtaining a single answer could be expensive in terms of monetary cost and latency, because we are likely to end up fetching many countries and languages that do not satisfy the predicate. Figure 3.2 shows an alternative query plan that uses the “reverse” fetch rule language \Rightarrow country underneath the Resolve operator. Note the only change from Figure 3.1 is operator 8a, so at a high level the behavior of the reverse plan is similar to the behavior of the basic plan in Figure 3.1. However, whenever Resolve operator 6 needs an additional country value, Fetch operator 8a invokes the fetch rule language \Rightarrow country with Spanish as the language value. (The association of predicate language='Spanish' with the

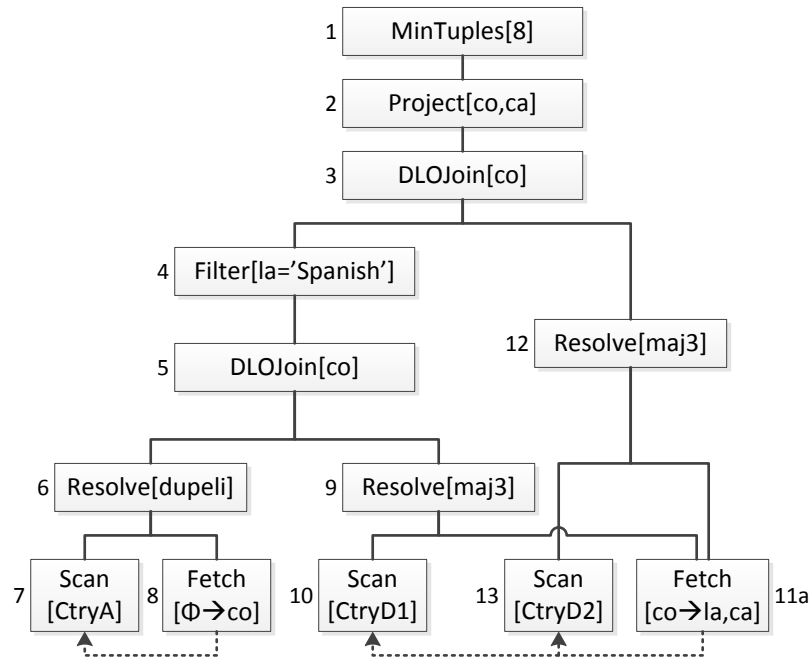


Figure 3.3: Combined query plan

Fetch operator is made when this query plan is built.) When completed, this fetch rule invocation obtains a country that is Spanish-speaking according to one worker, rather than a random country. Although the country may still turn out to be not Spanish-speaking after resolving its language value through additional fetches, it is more likely to pass the predicate, avoiding unnecessary fetches.

Combined Query Plan

It may be less expensive to use a fetch rule that gathers multiple dependent attributes at the same time, rather than fetching attributes separately. For the example query, we can use the “combined” fetch rule $\text{country} \Rightarrow \text{language, capital}$ instead of the two fetch rules $\text{country} \Rightarrow \text{language}$ and $\text{country} \Rightarrow \text{capital}$. Figure 3.3 shows a query plan that uses this approach. Note that Figure 3.3 differs from Figure 3.1 only in operator 11a and the absence of operator 14. Both Resolve operators 9 and 12 ask Fetch operator 11a when they need more tuples. (Fetch operators keep track of outstanding fetches to ensure that redundant fetches are

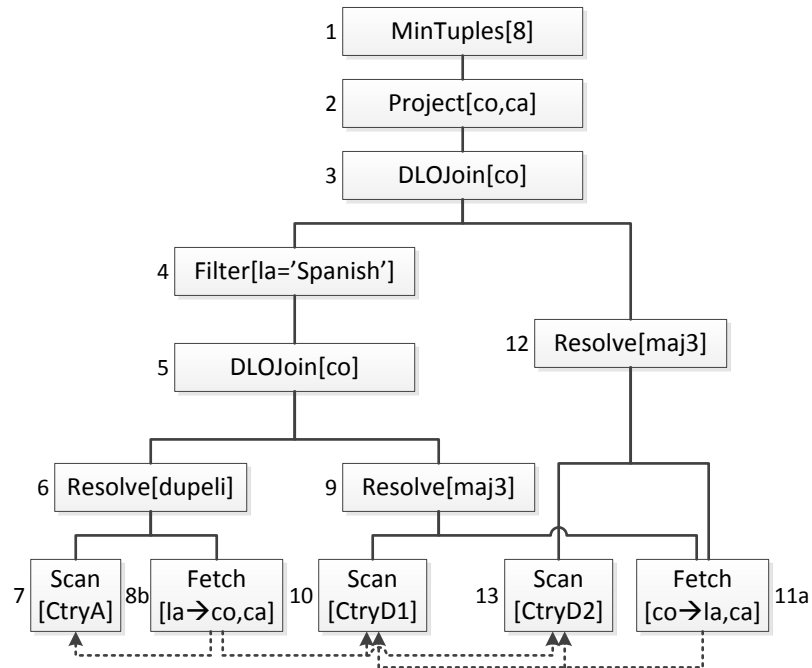


Figure 3.4: Hybrid query plan

not invoked.) As motivated briefly in Section 3.1.5, there are complexities involved when using combined fetch rules. We will see in Section 3.3.3 in detail how Deco’s execution engine handles this plan.

Hybrid Query Plan

Since the reverse and combined query plans described above improve upon the basic plan in two complimentary ways, we can take a “hybrid” approach where a single query plan contains a reverse fetch rule as well as a combined fetch rule. Figure 3.4 shows a query plan that uses this approach, employing two fetch rules $\text{language} \Rightarrow \text{country, capital}$ and $\text{country} \Rightarrow \text{language, capital}$. Note the only change from Figure 3.3 is operator 8b. In this query plan, each of the two fetch rules provides a pair of (country, language) values to raw table CountryD1 and a pair of (country, capital) values to raw table CountryD2. Given the resolution function *majority-of-3* for both language and capital, producing one result tuple only

requires one instance of each fetch rule to be completed with correct worker answers. We will walk through this scenario in more detail in Section 3.3.4. Note a query plan with fetch rules $\text{language} \Rightarrow \text{country}$ and $\text{country} \Rightarrow \text{language, capital}$ would have been slightly less efficient: Obtaining two pairs of (country, capital) values using fetch rule $\text{country} \Rightarrow \text{language, capital}$ produces three pairs of (country, language) values including a pair from fetch rule $\text{language} \Rightarrow \text{country}$, so the third language value may be wasted.

3.3 Query Execution with No Existing Data

In this section, we describe in detail how we execute a Deco query plan starting with empty raw tables. Under this assumption, the “current” query result computed in the materialization phase (recall Section 3.1.1) is always empty and thus does not satisfy the “MinTuples n ” constraint. Therefore, we focus on the accretion phase, which fetches new data to obtain n result tuples. Section 3.4 addresses the general case when there is existing raw data.

Query operators communicate with each other by exchanging three kinds of *messages*: bind messages pull more tuples from child operators, and add and remove messages push incremental changes to parent operators. Each query operator has a queue that stores messages received from other operators, and runs on its own thread that dequeues and processes one message at a time. Processing a message typically involves changing the operator’s local state as well as sending messages to its parent or child operators. For some operators, the local state includes a buffer that stores all input tuples to the operator. (Note DLOJoin and DepJoin have two local buffers, one for input tuples from the outer child and another for input tuples from the inner child.) These buffers are needed to determine the incremental changes to be propagated up the plan.

Since our operators must deal with asynchrony and handle both push and pull requests, their details differ significantly from their iterator model counterparts [27]. Table 3.1 summarizes the behavior of our operators for the case of initially empty raw tables. The entries in the tables will be explained in the next subsections.

Operator	On receiving bind[t] (from parent)	On receiving add[t] or remove[t] (from child)
MinTuples	–	For add[t], add t to buffer. For remove[t], remove t from buffer. If the MinTuples constraint is met, terminate execution and return all non-NULL tuples in the buffer to the client.
Project	Forward bind[t] to child.	Forward add/remove[$\Pi_A(t)$] to parent.
Filter	Forward bind[t] to child. (For our space of plans, $t=\emptyset$ always.)	For add[t], if $p(t)$ is true (where p denotes the filter's predicate), forward add[t] to parent; if $p(t)$ is false, send bind[\emptyset] to child. For remove[t], if $p(t)$ is true, forward remove[t] to parent.
DLOJoin	Forward bind[t] to outer child.	For add[t], add t to outer or inner buffer. For remove[t], remove t from outer or inner buffer. Propagate any changes to outerjoin result to parent using add/remove. For add[t], suppose t is from outer and has no matching tuple in inner buffer. If the join values $\Pi_A(t)$ are being seen for the first time, send bind[$\Pi_B(t)$] to inner child (where A denotes the joining anchor attributes from outer, and B denotes all anchor attributes from outer).
DepJoin	Forward bind[t] to outer child. (For our space of plans, $t=\emptyset$ always.)	If the join values in t contain one or more NULLs, do nothing and return. For add[t], add t to outer or inner buffer. For remove[t], remove t from outer or inner buffer. Propagate any changes to join result to parent using add/remove. For add[t], suppose t is from outer and has no matching tuple in inner buffer. If the join values $\Pi_A(t)$ are being seen for the first time, send bind[$\Pi_A(t)$] to inner child (where A denotes outer attributes in the join predicates). Otherwise, if bind[$\Pi_A(t)$] has failed to obtain matching inner tuples, send bind[\emptyset] to outer child. For add[t], if t is from inner and has no matching outer tuples, send bind[\emptyset] to outer child.
Resolve	Forward bind[t] to child Fetch operator.	Add $\Pi_{L \rightarrow R}(t)$ to buffer (where $L \rightarrow R$ denotes the resolution rule). Propagate any changes to resolution result to parent using add/remove. If the resolution function indicates that more input tuples are needed to produce an output, send bind[t] to child Fetch operator.

Operator	On receiving bind[t] (from parent)	On receiving add[t] or remove[t] (from child)
Scan	–	Insert $\Pi_A(t)$ into raw table (where A denotes attributes in the raw table), and send add[t] to parent.
Fetch	Invoke the fetch rule $L \Rightarrow R$ with argument $\Pi_L(t c)$ as left-hand side, where $ $ denotes tuple concatenation, and c is a tuple consisting of constant value v 's in all Where clause predicates of form $S.A=v$.	When the fetch completes, send add[t] to corresponding Scan operators. (See dotted arrows in Figures 3.1, 3.2, and 3.3.)

Table 3.1: Message exchanges during the accretion phase (empty raw tables)

3.3.1 Basic Query Plan

Consider the basic query plan in Figure 3.1, which we described at a high level in Section 3.2.2. We now explain in detail how Deco's engine executes the plan. We first explain how Deco invokes initial fetches. We then describe how Deco populates the result based on newly fetched data, and invokes more fetches as needed. Finally, we explain how Deco manages termination once the MinTuples constraint is met.

Initial Fetches: In every Deco plan, initially the MinTuples operator sends n bind[\emptyset] messages to its child operator, where n is the MinTuples constraint, and \emptyset denotes an empty tuple. When an operator receives a bind message, its job is to produce at least one output tuple for its parent. Under the assumption that query execution starts with empty raw tables, all operators except Fetch simply pass the responsibility (i.e., the bind message) on to a child (the outer child for DLOJoin/DepJoin and Fetch child for Resolve), and Fetch operators invoke their fetch rules to obtain tuples from the crowd.

In our example plan, MinTuples operator 1 initiates eight bind messages, and these bind messages propagate down to Fetch operator 8, which invokes eight instances of its fetch rule $\emptyset \Rightarrow \text{country}$ in parallel. At this point, there are eight outstanding fetches in parallel.

Populating Result: When an outstanding fetch completes, the Fetch operator sends an add message with the new tuple to its corresponding Scan operator. The Scan operator inserts the tuple into its raw table, then the change is propagated up the plan via add and remove messages, similar to incremental view maintenance. (In Sections 3.3.2 and 3.3.3 we will see that a single fetch can actually result in multiple changes.)

In addition to bind messages from the MinTuples root, some operators may initiate additional bind messages based on the tuples propagating up, in the following two cases:

- DLOJoin and DepJoin operators send bind messages to their inner children when they receive outer tuples that need to be matched.
- Filter and Resolve operators may initiate an additional bind message if an input tuple does not produce a new output tuple to propagate up.

We will discuss in Section 3.4.3 how the right degree of parallelism (based on the two parallelism objectives in Section 3.1.3) is maintained at all times through the additional bind messages.

Continuing our example, suppose an outstanding fetch $\emptyset \Rightarrow \text{country}$ completes with answer Peru. First, Fetch operator 8 sends an add message with tuple (Peru) to Scan operator 7. The Scan inserts the tuple into its raw table CountryA and forwards the add message to its parent. Then, Resolve operator 6 (duplicate elimination) forwards the add to its parent because its buffer contains no other (Peru) tuples. Upon receiving this add from the outer child, DLOJoin operator 5 sends add with NULL-padded tuple (Peru,NULL) to its parent because there are no matching tuples in its inner buffer. (Even though pushing padded tuples may seem unnecessary for this plan, in general all query operators must push all output tuples to their parents because they have no global view of the plan they belong to.) Filter operator 4 does not forward the padded tuple further because the tuple does not satisfy predicate `language='Spanish'`.

In addition to pushing new data up the plan, DLOJoin operator 5 sends a bind message with Peru to its inner child to obtain a language value for Peru. The job of Resolve operator 9 is to ultimately pass up an add message with tuple (Peru,X), where X is a resolved language value for Peru. It does so by sending two bind messages to Fetch operator 11, which invokes two instances of fetch rule $\text{Peru} \Rightarrow \text{language}$ in parallel. At this point, we have many fetches going on in parallel: some to fetch more countries, and some to fetch languages for given countries.

Now suppose both outstanding fetches $\text{Peru} \Rightarrow \text{language}$ complete with answer Spanish. For each completed fetch, Fetch operator 11 sends an add message with (Peru,Spanish) to Scan operator 10, which extends its raw table CountryD1 and forwards the add to the parent. Once Resolve operator 9 receives both adds with (Peru,Spanish), it sends add with (Peru,Spanish) to its parent.

Next, DLOJoin operator 5 modifies its output tuple (Peru,NULL) to (Peru,Spanish) using a remove and an add message. The new tuple passes through Filter operator 4 and reaches DLOJoin operator 3, which pushes a padded tuple (Peru,Spanish, NULL) up the plan. Also, DLOJoin operator 3 sends a bind message with Peru to its inner child. Once receiving an add message with (Peru,Y) from Resolve operator 12, the DLOJoin incorporates Y into the partial result by modifying its output tuple (Peru,Spanish,NULL) to (Peru,Spanish,Y).

As a final example, suppose an outstanding fetch $\emptyset \Rightarrow \text{country}$ completes with answer Korea. Then, similarly to how Peru is processed, two instances of $\text{Korea} \Rightarrow \text{language}$ are invoked in parallel. When both instances complete with the same answer Korean, DLOJoin operator 5 sends its parent a remove message with (Korea,NULL) and an add message with (Korea,Korean). At this point, Filter operator 4 finds out that the new input tuple does not pass its predicate, so it initiates a bind message to obtain another country-language pair. This bind message propagates down the plan and invokes another instance of fetch rule $\emptyset \Rightarrow \text{country}$. (Note that Filter operator 4 does not initiate bind messages for input tuples with NULL language values: an additional bind message is initiated only if the predicate is evaluated as “false” in a three-valued logic [27].)

Signaling Termination: Once the partial result at the root operator satisfies the MinTuples constraint, all query operators are terminated, and the operator buffers are cleaned up. We will see later that in the general case, outstanding fetches may need to be canceled, but in the special case of empty raw tables, there cannot be outstanding fetches. Finally, all non-NULL result tuples in the buffer at the MinTuples operator are returned to the client (through the Deco API, recall Figure 2.1), and query execution terminates.

3.3.2 Reverse Query Plan

Recall from Section 3.2.2 that the only change from the basic plan in Figure 3.1 to the reverse plan in Figure 3.2 is Fetch operator 8a. Thus we focus on execution details in the reverse plan that differ from the basic plan.

To start with, MinTuples operator 1 initiates eight bind messages as in the basic plan. When Fetch operator 8a receives the bind messages propagated down from the root operator, it invokes eight instances of the fetch rule language \Rightarrow country with Spanish as the language value. When one of the fetch rule invocations completes with a new country name Z, it adds tuples to both CountryA and CountryD1 via Scan operators 7 and 10, and the two added tuples propagate up the plan separately. Assuming the country name Z is not a duplicate, Resolve operator 6 pushes Z up to DLOJoin operator 5, which sends a bind message to its inner child to complete the language of Z. Meanwhile, Scan operator 10 pushes (Z,Spanish) to Resolve operator 9. Since the Resolve already has one raw language value for Z, it asks for one additional language value using the fetch rule country \Rightarrow language (Fetch operator 11) to compute a resolved language value for Z. (In contrast, the basic plan would have asked for two language values using the same fetch rule because the fetch rule $\emptyset \Rightarrow$ country does not provide a language value. Once one of the two country \Rightarrow language fetch rules invoked by the basic plan is answered as Spanish, its execution state is effectively identical to the reverse plan with the country Z just obtained.) The rest of plan execution proceeds as in the basic plan.

3.3.3 Combined Query Plan

We now consider the combined plan in Figure 3.3. One unique aspect of the combined plan is the fact that Fetch operator 11a has two parent operators: Resolve operators 9 and 12. Both of the Resolve operators send bind messages to Fetch operator 11a when they need more input tuples. Once an outstanding fetch for $\text{country} \Rightarrow \text{language, capital}$ completes, Fetch operator 11a sends add messages to both Scan operators 10 and 13. As in the reverse plan, those add messages propagate up the plan separately.

To illustrate a more interesting scenario, let us consider the query plan in Figure 3.3 but with *majority-of-5* as the resolution function for capital in Resolve operator 12 (with shortcutting, so three matching values are needed). Suppose an outstanding fetch $\emptyset \Rightarrow \text{country}$ completes with answer Bolivia. DLOJoin operator 5 initiates a bind message, and Fetch operator 11a invokes two instances of $\text{Bolivia} \Rightarrow \text{language, capital}$ in parallel. Suppose both fetches complete with the same answer (Spanish, La Paz). Then, a new output tuple (Bolivia, Spanish) reaches DLOJoin operator 3, and a partial result tuple (Bolivia, NULL) is stored at MinTuples operator 1. At the same time, another instance of $\text{Bolivia} \Rightarrow \text{language, capital}$ is invoked to obtain another capital value. Upon receiving (Quechua, Sucre) as the third answer, the same fetch rule is invoked again. Suppose the fourth answer is (Aymara, La Paz). While Resolve operator 12 finally produces La Paz as the resolved capital value, Resolve operator 9 no longer has a majority for the language of Bolivia. Thus it sends a remove message with (Bolivia, Spanish) to its parent, which propagates up and eventually invalidates the partial result tuple (Bolivia, NULL). More bind messages are initiated until resolved values for country and capital are sufficient to produce the query result.

3.3.4 Hybrid Query Plan

Finally, consider the hybrid plan in Figure 3.4. As in the other plans, this plan starts with eight instances of the fetch rule obtaining country names, which is $\text{Spanish} \Rightarrow \text{country, capital}$ (Fetch operator 8b) in this case. Suppose one of the outstanding fetches completes with answer (Argentina, Buenos Aires). Based on

this answer, tuples are added to all three raw tables CountryA, CountryD1, and CountryD2 via Scan operators 7, 10, and 13, respectively. As in the reverse plan, Resolve operator 9 asks for an additional language value using the fetch rule Argentina \Rightarrow language,capital (Fetch operator 11a). Resolve operator 12 also asks for an additional capital value to produce a resolved capital value for Argentina, but Fetch operator 11a recognizes the redundant outstanding fetch and does not invoke its fetch rule again. Now suppose fetch rule Argentina \Rightarrow language,capital completes with answer (Spanish,Buenos Aires). This time tuples are added to dependent tables CountryD1 and CountryD2 via Scan operators 10 and 13. Once these tuples propagate up the plan, resolved language and capital values for Argentina are produced, and the root operator eventually gets a result tuple (Argentina,Buenos Aires).

3.3.5 Join of Conceptual Relations

Although our example query from Section 3.2.2 contains only one conceptual relation Country in its From clause, Deco's query execution engine also supports joining conceptual relations using the DepJoin operator (recall Section 3.2.1). While DLOJoin handles outerjoins between anchor and dependent tables according to Deco's data model semantics, DepJoin handles join predicates explicitly specified in Where clauses. Both join operators are variants of dependent join, so they send bind messages with join values extracted from outer tuples to their inner children to receive matching inner tuples. However, DepJoin is somewhat more complex than DLOJoin. By definition, DLOJoin is always an equijoin over anchor attributes, whose values are fixed by the time a bind message is issued to the inner child, i.e., no additional fetches or resolution functions are applied for these anchor values. On the other hand, DepJoin predicates can be over any attributes (although restricted to equijoins; non-equijoins are handled as filters). When a DepJoin sends a bind message with join attribute values to the inner child, due to the behavior of resolution functions, it cannot be assured that the inner tuple returned will have matching join values. If the values do not match, additional bind messages must be sent to produce the needed join result tuple.

3.4 Query Execution with Existing Data

Now we describe how Deco queries are executed in the general case where there may be existing data in raw tables. As discussed in Section 3.1, this general case is significantly more difficult to handle than the special case of empty raw tables, in terms of minimizing monetary cost and reducing latency. We first describe how an initial query result is computed in the materialization phase, then how the accretion phase differs from the special case described in Section 3.3.

3.4.1 Materialization Phase

The materialization phase computes an initial query result based on the existing contents of the raw tables, and as a side effect populates operator buffers. Starting from Scan operators at the bottom, each operator in the plan pushes its initial output tuples to its parent using populate messages, then sends a shift message to the parent indicating the end of the initial output tuples at the operator (thus “shifting” to the accretion phase, if needed). Table 3.2 summarizes the behavior of operators for materialization. When the root MinTuples operator receives a shift message, its buffer contains the initial partial result. If the partial result satisfies the MinTuples constraint, execution terminates. Otherwise, execution continues to the accretion phase.

Although our current approach for computing the initial result is fairly naive, we could easily incorporate many conventional techniques into the materialization phase to improve performance. For example, we could cache the resolved versions of the raw tables across queries so that Scan operators do not have to send all raw tuples again for each query. In this case, the materialization phase starts from Resolve operators, and Scan operators do not participate. We could also incorporate a more traditional iterator model for the materialization phase, as long as it is properly extended to populate the operator buffers. Note that unless we have sufficient raw data, the bulk of query execution time is spent waiting for human answers, so latency improvement from speeding up the materialization phase is marginal.

As an example, consider again the basic query plan in Figure 3.1, and suppose

Operator	On receiving populate[t] (from child)	On receiving shift (from child)
MinTuples	Add t to buffer.	If the “MinTuples n ” constraint is met, terminate execution and return all non-NULL tuples in the buffer to the client. Otherwise, proceed to the accretion phase.
Project	Forward populate[$\Pi_A(t)$] to parent.	Forward shift to parent.
Filter	If $p(t)$ is true (where p denotes the filter’s predicate), forward populate[t] to parent.	Forward shift to parent.
DLOJoin	Add t to outer or inner buffer.	Send the left outerjoin of the two buffers to parent using populate messages. Then send shift to parent.
DepJoin	If all join attribute values are non-NULL, add t to outer or inner buffer.	Send the join of the two buffers to parent using populate messages. Then send shift to parent.
Resolve	Add t to buffer.	Group all tuples in buffer by the left-hand side attributes of the resolution rule, apply the resolution function for each group, and send the resolved tuples to parent using pop- ulate messages. Then send shift to parent.
Scan	At the start of the materialization phase, for each t in the raw table, send populate[t] to parent. Then send shift to parent.	

Table 3.2: Message exchanges for materialization

we have the following initial contents for the raw tables:

CountryA	CountryD1	CountryD2
country	country	country
Chile	Chile	capital
Italy	Chile	Italy
Korea	Chile	Rome
Peru	Italy	Korea
Spain	Italy	Seoul
Spain	Italy	Spain
	Peru	Madrid
	Spain	Spain
	Spain	Barcelona
	Spain	Spain
	Spain	Madrid
	Spain	Madrid

Recall the resolution function for both language and capital is *majority-of-3*, with

shortcutting. During the materialization phase, DLOJoin operator 5 accumulates the following input tuples in its buffers:

Outer buffer	Inner buffer	
country	country	language
Chile	Chile	Spanish
Italy	Italy	Italian
Korea	Spain	Spanish
Peru		
Spain		

Among the five output tuples of the DLOJoin, three of them do not pass Filter operator 4: (Italy,Italian), (Korea,NULL), and (Peru,NULL). Thus, DLOJoin operator 3 accumulates the following input tuples in its buffers:

Outer buffer		Inner buffer	
country	language	country	capital
Chile	Spanish	Spain	Madrid
Spain	Spanish		

The initial partial result at MinTuples operator 1 contains (Chile,NULL) and (Spain, Madrid).

3.4.2 Accretion Phase

Similarly to how the special case of empty raw tables proceeds (Section 3.3), the accretion phase first invokes some initial fetches to get started. Then, it extends result based on newly fetched data, possibly invokes more fetches as necessary, and finally signals termination once the MinTuples constraint is met.

Initial Fetches: Table 3.3 shows initial message exchanges that start the accretion phase, in the general case of non-empty raw tables. As seen in the table, initial bind messages are generated for two distinct purposes:

- **Gathering Completely New Tuples:** The MinTuples operator initiates k bind messages, where k is the minimum number of additional partial result tuples needed to satisfy the MinTuples constraint (based on the partial result after materialization).

Operator	At the start of accretion phase
MinTuples	Send $\max(n-m, 0)$ bind[\emptyset] messages to the child, where n is the MinTuples constraint, and m is the number of tuples in the buffer.
DLOJoin	For each set of values $\Pi_A(t)$ in outer buffer with no matching inner tuples, send bind[$\Pi_B(t)$] to inner child (where A denotes the joining anchor attributes from outer, and B denotes all anchor attributes from outer).
DepJoin	For each set of values $\Pi_A(t)$ in outer buffer with no matching inner tuples, send bind[$\Pi_A(t)$] to inner child (where A denotes outer attributes in the join predicates).

Table 3.3: Message exchanges to start accretion phase

- Joining Existing Tuples with New Tuples:** The other method of increasing the size of the result is to attempt to join existing outer tuples with new inner tuples. (For DLOJoin, this means replacing NULL values with actual values.) The bind messages to do so are generated by the DLOJoin and DepJoin operators. Each DLOJoin or DepJoin operator sends a bind message to its inner child for each join attribute values in its outer buffer with no matching tuples in its inner buffer.

Note that in the special case of empty raw tables, Table 3.3 says for MinTuples to initiate n binds, and none from joins, consistent with Section 3.3.

For most operators, behavior in the general case is the same as for the special case (Table 3.1). However, for Filter and DepJoin, the behavior upon receiving a bind message differs from the special case: These operators might be able to produce an output tuple without passing the bind message on to their child because of existing data. As mentioned above, DLOJoin operators initiate bind messages to replace NULL values with actual values. If a NULL is replaced with a value that satisfies the predicate of a Filter (or DepJoin) higher up in the plan, the Filter (or DepJoin) produces an output tuple spontaneously. In Section 3.4.3, we will further argue that Filter and DepJoin operators propagate bind messages down the plan in a manner that satisfies the two parallelism objectives from Section 3.1.3.

Continuing our example from the end of Section 3.4.1, the accretion phase proceeds as follows:

- MinTuples operator 1 needs at least eight result tuples, but its buffer contains only two partial result tuples. To obtain six more partial result tuples, it sends six bind messages to its child. Among these six binds, Filter operator 4 stops two binds because two input tuples may pass the predicate once their actual language values replace NULLs: (Korea,NULL) and (Peru,NULL). Eventually, four bind messages reach Fetch operator 8 and invoke fetch rule $\emptyset \Rightarrow \text{country}$ four times.
- DLOJoin operator 3 scans its outer buffer and finds one tuple (Chile,Spanish) that does not match any inner tuples. Thus, it sends a bind message to its inner child to complete the capital value of Chile. This message reaches Fetch operator 14, which invokes two instances of $\text{Chile} \Rightarrow \text{capital}$ in parallel.
- Similarly, DLOJoin operator 5 sends two bind messages to its inner child to complete the language values for Korea and Peru. Fetch operator 11 invokes two instances of $\text{Korea} \Rightarrow \text{language}$ and one instance of $\text{Peru} \Rightarrow \text{language}$.

Once these initial fetches are invoked, the query execution engine handles each completed fetch similarly to the examples from Section 3.3, possibly invoking additional fetches until eight result tuples are produced.

3.4.3 Meeting Parallelism Objectives

Now we discuss in detail how Deco's query execution meets the two parallelism objectives laid out in Section 3.1.3:

- Objective #1: Never parallelize accesses to the crowd if it might increase the monetary cost.
- Objective #2: Always parallelize accesses to the crowd if it cannot increase the monetary cost.

To meet the objectives, Deco's execution engine should only invoke fetches that are definitely needed to satisfy the MinTuples constraint, and should always invoke those fetches as early as possible. To do so, each query operator sends bind messages to its child as soon as it is known that the required number of output tuples cannot be produced without those additional bind messages. We consider each operator:

- The MinTuples operator initiates the minimum number of bind messages needed to satisfy the MinTuples constraint at the start of the accretion phase.
- Each Project operator simply forwards received bind messages to its child operator.
- Each Filter operator keeps track in local variables of how many bind messages have been received from its parent and how many tuples might be generated from below (as NULL values are replaced with actual values, or due to bind messages sent). It sends bind messages based on the difference between these quantities.
- Each DepJoin operator also keeps track of how many bind messages have been received from its parent and how many tuples might be generated from below, taking into account its buffers of tuples received from its outer and inner children. Similarly to Filter, it sends bind messages to its outer child based on the difference between these quantities. We will discuss momentarily bind messages sent from a DepJoin to its inner child.
- Each DLOJoin operator forwards bind messages received from its parent to its outer child operator. Again we will discuss momentarily bind messages sent from a DLOJoin to its inner child.
- Each Resolve operator buffers all input tuples grouped by the values corresponding to the left-hand side of its resolution rule. It essentially treats those groups separately, keeping track of the number of existing raw tuples and the number of raw tuples being fetched, per group. It first initiates one or more bind messages to obtain the minimum number of input values needed for its resolution function to produce an output; additional bind messages are sent if the resolution function indicates more input values are needed.
- Each Fetch operator keeps track of all outstanding fetches it has invoked along with the parent Resolve operators that initiated those fetches, so that redundant fetches across different parents can be coalesced.

Note Scan operators neither receive nor send bind messages.

Finally let us consider bind messages sent from DLOJoin and DepJoin operators to their inner children. For this case, it may appear that DLOJoin and DepJoin

clearly violate Objective #1, since the number of binds they issue depends only on the number of non-joining tuples (which could be very large), and not on the “MinTuples n ” constraint (Tables 3.1 and 3.3). Our approach to this specific case relies on two features:

- *prioritizing* outstanding fetches, so the ones most likely produce useful data are returned first
- *canceling* outstanding fetches (without incurring cost) once sufficient data has been obtained

(Both of these capabilities are dependent on the crowdsourcing platform; we discuss Amazon Mechanical Turk specifically in Section 3.5.4.) Given the limited information at each DLOJoin and DepJoin operator, it is impossible for those operators to choose exactly the right number of bind messages to maximize parallelism while minimizing cost. Thus, we allow DLOJoin and DepJoin to issue bind messages to their inner children for *all* non-matching outer tuples, but we carefully prioritize the resulting fetches, and cancel any that are outstanding once the MinTuples constraint is met. The prioritization problem turns out to be quite interesting in our setting, so it is discussed in detail in the next section.

3.5 Fetch Prioritization

We now explain how we prioritize fetching data from the crowd, in order to minimize cost. Recall that all fetches are the result of bind messages passed down the plan until they reach Fetch operators. After a fetch rule has been invoked and before it returns, we refer to that instance as an “outstanding fetch.” As discussed in Sections 3.3 and 3.4, at any given time there may be a large number of outstanding fetches executing in parallel; our goal is to influence the order in which these outstanding fetches are handled, so the more “profitable” (to be defined) ones are likely to execute first. To separate our optimization goal from the details in crowdsourcing platforms, we assign *scores* to each outstanding fetch, reflecting our prioritization. How the scores result in actual fetch prioritization is platform-specific.

Based on our discussion in the previous section, all outstanding fetches have one of two purposes:

- (1) To create new partial result tuples. These fetches may be a result of bind messages originating from `MinTuples`, `DepJoin`, or `DLOJoin` operators.
- (2) To fill in `NULL` values in the partial result. These fetches are always a result of binds originating from `DLOJoin` operators.

For a “`MinTuples n`” constraint, our overall prioritization strategy is to first create at least n partial result tuples (purpose 1), then fill in `NULL` values (purpose 2) until we have n result tuples. Although this strategy may not be optimal in all cases, intuitively we must have n partial result tuples regardless in order to produce a query result. Thus, we focus our optimization on filling in `NULL` values in a fashion that minimizes cost. As future work, we could consider the more difficult problem of globally optimizing purposes 1 and 2 together, which will undoubtedly depend in complex ways on the query plan itself.

To find the optimal set of fetches that can fill in enough `NULL` values to generate the query result, there are several important factors to consider:

- Our goal is to generate tuples with no `NULL` values, so we may prefer to fill in `NULL` values for tuples that are “nearly” complete.
- Sometimes a single fetch can fill in multiple `NULL` values.
- Since resolution rules often require multiple input values in order to produce one output value, multiple fetches may be needed to fill in one `NULL` value.

We first formalize our fetch prioritization problem and show that it is NP-hard. We then propose two heuristic scoring functions and describe their implementation in the context of Deco’s query execution engine. Lastly we describe how fetch prioritization is implemented in one crowdsourcing platform, Amazon Mechanical Turk.

3.5.1 Formal Problem Definition

Suppose we have m partial result tuples for a query plan with “`MinTuples n`”. We are interested in the case where $m \geq n$, but fewer than n tuples are non-`NULL`. The query processor creates a set of outstanding fetches, each one associated with

one or more NULLs in the partial result. (For formalizing the problem, we assume resolution functions require a fixed number of input tuples, however we will see that our heuristics allow us to drop this assumption.) Our goal is to select a subset of the outstanding fetches that completes the result at minimum cost.

Finding the optimal solution for our fetch prioritization problem turns out to be NP-hard in the data size and the number of desired result tuples. (A proof by another author using a reduction from the three-dimensional matching problem is given in [44].) Thus, we use a heuristic approach, described in Section 3.5.2. Here we formulate the fetch prioritization problem as a *polynomial zero-one program* [33] to motivate our heuristics. (In fact, the fetch prioritization problem can be posed as an integer linear program, making it NP-complete; however, we use the polynomial zero-one program for ease of exposition.)

Let $\text{NULL}_{i1}, \dots, \text{NULL}_{ip_i}$ denote the p_i NULL values of the i -th partial result tuple ($p_i \geq 0$). To fill in NULL_{ij} , $q(i, j)$ identical outstanding fetches $f_{g(i,j)1}, \dots, f_{g(i,j)q(i,j)}$ must complete. (Note that an outstanding fetch f_{hk} contributes to all NULL_{ij} 's with $g(i, j) = h$ and $q(i, j) \geq k$.) Finally, let x_{hk} be an indicator variable specifying if fetch f_{hk} is chosen. Then, the fetch prioritization problem is directly translated to the following:

$$\begin{aligned} \text{minimize } \sum_h \sum_k x_{hk} \quad \text{subject to } & \sum_{i=1}^m \prod_{j=1}^{p_i} \prod_{k=1}^{q(i,j)} x_{g(i,j)k} \geq n \\ & x_{hk} = 0, 1 \quad \forall h, k \end{aligned}$$

The objective function is the number of chosen fetches. The inequality constraint says that at least n tuples have to be completed.

Example: Consider the following raw tables for the Country and City relations from the example introduced in Section 2.1. Recall that the City relation contains two anchor attributes city and country, and one dependent attribute population. Also recall that the resolution functions for the language, capital, and population attributes are *majority-of-3*, *majority-of-3*, and *average-of-2*, respectively.

city	country
Istanbul	Turkey
Venice	Italy
Trento	Italy

city	country	population
Venice	Italy	270660

country
Turkey
Italy

country	language
Italy	Italian

country	capital
Turkey	Istanbul

Suppose we are processing a query that asks for any two cities along with their population and language:

```

SELECT city, country, population, language
FROM City, Country
WHERE City.country = Country.country
MINTUPLES 2

```

Further suppose that we use a query plan with the following five fetch rules. (Note for each resolution rule $A_1 \rightarrow A_2$ for City and Country, there is a corresponding fetch rule $A_1 \Rightarrow A_2$, as in the basic query plan in Figure 3.1.)

- [Country] $\emptyset \Rightarrow$ country
- [Country] country \Rightarrow language
- [Country] country \Rightarrow capital
- [City] $\emptyset \Rightarrow$ city,country
- [City] city,country \Rightarrow population

Based on the current contents of the raw tables above, we get the following partial result at the end of the materialization phase:

city	country	population	language
Istanbul	Turkey	NULL	NULL
Venice	Italy	NULL	NULL
Trento	Italy	NULL	NULL

At the start of the accretion phase (recall Section 3.4.2), the bind messages initiated by DLOJoin operators to fill in NULL values propagate down the plan and invoke the following ten fetches:

- f_{11}, f_{12} : (Istanbul,Turkey) \Rightarrow population
- f_{21} : (Venice,Italy) \Rightarrow population
- f_{31}, f_{32} : (Trento,Italy) \Rightarrow population
- f_{41}, f_{42}, f_{43} : Turkey \Rightarrow language
- f_{51}, f_{52} : Italy \Rightarrow language

Note that the MinTuples and DepJoin operators do not initiate any bind messages based on Table 3.3.

Formulating this case as a polynomial zero-one program, we get:

$$\begin{aligned} &\text{minimize } x_{11} + x_{12} + x_{21} + x_{31} + x_{32} + x_{41} + x_{42} + x_{43} + x_{51} + x_{52} \\ &\text{subject to } x_{11}x_{12}x_{41}x_{42}x_{43} + x_{21}x_{51}x_{52} + x_{31}x_{32}x_{51}x_{52} \geq 2 \\ &\quad x_{hk} = 0, 1 \quad \forall h, k \end{aligned}$$

Recall the objective is to minimize the number of chosen fetches out of all f_{hk} 's. Each term in the constraint indicates if its corresponding partial result tuple is completed; for example, $x_{21}x_{51}x_{52}$ corresponds to the second partial result tuple for (Venice,Italy). The optimal solution is

$$x_{21}=x_{31}=x_{32}=x_{51}=x_{52}=1, \quad x_{11}=x_{12}=x_{41}=x_{42}=x_{43}=0$$

which translates to actual outstanding fetches (Venice,Italy) \Rightarrow population, (Trento, Italy) \Rightarrow population, and Italy \Rightarrow language.

Note that the number of outstanding fetches $q(i, j)$ required to fill in a NULL value may not be fixed as in our formulation, depending on the resolution function. Even in our simple running example, *majority-of-3* can stop after obtaining two raw values if they agree with each other. We will see in the next section that our heuristic approach accommodates variable values for $q(i, j)$.

3.5.2 Heuristic Algorithm

Our practical solution to the NP-hard prioritization problem is based on assigning scores to outstanding fetches. The scores can then be used by the crowdsourcing platform to influence which fetches are most likely to complete (Section 3.5.4). Initial scores are assigned to outstanding fetches at the start of the accretion phase based on a heuristic solution to the problem formalized in Section 3.5.1. As fetches complete and the partial result changes, scores for the remaining outstanding fetches may be adjusted. We will see in Section 3.6 that performance with our online heuristic approach can be close to optimal. We present two scoring functions. The first one assumes that only one fetch is needed to complete each NULL value, i.e., $q(i, j) = 1$. The second scoring function takes the resolution function into account and therefore multiple fetches may be needed, i.e., $q(i, j) \geq 1$.

Our first scoring function, $score_1$, is expressed in terms of p_i , the number of remaining NULL values in each partial result tuple. Specifically, we regard the contribution of f_{hk} to the i -th partial result tuple as $1/p_i$ (if f_{hk} contributes to the tuple at all):

$$score_1(f_{hk}) = \sum_{i=1}^m ([\exists j : g(i, j) = h] \times \frac{1}{p_i})$$

In our example in Section 3.5.1, we have $p_i=2$ for $i=1, 2, 3$, so $score_1(f_{hk})$ depends on the number of partial result tuples to which f_{hk} contributes. Thus, we have $score_1(f_{hk})=1/2$ for $h=1, 2, 3, 4$ and $score_1(f_{5k})=1$.

Our second scoring function, $score_2$, is expressed in terms of the number of remaining fetches to complete each partial result tuple, $\sum_{j=1}^{p_i} q(i, j)$. The contribution of f_{hk} to the i -th partial result tuple is $1/\sum_{j=1}^{p_i} q(i, j)$:

$$score_2(f_{hk}) = \sum_{i=1}^m ([\exists j : g(i, j) = h] \times \frac{1}{\sum_{j=1}^{p_i} q(i, j)})$$

In our example in Section 3.5.1, we have $\sum_{j=1}^{p_i} q(i, j)=5, 3, 4$ for $i=1, 2, 3$, respectively. Since each of f_{1k}, \dots, f_{4k} contributes to a single partial result tuple, we get $score_2(f_{1k})=score_2(f_{4k})=1/5$, $score_2(f_{2k})=1/3$, and $score_2(f_{3k})=1/4$. Since f_{5k}

contributes to two partial result tuples, we get $score_2(f_{5k})=1/3+1/4$. Notice the top-5 initial $score_2$ values correspond to the optimal solution.

3.5.3 Query Execution Engine Extension

We now explain how we have extended Deco's query execution engine to compute scoring function $score_1$ or $score_2$ for each outstanding fetch. Actual prioritization of fetches based on the scores depends on fetch procedures and crowd interfaces. For Amazon Mechanical Turk, our fetch procedure always presents the outstanding fetch with the highest score whenever a worker arrives (see Section 3.5.4).

Since $score_1$ relies on the entire set of current partial result tuples, the MinTuples operator is responsible for computing it. However, computing $score_1$ incurs little overhead due to the nature of the execution model: the MinTuples operator can incrementally maintain the scores as partial result tuples are updated by add and remove messages from its child operator. Specifically, when a partial result tuple t is added or removed, the MinTuples operator can adjust $score_1(f)$ for each outstanding fetch f that can fill in NULL values in the tuple t .

Our approach requires us to track all outstanding fetches that can fill in a NULL value in a given partial result tuple t . To do so, we assign a unique identifier for each outstanding fetch f and embed the identifier in the NULL values that f can fill in. DLOJoin and Fetch operators cooperate to perform this embedding.

For $score_2$, we can use the same overall approach as for $score_1$ except we need to manage q , the number of remaining fetches to complete each NULL value. Resolve operators calculate q based on the numbers of required input tuples for the resolution function, and the tuples currently in its buffer. Resolve operators must propagate calculated q values up to the MinTuples operator so that $score_2$ can reflect up-to-date q values. In this case, simply embedding q in its corresponding NULL value is not enough: Whenever q changes, Resolve operators must modify the q part of the NULL value using a remove plus an add message. When these messages reach the MinTuples operator, the $score_2$ values can be updated without affecting the partial result. Due to the additional computation and messages, computing $score_2$ is somewhat more expensive than computing $score_1$; however,

we will see in Section 3.6 that $score_2$ is a better heuristic.

3.5.4 Amazon Mechanical Turk Support

We now describe how Deco’s generic Amazon Mechanical Turk (MTurk) fetch procedure supports fetch prioritization based on scoring as described in the previous subsection. The MTurk fetch procedure translates fetch rule invocations from the query execution engine into HITs (Human Intelligence Tasks, unit tasks on MTurk) so that workers can answer them. For prioritization, the goal of the MTurk fetch procedure is to present the outstanding fetch with the highest score whenever a worker “accepts” one of the HITs. To do so, the MTurk fetch procedure works together with a dedicated *Deco form server*, which hosts HTML forms that the workers fill in, and keeps track of scores for outstanding fetches. (In Deco’s architecture shown in Figure 2.1, the MTurk component contains the form server and the MTurk service side-by-side: both the MTurk fetch procedure and the workers communicate with the MTurk service as well as the form server.)

More specifically, when the MTurk fetch procedure is invoked by a Fetch operator, the procedure instantiates a form template for the fetch rule, sends the HTML form and its score to the form server using a web service API, and creates a HIT that points to the form server. The MTurk fetch procedure also forwards any changes in scores for outstanding fetches from the query execution engine to the form server. When a worker accepts any of the HITs, the worker’s web browser loads the address of the form server, which then chooses the HTML form with the highest score and serves it to the worker. Typically MTurk workers accept a series of HITs within their chosen HIT type, presented by MTurk in random order. Overall, our approach replaces random scheduling with score-based prioritization using our form server to delay the binding of HTML forms to HITs.

3.6 Experimental Evaluation

In this section we present our experimental evaluation of Deco’s query execution engine. We first present an experiment comparing performance of alternative query plans with different fetch rules, using Amazon Mechanical Turk. Then we evaluate the Deco execution model focusing on the interactions among parallelism, cost, and latency. Finally, we evaluate the effectiveness of our fetch prioritization scheme, compared against optimal prioritization and random prioritization.

3.6.1 Performance of Different Query Plans

Experiment 1: Varying Fetch Rule Configurations: In our first experiment, we evaluated how different fetch rule configurations affect query performance, to study the usefulness of the flexibility of Deco fetch rules. We used our example query from Section 3.2.2 that finds eight Spanish-speaking countries and their capitals:

```
SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES 8
```

We report performance of three example query plans described in Section 3.2.2: the basic plan (Figure 3.1), the reverse plan (Figure 3.2), and the hybrid plan (Figure 3.4). Recall that these query plans employ the following fetch rules:

- Basic plan: $\emptyset \Rightarrow$ country, country \Rightarrow language, and country \Rightarrow capital
- Reverse plan: language \Rightarrow country, country \Rightarrow language, and country \Rightarrow capital
- Hybrid plan: language \Rightarrow country, capital and country \Rightarrow language, capital

We ended up excluding the combined plan (Figure 3.3) from this experiment. As we will see shortly from the performance of the basic plan, the fetch rule $\emptyset \Rightarrow$ country was not effective in obtaining Spanish-speaking countries, causing very poor performance. Since the basic and combined plans obtain Spanish-speaking countries in the same way, the cost of the combined plan would have

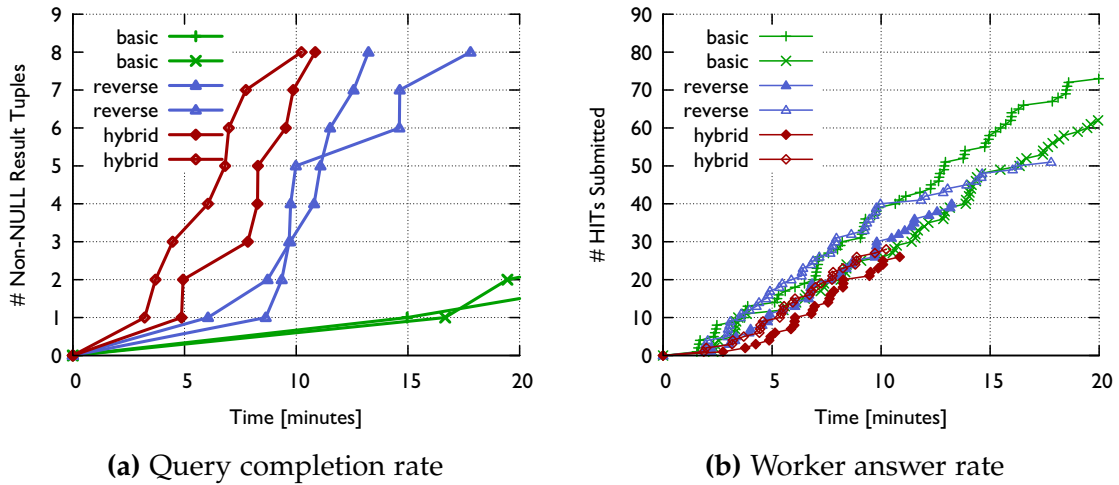


Figure 3.5: Performance of different query plans

been very similar to the cost of the basic plan: although the combined plan may avoid additional fetches for capital values, the effect is negligible against the overwhelming cost of fetching countries.

On Amazon Mechanical Turk, we paid a fixed amount of five cents per HIT (i.e., per fetch rule invocation) as compensation for completed work. For attributes country and capital, workers were allowed to enter free text, but our form server validated this input to ensure that the text had no leading or trailing whitespaces and that all letters were uppercase. For language, workers were allowed to select one language from a drop-down list of about 90 languages. All experiments were conducted on weekends in February 2012.

For each query plan, we ran the query twice starting with empty raw tables, and noted similar results. Both runs are included in our graphs. Figure 3.5a depicts the number of non-NULL result tuples obtained over time. Since our query specified `MinTuples 8`, reaching eight output tuples signifies the completion of the query. In addition, Figure 3.5b shows the number of HITs submitted by workers over time. Note that the monetary cost of a query is proportional to the total number of HITs submitted.

Using the hybrid plan, the query took 10.5 minutes and cost \$1.35 for 27 worker answers on average (across two runs). Using the reverse plan, the query

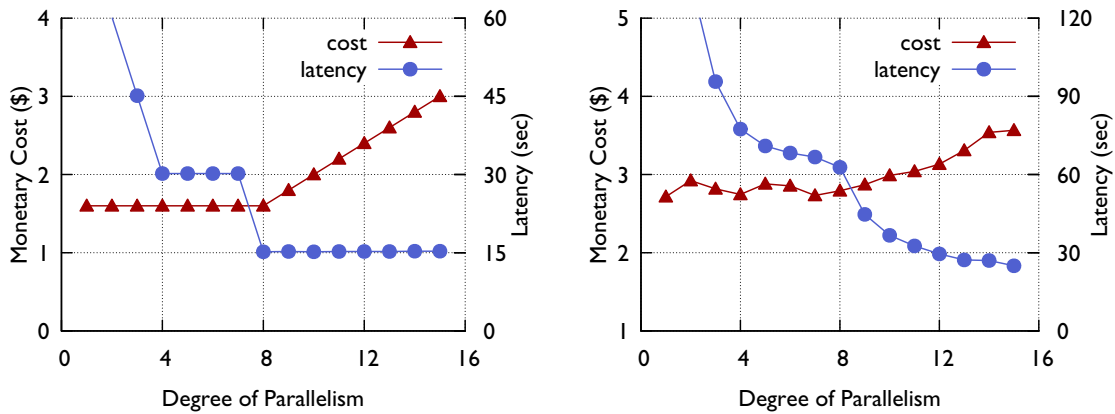
took 15 minutes and cost \$2.30 for 46 worker answers on average. In comparison, the basic plan performed very poorly: the query took two hours overall and cost around \$12.05. (We ended up collecting 64 countries and their languages.) Thus, we find that it is important to consider a variety of query plans, and the decisions of the query optimizer (Chapter 4) can significantly impact query performance in terms of latency and total monetary cost. In terms of quality, the results were clean and correct except one typo that persisted (“Ddominican Republic”). Not surprisingly, individual worker answers had several errors that were not exposed in the result.

3.6.2 Parallelism, Cost, and Latency

For the next experiment we report, and two additional experiments in Section 3.6.3, we built a *crowd simulator*, which enables a large number of experiments without significant latency and dollar cost. (Note that repeating these experiments on a real crowdsourcing platform would either be extremely costly, or would mean far fewer trials.) The simulator responds to fetch requests by selecting values from a predetermined set; we can either set our simulator to always give “correct” answers (e.g., it always returns Lima as the capital of Peru), or specify a fixed probability for each fetch rule that incorrect answers are given.

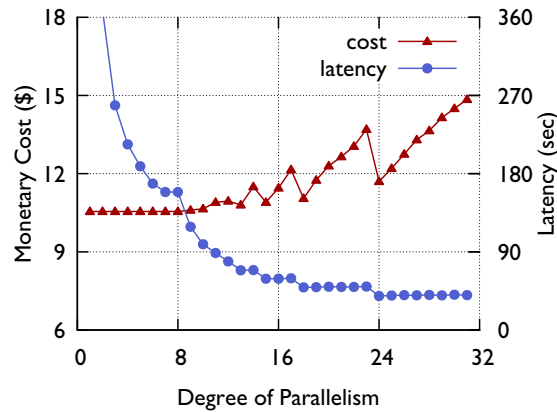
Experiment 2: Varying Degrees of Parallelism Our second experiment explores whether Deco’s query execution engine accomplishes its optimization goal of minimizing monetary cost and reducing latency when executing a given query plan, by choosing the right degree of parallelism based on the two objectives in Section 3.1.3. To do so, we compared monetary cost and latency of executing the same query plan with varying degrees of parallelism, using the basic (Figure 3.1) and reverse (Figure 3.2) query plans for our example query from Section 3.2.2:

```
SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES 8
```



(a) Reverse plan, correct answers

(b) Reverse plan, erroneous answers



(c) Basic plan, correct answers

Figure 3.6: Interactions among parallelism, cost, and latency

For this experiment we start with empty raw tables. Thus, Deco begins by invoking eight anchor-value fetches in parallel, so the crowd can contribute to all eight (potential) result tuples at the same time. Additional fetches are invoked as needed when some of the original eight complete, but never more fetches than would contribute to eight result tuples. In our graphs, the x-axis corresponds to the degree of parallelism d , defined as the maximum number of result tuples being produced in parallel. Thus, Deco's behavior for our example query is at $d=8$.

Figure 3.6a shows the monetary cost and latency of executing the reverse plan for $d=1..15$, with our crowd simulator configured to always give correct

answers. We assume each fetch takes 5 seconds and costs \$0.05, for all fetch rules. Since each result tuple needs three rounds of answers (e.g., Spanish \Rightarrow Peru, Peru \Rightarrow Spanish, and two instances of Peru \Rightarrow Lima in parallel), the minimum cost and latency are $8 \times 4 \times \$0.05 = \1.6 and $3 \times 5s = 15s$, respectively. In fact, Deco's behavior ($d=8$) minimizes both monetary cost and latency. As we motivated informally in Section 3.1.3, too much parallelism ($d>8$) increases monetary cost, while too little ($d<8$) increases latency. Compared to no parallelism ($d=1$, equivalent to the traditional iterator model), Deco reduced latency dramatically without increasing the cost.

Figure 3.6b shows the cost and latency of executing the reverse plan, with our simulator configured to give incorrect answers with 25% chance. Each data point reported is the average of 50 trials. Due to incorrect answers, Deco often ends up invoking fetch rule language \Rightarrow country more than eight times, so the cost remains at the minimum until $d=9$, which is the best degree of parallelism. Although Deco did not quite achieve the minimum, latency is still very close to the minimum across $d=1..9$.

Figure 3.6c shows the cost and latency of executing the basic plan, with our simulator set to always give correct answers. Since cost and latency of executing this plan depend on the order in which countries are obtained, we picked a random order and used the same order for all data points. As in Figure 3.6a, Deco found the best degree of parallelism ($d=8$), reducing latency as much as possible while minimizing cost. (As a side note, the non-monotonicity of cost increase for $d>8$ is related to the positions Spanish-speaking countries appear in the random order of countries obtained.)

3.6.3 Effectiveness of Fetch Prioritization

Now we present two experiments evaluating the effectiveness of fetch prioritization. Experiment 3 evaluates our two scoring functions for varying amounts of existing data in the raw tables, while Experiment 4 evaluates the scoring functions for different data distributions. In both experiments, we compare our scoring

functions against the optimal prioritization computed using a brute-force enumeration, and against random prioritization (i.e., equal scores).

For both experiments, we use the following fetch rules:

- [Country] country \Rightarrow language and country \Rightarrow capital
- [City] city,country \Rightarrow population

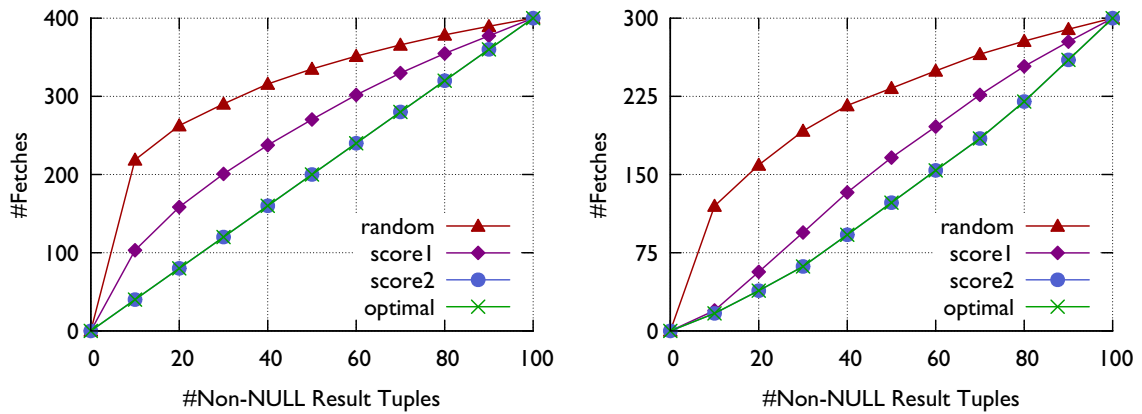
Recall the resolution functions for the language, capital, and population attributes are *majority-of-3*, *majority-of-3*, and *average-of-2*. Also, we set our simulator to always give correct answers, so that comparison against the optimal case makes sense. Each data point in the figures is the average of ten trials, which had little variance.

Experiment 3: Varying Amount of Existing Data We use the following simple query:

```
SELECT country, language, capital
FROM Country
MINTUPLES X
```

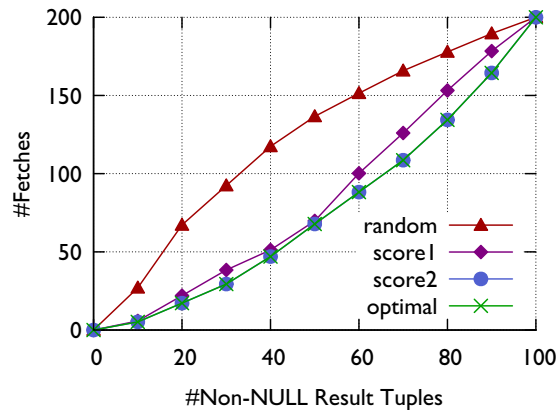
We seed anchor table CountryA with 100 different country names. In this setting, each outstanding fetch contributes to only one NULL value (either language or capital for a particular country), so the partial result tuples are independent from each other. We can achieve the optimal prioritization by completing first those partial result tuples requiring fewer overall fetches to fill in. In fact, $score_2$ behaves exactly that way in this experiment: For a partial result tuple requiring k fetches to fill in, each of those k fetches has a $score_2$ of $1/k$. On the other hand, $score_1$ first completes partial result tuples with fewer NULLs, ignoring the effect of resolution functions on number of overall fetches.

Figure 3.7a shows the number of fetches completed to obtain X result tuples using $score_1$ and $score_2$, starting with empty dependent tables. Because our simulator gives correct answers only, we always need 400 fetches (two fetches for each NULL value) to obtain 100 result tuples. We observe that prioritization based on $score_1$ and $score_2$ needed 22% and 34% fewer fetches on average than random prioritization. As expected, $score_2$ achieves the optimal prioritization for this particular experiment.



(a) Empty dependent tables

(b) 100 tuples in dependent tables



(c) 200 tuples in dependent tables

Figure 3.7: Effectiveness of fetch prioritization (Experiment 3)

In Figures 3.7b and 3.7c, we start the query with 100 and 200 tuples, respectively, in the two dependent tables combined. These tuples are randomly chosen from 400 correct tuples that would fill in all NULL values. In Figure 3.7b, the overall trends are similar to Figure 3.7a: prioritization based on $score_1$ and $score_2$ needed 28% and 41% fewer fetches on average than random prioritization. In Figure 3.7c, prioritization based on $score_1$ and $score_2$ needed 32% and 39% fewer fetches on average than random prioritization. In Figure 3.7c the difference between $score_1$ and $score_2$ is much smaller because the simplifying assumption behind $score_1$ holds more often: As we seed more tuples in the dependent tables, more NULL values can be filled in by completing one outstanding fetch.

Experiment 4: Different Data Distributions For this experiment we use the following join query:

```
SELECT city, country, population, language
FROM Country, City
WHERE City.country = Country.country
MINTUPLES X
```

Anchor tables CountryA and CityA are already populated with a varying number and distribution of values. In this query, some outstanding fetches for language might complete multiple NULL values in the partial result, depending on the distribution of city-country pairs. Because the previous experiment showed that existing data has little impact on the overall trends, we start with empty dependent tables.

Figures 3.8a and 3.8b show the number of fetches to obtain X result tuples for two real datasets: the 100 largest European cities and the 200 largest cities in the world. Each country in the European and world city dataset has 4.3 and 2.9 cities on average, respectively. The European city dataset contains eight countries (out of 23) with only one city, while the world city dataset contains a long tail of 39 countries (out of 69) with only one city. In Figure 3.8a, $score_2$ -based and optimal prioritization needed 34% and 40% fewer fetches on average than random prioritization. In Figure 3.8b, $score_2$ -based and optimal prioritization needed 37% and 42% fewer fetches on average than random prioritization. Overall, prioritization based on $score_2$ is quite close to the optimal across the entire range of X .

For Figure 3.8c, we deliberately generated a synthetic set of 100 city-country pairs to make our $score_2$ -based prioritization work as poorly as possible: all language values must be completed before any population values. Considering the resolution functions *majority-of-3* and *average-of-2*, $score_2$ for country \Rightarrow language can be as low as $k/5$ where k is the number of cities for the country value. Since $score_2$ for city, country \Rightarrow population can be as high as $1/2$ in this scenario, k should be at least 3 for all countries. Thus, the worst-case dataset contains 32 countries with three cities and one country with four cities. In Figure 3.8c, $score_2$ -based prioritization and the optimal schedule needed 19% and 35% fewer fetches on average than random prioritization.

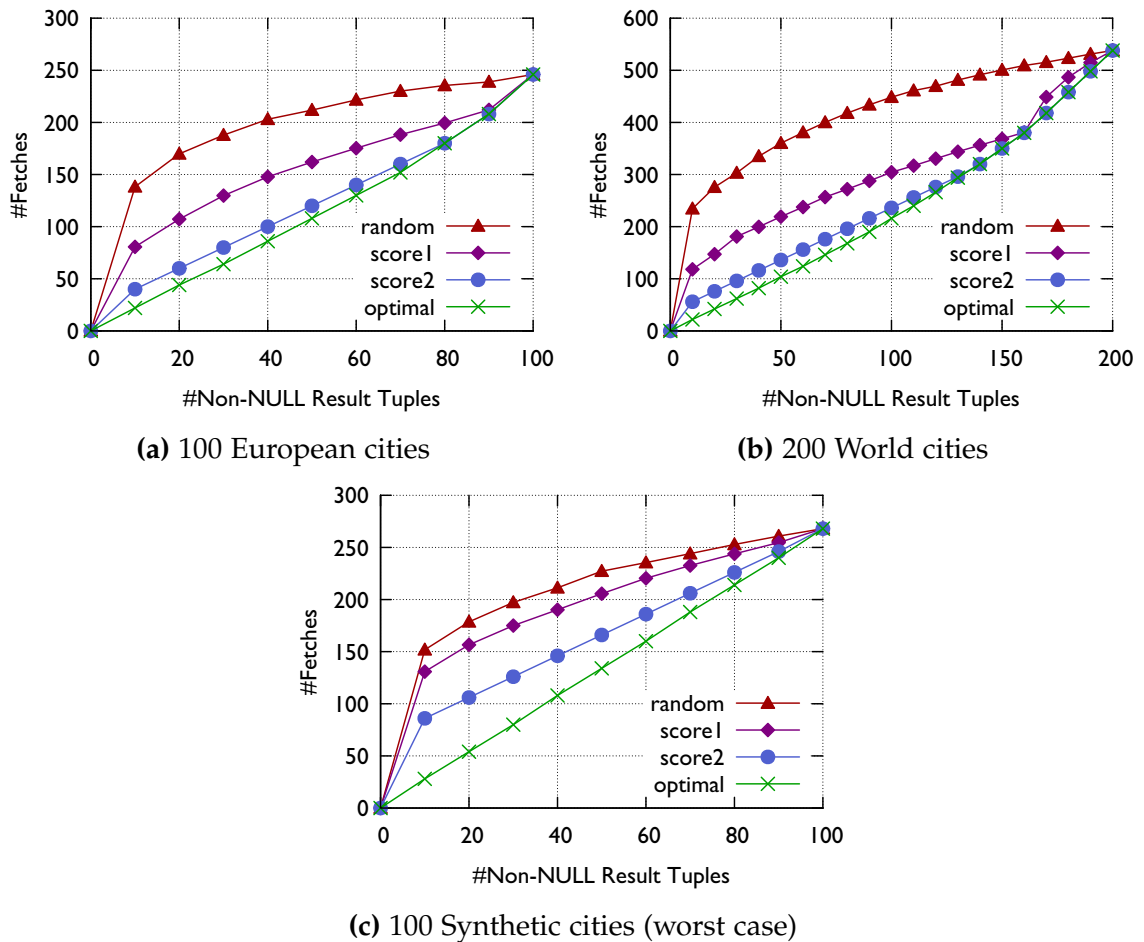


Figure 3.8: Effectiveness of fetch prioritization (Experiment 4)

3.7 Related Work

Among prior work in the general area of query processing, WSQ-DSQ [28] and incremental view maintenance [12] are most relevant to Deco’s approach to query execution. WSQ-DSQ enables an iterator execution model to concurrently access high-latency data sources, in the context of integrating web search results into a relational database. However, the WSQ-DSQ “placeholder” solution does not apply to Deco because of Deco’s flexible fetch and resolution rules. Parallel database systems [21] also access data sources in parallel. However, they typically rely on

static data partitioning and operator replication, so their approach is not applicable to Deco, where arbitrary and dynamically adjustable degrees of parallelism must be supported. View maintenance algorithms (e.g., [12]) propagate base table updates to a materialized view. Deco’s query execution engine similarly applies given base (raw) table changes to a partial query result, but in Deco base-data changes also influence further query execution.

3.8 Conclusion

We presented the query execution component of Deco’s query processor. To implement Deco’s data model and query semantics we introduced several Deco-specific query operators, and we developed a novel query plan execution strategy inspired in part by incremental view maintenance. To achieve our optimization goals of minimizing monetary cost and reducing latency when executing a given query plan, we incorporated several novel techniques into the execution engine, including two-phase query execution, a hybrid execution model, and dynamic fetch prioritization. The hybrid execution model enables the query execution engine to carefully control parallel access to the crowd, with the right degree of parallelism based on two key parallelism objectives: Never parallelize accesses if it might increase the monetary cost, and always parallelize accesses if it cannot increase the monetary cost. We validated experimentally that different query plans have different performance characteristics, and that it is worthwhile to support Deco’s flexible fetch rules in the execution engine. Moreover, our experiments showed that the hybrid execution model and fetch prioritization are effective in achieving our optimization goals.

Chapter 4

Query Optimization in Deco

We now consider Deco’s cost-based query optimizer. Given a query Q in Deco’s query language, the function of the query optimizer is to find the best query plan to answer Q , where “best” means the least estimated monetary cost across all possible query plans. The query plan chosen by the query optimizer will be executed by the query execution engine as described in Chapter 3, so that Deco’s query processor as a whole can achieve the primary and secondary goals of minimizing cost, then reducing latency. As we saw in Section 3.6.1, total monetary cost incurred by fetches during query execution does vary significantly across different query plans; therefore, plan selection by the query optimizer plays a critical role in accomplishing our primary goal of minimizing monetary cost when executing a query.

In this chapter, we present details of the query optimizer, which enumerates valid query plans in a search space, estimates monetary costs of executing those plans, and ultimately chooses the estimated best one among the plans.¹ In Section 4.1, we explain the overall steps involved in query optimization, then discuss several new challenges in Deco’s query optimization, along with our approach to tackling them. In Section 4.2, we describe how Deco estimates the monetary cost of executing a given query plan. In Section 4.3, we describe the search space of alternative valid plans for a query. In Section 4.4, we present Deco’s plan enumeration algorithm, which explores the search space and applies the cost estimation

¹The material presented in this chapter first appeared in [45].

algorithm. Section 4.5 describes our experimental evaluation of Deco’s query optimizer, Section 4.6 covers related work, and we conclude in Section 4.7.

4.1 Challenges and Approach

The query optimizer in a traditional database system typically consists of three components [15]: First, a *cost estimation algorithm* is used to produce the estimated cost of executing a given query plan. Second, a *search space* defines a set of valid query plans to be considered by the query optimizer. Lastly, a *plan enumeration algorithm* generates query plans in the search space, applies the cost estimation algorithm to each query plan, and outputs the plan with the least estimated cost.

Estimating the cost of a query plan typically involves *cardinality estimation*, which computes the expected number of output tuples for each component (sub-plan) of the plan [27]. Cardinality estimation is considered a key component of cost estimation, because the overall execution cost typically depends heavily on the cardinality. Traditional database systems maintain *statistics* that concisely capture distributions of underlying data [16], so that cardinality can be estimated based on the statistics, without consulting the full data.

In standard database systems, cardinality does not depend on execution details in query plans such as access methods and join algorithms. Thus, many different query plans may share exactly the same cardinality. Plan enumeration typically invokes cardinality estimation only once for all query plans corresponding to the same algebraic representation of the query (often called a *logical plan*), to avoid redundant work and speed up the entire process.

Although Deco’s query optimizer has similar overall structure to a traditional query optimizer, Deco’s query semantics and approach to plan execution give us several unique challenges in cost estimation and plan enumeration. In this section we motivate those challenges and discuss how Deco’s query optimizer addresses them.

4.1.1 Cost and Cardinality Estimation

Recall Deco’s valid-instance query semantics from Section 2.2: The answer to a Deco query must represent the result of evaluating the query over some valid instance of the database obtained by a Fetch-Resolve-Join sequence, starting with the current contents of the database. Thus, a query result must reflect all existing data in the database, and Deco obtains new data from the crowd while inserting the new data into the database until a sufficient number of result tuples are present. Since existing data is “free” and new data is not, Deco’s cost model must distinguish between existing data and new data to estimate monetary cost properly. We will see in Section 4.2.1 how Resolve and Fetch operators in Deco query plans (Section 3.2.2) take into account the amount of relevant existing data to estimate the amount of new data required to produce the result. Moreover, cardinality estimation must be based on some estimated final database state; thus, Deco’s cardinality estimation algorithm estimates cardinality and the final state of database simultaneously.

4.1.2 Statistics

To estimate cardinality in Deco query plans, we require some statistical information about both existing data and new data. For existing data, we use the statistical information maintained by the back-end RDBMS storing raw tables (recall Section 2.1.2). For data obtained from the crowd, we primarily rely on information provided by the schema designer and/or end-user (recall Section 2.1.1). Specifically we require a *selectivity factor* to be provided by the schema designer for each resolution function (further discussed below). For predicates, we also allow the end-user to provide a selectivity factor (below); if none is provided we resort to heuristic default values based on predicate types, also known as “magic numbers” [16].

A selectivity factor of σ for a predicate p says that the predicate p has a σ chance of being satisfied on a given data item. For example, the selectivity of predicate `language='Spanish'` may be around 0.1 in the Country relation, because there are about 20 Spanish-speaking countries out of about 200 countries in the

world. For resolution functions, the selectivity factor estimates how many output tuples are produced on average by each input tuple. For example, the selectivity factor of resolution function *majority-of-3* with shortcutting depends on how often shortcutting is expected to happen, ranging from $1/3$ (when the first two values are never expected to agree) to $1/2$ (when the first two values are always expected to agree).

4.1.3 Plan Enumeration

As we will see in Section 4.3, Deco’s query plans are based on a join tree that becomes an algebraic representation of the query, followed by fetch rule selections. Unlike in traditional database systems, different fetch rule selection for the same algebraic plan may produce query results based on different valid instances of the database, resulting in different cardinality. Thus, in comparison with traditional plan enumeration, there are far fewer opportunities to reuse cardinality estimates across alternative query plans, or prune inferior subplans early. Deco’s plan enumeration does reuse computations to the extent possible, achieving better efficiency than a fully naive plan enumeration.

4.2 Cost Estimation

We describe how Deco’s query optimizer estimates the cost of executing a given query plan. We will see in Section 4.4 that Deco’s plan enumeration algorithm invokes cost estimation for each query plan in the search space, to determine the best overall query plan. Recall that our optimization metric is the total monetary cost incurred by fetches. Thus, Deco’s cost estimation algorithm takes as input a query plan and statistics about data (both existing data and crowdsourced data), and produces as output the estimated cost in dollars. In a sense our cost model estimates resource consumption in a similar fashion to traditional database systems [27], except the resource is money instead of CPU, I/O, and communication costs.

Recall from Chapter 3 that a Deco query plan is a rooted DAG of query operators. Figure 4.1 repeats three of our example query plans from Section 3.2.2 for the following query:

```
SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES 8
```

Recall from Section 2.1.3 that there is a fixed monetary cost associated with each fetch rule, and this cost is specified by the schema designer. For example, referring back to our running example introduced in Chapter 2, fetch rules $\text{country} \Rightarrow \text{language}$ and $\text{country} \Rightarrow \text{capital}$ may cost \$0.03 and \$0.05 per fetch, respectively. Although costs may differ across fetch rules, we assume the cost-per-fetch of one rule $A_1 \Rightarrow A_2$ does not depend on the specific values for A_1 (even though, conceivably, the level of difficulty to answer such questions may vary based on the values for A_1).

It turns out we can reduce the monetary cost estimation problem to the cardinality estimation problem. However, the notion of cardinality from traditional databases [27] has to be adjusted, since Deco inserts new tuples into raw tables during query execution. In Deco, we estimate cardinality of a subplan as the total number of output tuples expected in order to obtain a query result with a sufficient number of tuples. Since no Deco query operators except Fetch cost money, we have the following formula for estimated monetary cost:

$$\text{monetary cost} = \sum_{\text{Fetch operator } F} F.\text{cost-per-fetch} \times F.\text{card} \quad (4.1)$$

where $F.\text{cost-per-fetch}$ and $F.\text{card}$ denote the cost-per-fetch and estimated cardinality of Fetch operator F , respectively.

Because the schema designer specifies $F.\text{cost-per-fetch}$, estimating the monetary cost amounts to estimating $F.\text{card}$, the cardinality of each Fetch operator F . We will see that to estimate the cardinality of each Fetch operator, we need to estimate cardinality for other parts of the plan as well. Section 4.2.1 specifies

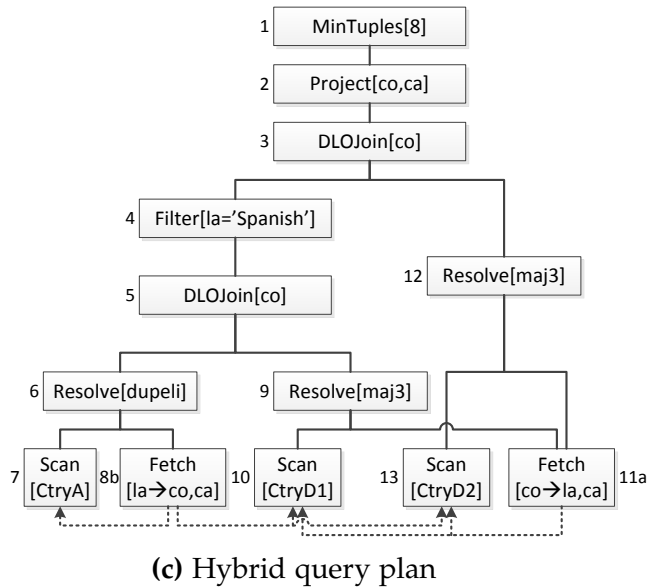
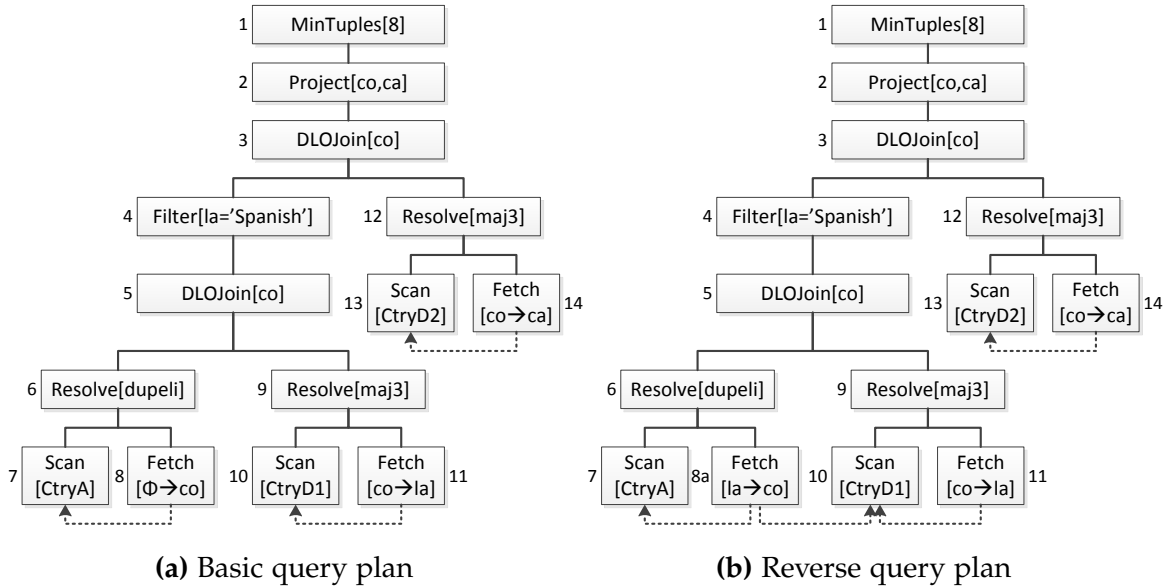


Figure 4.1: Example query plans

our cardinality estimation algorithm. Section 4.2.2 provides cardinality and cost estimation examples.

4.2.1 Cardinality Estimation Algorithm

We describe a procedure `EstimateCard` that estimates cardinality for each operator op in a query plan. Specifically, we estimate the total number of output tuples from the subplan rooted in op that are needed in order to produce the required number of result tuples for the entire plan. Procedure `EstimateCard` is specialized for each operator type and takes two parameters:

- `preds`: an array of k predicates (with their selectivities)
- `target`: a target number of output tuples (at operator op) satisfying all of the predicates in `preds`

Using these two parameters, operator op receives from its parent operator the requirement for obtaining a sufficient number of overall result tuples, confined to the subplan rooted at op : operator op must produce at least `target` output tuples satisfying all predicates in `preds`. To estimate cardinality, operator op recursively calls the `EstimateCard` procedure on its children with appropriate parameters. Before the `EstimateCard` procedure on operator op returns, the output of the procedure is stored at operator op using the following three member variables:

- `op.card`: estimated cardinality of operator op .
- `op.cards`: an array of 2^k elements representing the breakdown of `op.card` by possible evaluation results of the k predicates in `preds`.
- `op.distincts`: an array containing the number of estimated distinct values for each attribute in op 's output tuples.

These three outputs of the `EstimateCard` procedure are stored rather than simply returned to the caller for two reasons: First, we need to gather `card` values of all `Fetch` operators at the end of the process to compute the total monetary cost. Second, a `card` value of a `Fetch` operator may be updated when the `EstimateCard` procedure is called again through another parent. (Thus, except for `Fetch` operators, technically these outputs could be returned to the caller.) To initiate cardinality estimation for a plan, we call `root.EstimateCard(0, ∅)`.

We digress briefly to explain some implementation considerations of these outputs. To define $op.cards[i]$ ($0 \leq i < 2^k$) precisely, let b_0, b_1, \dots, b_{k-1} denote the binary representation of i (i.e., $i = \sum_{j=0}^{k-1} b_j 2^{k-j-1}$ and $b_j \in \{0, 1\}$). The binary value b_j encodes whether predicate $preds[j]$ is satisfied ($b_j=1$) or not. Thus, $op.cards[i]$ is the estimated cardinality corresponding to the combination of predicate evaluation results encoded by the index i . The following equation holds by definition: $op.card = \sum_{0 \leq i < 2^k} op.cards[i]$. Note that the default cards array is exponential in size, i.e., 2^k . If k is expected to be large, the implementation could always assume independence of the k predicates and represent cards by a set of arrays of size k , each of which captures different distributions under the independence assumption, resulting in reduced space complexity for storing cards. We have not implemented this approach, but it is a simple modification to our system. For $op.distincts$, no array element can be larger than $op.card$. Note cardinality estimation of a given query plan only uses certain elements in the $op.distincts$ array based on join predicates, although our notation is defined for all attributes in the schema.

Having defined the signature of the EstimateCard procedure, we now go through the actual implementation for each operator type. Recall from Section 3.2.1 that our operators are MinTuples, Project, Filter, DLOJoin, Resolve, Fetch, DepJoin, and Scan. Note Scan operators do not explicitly participate in cardinality estimation.

```
MinTuples.EstimateCard(target, preds)
1 child.EstimateCard(this.minTuples,  $\emptyset$ )
2 this.card  $\leftarrow$  child.card
3 this.cards  $\leftarrow$  child.cards
4 this.distincts  $\leftarrow$  child.distincts
```

Since the root of every Deco query plan is a MinTuples operator, MinTuples.EstimateCard is the entry point of our cardinality estimation algorithm as a whole. It calls EstimateCard recursively on its child, with parameter target set to the number of tuples in the MinTuples clause, and no predicates for parameter preds. (Note that output tuples of the child operator of the root are guaranteed to satisfy all

predicates in the query.) When the recursive call returns, the cardinality estimation algorithm terminates, and each F .card stores estimated cardinality for Fetch operator F , from which we calculate the estimated monetary cost using Equation (4.1).

```
Project.EstimateCard(target, preds)
1 child.EstimateCard(target, preds)
2 this.card  $\leftarrow$  child.card
3 this.cards  $\leftarrow$  child.cards
4 this.distincts  $\leftarrow$   $\Pi$  child.distincts
```

For the Project operator, the EstimateCard procedure simply continues the recursion because the cardinality does not change at all.

```
Filter.EstimateCard(target, preds)
1 child.EstimateCard(target, preds  $\cup$  {this.pred})
2 this.card  $\leftarrow$  0
3 this.cards  $\leftarrow$  {0, ..., 0}
4 for  $i = 0$  to  $2^{\text{len}(\text{preds})} - 1$  do
5   this.card  $\leftarrow$  this.card + child.cards[2i+1]
6   this.cards[i]  $\leftarrow$  child.cards[2i+1]
7 end for
8 this.distincts  $\leftarrow$   $\text{min}_{\text{elementwise}}$ (child.distincts, {this.card, ..., this.card})
```

The Filter operator recursively calls EstimateCard on its child operator with its own predicate as well as the k predicates received from its parent operator. When the recursive call returns, child.cards contains 2^{k+1} elements. Then, the Filter computes its estimated cardinality by summing up the 2^k elements whose indexes indicate that its own predicate is satisfied.

```
DLOJoin.EstimateCard(target, preds)
1 outer.EstimateCard(target, preds)
2  $d \leftarrow$  outer.distincts[this.pred.left]
3 if outer.cards[ $2^{\text{len}(\text{preds})} - 1$ ] > target then
4   inner.EstimateCard( $\alpha \times$  target  $\times d /$  outer.card +  $(1 - \alpha) \times d$ ,  $\emptyset$ )
5 else
6   inner.EstimateCard( $d$ ,  $\emptyset$ )
```

```

7 end if
8 this.card ← outer.card
9 this.cards ← outer.cards
10 this.distincts ← outer.distincts ∪ inner.distincts

```

For the DLOJoin operator, we first call EstimateCard on the outer child operator. We intentionally pass all predicates in preds to the outer, even though some dependent attributes are obtained from the inner. (We will see shortly how Resolve and Fetch operators use these predicates to estimate the number of required anchor values.) Note that we do not pass the left outerjoin predicate, which is always satisfied by definition.

Once the recursive call on the outer returns, we call EstimateCard on the inner child to eventually estimate the number of new dependent values required to produce a query result. Without considering dynamic fetch prioritization (Section 3.5), parameter target would simply be $d = \text{outer.distincts}[\text{this.pred.left}]$. However, when there are more anchor values than needed due to existing data, fetch prioritization takes effect. Deco prioritizes those fetches filling in dependent attributes so that a sufficient number of result tuples are produced as soon as possible. As a result, some anchor values are not expected ever to be joined.

Since it is very difficult to predict the exact outcome of fetch prioritization due to its heuristic approach, we discount parameter target using a configurable weight $0 \leq \alpha \leq 1$. With $\alpha = 0$, we overestimate the number of new dependent values, because joining all d anchor values may eventually produce far more result tuples than needed. On the other hand, with $\alpha = 1$, we assume optimal fetch prioritization as well as no unfavorable correlations in the existing data, so we underestimate the number of dependent values. In Section 4.5.1 (Experiment 2), we empirically determine a good range for α .

```

Resolve.EstimateCard(target, preds)

```

```

1 compute this.card, this.cards and this.distincts based on the existing data in
  the resolved raw tables (using the back-end RDBMS)
2  $t \leftarrow \text{this.cards}[2^{\text{len}(\text{preds})} - 1]$ 
3 fetch.EstimateCard(max(0, target - t), preds ∪ {this.resolution_function})
4 card ← 0
5 for  $i = 0$  to  $2^{\text{len}(\text{preds})} - 1$  do

```

```

6   card ← card + fetch.cards[2i+1]
7   this.card ← this.card + fetch.cards[2i+1]
8   this.cards[i] ← this.cards[i] + fetch.cards[2i+1]
9   end for
10  this.distincts ← this.distincts + {card, ..., card}

```

The Resolve operator first computes `card`, `cards`, and `distincts` based on the existing data in the resolved raw tables, without considering parameter `target`. Our approach is to exploit the available statistics provided by the back-end RDBMS. Once `card`, `cards`, and `distincts` are computed based on the existing data, the Resolve operator calls the EstimateCard procedure on its child Fetch operator to estimate the cardinality of required new data. Since existing data contributes to $t = \text{cards}[2^{\text{len}(\text{preds})} - 1]$ output tuples satisfying all predicates in `preds`, parameter `target` is set to $\max(0, \text{target} - t)$. Also, the Resolve operator passes the selectivity of its resolution function as part of parameter `preds`.

```

Fetch.EstimateCard(target, preds)
1  card ← target
2  for each pred ∈ preds do
3    if pred.left ∉ this.lhs_attrs then
4      card ← card / pred.selectivity
5    end if
6  end for
7  if this.card < card then
8    this.card ← card
9    this.cards ← {card, ..., card}
10 for i = 0 to 2len(preds) - 1 do
11   for j = 0 to len(preds) - 1 do
12     if preds[j].left ∈ this.lhs_attrs then
13       selectivity ← 1.0
14     else
15       selectivity ← preds[j].selectivity
16     end if
17     if i & 2len(preds) - 1 - j == 0 then
18       this.cards[i] ← this.cards[i] × (1 - selectivity)
19     else
20       this.cards[i] ← this.cards[i] × selectivity
21     end if
22   end for

```

```

23   end for
24 end if

```

As a base case of our recursive process, the Fetch operator estimates its cardinality based on its associated fetch rule and the parameters *target* and *preds*. We assume that new data obtained by the Fetch operator always satisfies those predicates in *preds* that are used to instantiate the left-hand side of the fetch rule. (For example, in Figure 3.2, new countries obtained by Fetch operator 8a satisfy predicate *language*='Spanish'.) Moreover, we assume the other predicates in *preds* are independent for the new data: the probability of a new tuple satisfying a predicate *p* is the selectivity of *p*.

Unlike all other operator types, a Fetch operator may have more than one parent operator, when the right-hand side of its fetch rule contains dependent attributes spanning multiple raw tables and no anchor attributes. (Fetch operator 11a in Figure 3.4 is one such example.) In this case, the EstimateCard procedure is called on the Fetch operator as many times as the number of its parent Resolve operators. Since the distribution of new data remains the same, the Fetch operator simply retains the maximum estimated cardinality across all calls.

```

DepJoin.EstimateCard(target, preds)
1  outer.EstimateCard(target, preds  $\cup$  {this.pred})
2  inner.EstimateCard(outer.distincts[this.pred.left],  $\emptyset$ )
3  this.card  $\leftarrow$  0
4  this.cards  $\leftarrow$  {0, ..., 0}
5  for  $i = 0$  to  $2^{\text{len}(\text{preds})} - 1$  do
6    this.card  $\leftarrow$  this.card + outer.cards[2i+1]
7    this.cards[i]  $\leftarrow$  outer.cards[2i+1]
8  end for
9  this.distincts  $\leftarrow$  minelementwise(
    outer.distincts  $\cup$  inner.distincts, {this.card, ..., this.card})

```

Roughly, our implementation of EstimateCard for DepJoin combines EstimateCard of DLOJoin and Filter. One simplifying assumption we make is that new outer tuples produced by the crowd do not match existing inner tuples. This assumption will hold if queries typically join the same combinations of conceptual relations,

since past queries will tend to have materialized either both or neither of a joining pair of tuples. Of course this assumption may not always hold. For example, if the inner relation is queried more frequently than the outer, the assumption can be violated to a large degree, resulting in large error. Note in the case where we have a bound on the number of distinct values for a join attribute, we could potentially remove this assumption.

4.2.2 Cost Estimation Examples

Let us walk through two simple example runs of our cost estimation algorithm. Suppose we execute the example query from Section 3.2.2:

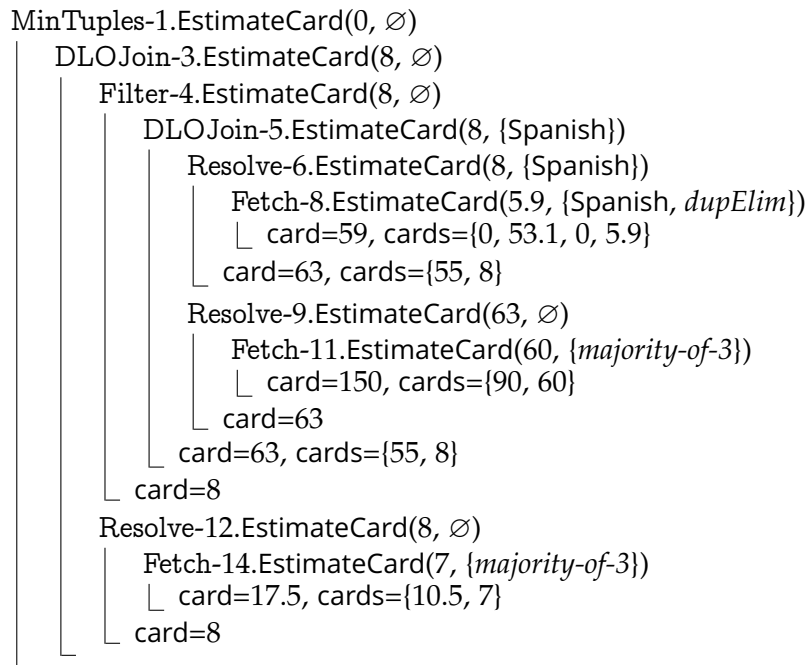
```
SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES 8
```

We consider two plans for this query: the basic plan (Figure 4.1a) and the reverse plan (Figure 4.1b). For both examples we start with the following resolved raw tables:

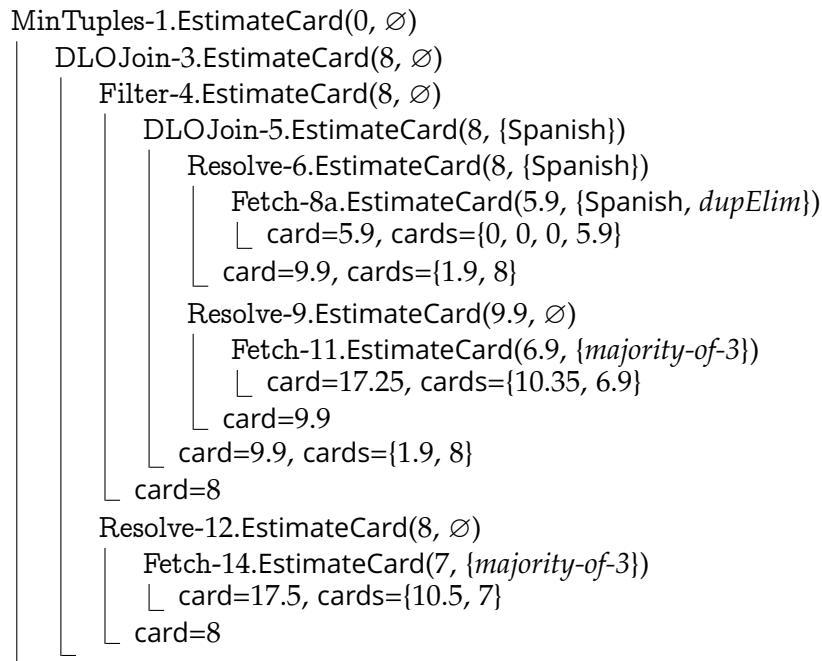
CountryA	CountryD1		CountryD2	
country	country	language	country	capital
Chile	Korea	Korean	Korea	Seoul
Korea	Peru	Spanish	Spain	Madrid
Peru	Spain	Spanish		
Spain				

We assume the selectivity factor of predicate `language='Spanish'` is 0.1. Also, we assume the selectivity factors of resolution functions `dupElim` and `majority-of-3` are 1.0 and 0.4, respectively. Finally we assume each fetch costs \$0.05, for all fetch rules.

Basic plan: Figure 4.2a shows a trace of our cardinality estimation algorithm for the basic plan. Starting with calling `EstimateCard(0, ∅)` on the `MinTuples` operator 1, we recursively call `EstimateCard` until we reach `Resolve` operator 6. At



(a) Basic plan



(b) Reverse plan

Figure 4.2: Trace of EstimateCard

this point, $\text{target} = 8$, and preds has predicate $\text{language} = \text{'Spanish'}$. Considering existing data in resolved CountryA and CountryD_1 , there are two tuples satisfying the predicate (Peru and Spain), one tuple not satisfying the predicate (Korea), and one tuple unknown (Chile). Since the unknown tuple has 10% chance of satisfying the predicate, we have $\text{card} = 4$ and $\text{cards} = \{1.9, 2.1\}$. To produce eight tuples satisfying the predicate, we need $8 - 2.1 = 5.9$ more tuples, so Resolve operator 6 calls $\text{EstimateCard}(5.9, \{\text{language} = \text{'Spanish'}, \text{dupElim}\})$ on Fetch operator 8. Since the fetch rule of operator 8 is $\emptyset \Rightarrow \text{country}$, we have $\text{card} = 59$ and $\text{cards} = \{0, 53.1, 0, 5.9\}$ at Fetch operator 8.

As the recursion unwinds, Resolve operator 6 and DLOJoin operator 5 have estimated cardinality of 63. Since resolved CountryD_1 has three existing tuples, Resolve operator 9 needs to produce $63 - 3 = 60$ more tuples, and Fetch operator 11 has estimated cardinality of $60/0.4 = 150$. Similarly, Filter operator 4 has estimated cardinality of eight, and Resolve operator 12 needs to produce $8 - 1 = 7$ more tuples. (Note only the Spain tuple in resolved CountryD_2 is “relevant” at this point.) Thus, Fetch operator 14 has estimated cardinality of $7/0.4 = 17.5$. The final estimated monetary cost is $\$0.05 \times (59 + 150 + 17.5) = \11.325 .

Reverse plan: Figure 4.2b shows a trace of our cardinality estimation algorithm for the same query but using the reverse plan. Notice that the trace is exactly same until we reach Fetch operator 8a with target of 5.9 and preds being $\{\text{language} = \text{'Spanish'}, \text{dupElim}\}$. Since the fetch rule of operator 8a is $\text{language} \Rightarrow \text{country}$, we assume all new tuples satisfy predicate $\text{language} = \text{'Spanish'}$. Thus, we have $\text{card} = 5.9$ and $\text{cards} = \{0, 0, 0, 5.9\}$ at Fetch operator 8a. As recursion unwinds, Resolve operator 6 and DLOJoin operator 5 have estimated cardinality of 9.9. Thus Resolve operator 9 needs to produce $9.9 - 3 = 6.9$ more tuples, and Fetch operator 11 has estimated cardinality of $6.9/0.4 = 17.25$. The rest of the trace is the same as above, and the estimated monetary cost is $\$0.05 \times (5.9 + 17.25 + 17.5) = \2.0325 .

Our cost estimation indicates that the basic plan is expected to be about six times as expensive as the reverse plan for this setting.

4.3 Search Space and Plan Generation

Next we define the search space of alternative plans that Deco's query optimizer considers, then we describe how to generate all plans in the search space given a query. Deco's Fetch-Resolve-Join semantics (Section 2.1.5) enables interesting plan alternatives. It turns out we can obtain all possible query plans for a given query by considering all possible "join trees", and considering all possible sets of fetch rules for each join tree. (Later we will discuss how Deco reduces the search space by excluding some of these plans.)

- (1) **Join tree:** A join tree is a binary expression tree whose operators (internal nodes) are either cross-products or left outerjoins, and whose operands (leaf nodes) are resolved raw tables.
- (2) **Fetch rules:** For each raw table, a Deco query plan requires one fetch rule assigned to obtain additional tuples for the raw table.

Specifically we obtain all query plans for a given query Q in the following four steps: First, we find all join trees for Q based on Q 's From clause. Second, we transform each join tree into an *algebraic representation* of Q by placing predicates in Q 's Where clause. Third, for each algebraic representation, we generate all possible combinations of fetch rules from the set of available fetch rules. Finally, for each algebraic representation and combination of fetch rules, we construct the corresponding complete query plan for Q .

Given our example query from Section 4.2, this strategy will generate many query plans including the three example plans in Figure 4.1 (assuming all fetch rules in the example plans are available). As it happens, all three plans correspond to the same join tree and the same algebraic representation; however, different fetch rules were chosen for each plan.

In the remainder of this section we describe the four steps in detail. Section 4.4 addresses how to explore the search space and find the best plan using the cost estimation algorithm from Section 4.2.

4.3.1 Join Tree

Given a Deco query Q over one or more conceptual relations, our goal is to find all join trees over raw tables that evaluate the cross product of all conceptual relations in the From clause of the query Q . We first describe several transformations that give us alternative join trees, then we identify a set of properties characterizing all valid join trees.

Let us first consider a Deco query containing only one conceptual relation R in its From clause. Here we are interested in finding all join trees equivalent to the following left-deep tree:

$$R = ((A \bowtie D_1) \bowtie D_2) \bowtie \cdots \bowtie D_k$$

where A and D_1, \dots, D_k denote the resolved anchor table and dependent tables for R , respectively. Although left outerjoins do not commute in general, we can reorder them using the following equation when neither outerjoin predicates are between Y and Z [49]:

$$(X \bowtie Y) \bowtie Z = (X \bowtie Z) \bowtie Y \quad (4.2)$$

By repeatedly applying Equation (4.2) to the left-deep join tree above, we obtain all $k!$ left-deep join trees with A as the first (left-most) operand and D_1, \dots, D_k in any order after that.

Now consider the general case where a query Q contains n conceptual relations R_1, \dots, R_n in its From clause. The goal is to find all join trees equivalent to the following default tree:

$$\begin{aligned} R_1 \times \cdots \times R_n = & (((A_1 \bowtie D_{11}) \bowtie D_{12}) \bowtie \cdots \bowtie D_{1k_1}) \times \cdots \\ & \times (((A_n \bowtie D_{n1}) \bowtie D_{n2}) \bowtie \cdots \bowtie D_{nk_n}) \end{aligned} \quad (4.3)$$

where A_i and D_{i1}, \dots, D_{ik_i} ($k_i \geq 0$) denote the resolved anchor table and dependent tables for R_i , respectively. In addition to Equation (4.2), the following

equation holds when the outerjoin predicate is between Y and Z [49]:

$$X \times (Y \bowtie Z) = (X \times Y) \bowtie Z \quad (4.4)$$

This equation allows us to “interleave” resolved raw tables from different conceptual relations in a join tree, opening up more opportunities to push down predicates. (We will see in Section 4.3.2 that evaluating predicates early can eliminate unnecessary fetches to obtain inner tuples at some joins, thus reducing monetary cost.)

Finally we can use the commutativity and associativity of cross-products:

$$X \times Y = Y \times X, \quad X \times (Y \times Z) = (X \times Y) \times Z \quad (4.5)$$

Now we are interested in every join tree derived by repeatedly applying Equations (4.2), (4.4), or (4.5) to the default join tree. The following theorem exactly characterizes our valid join trees, proving that the space of join trees we obtain satisfies certain properties, and furthermore any join tree satisfying those properties is equivalent to the default tree.

Theorem: Set S of join trees derived by repeatedly applying Equations (4.2), (4.4), or (4.5) to the default join tree in Equation (4.3) is equal to set T of join trees satisfying the following properties:

- All raw tables appear in the leaf nodes exactly once.
- Each outerjoin node has some dependent table D_{ij} as its right child and the corresponding anchor table A_i in its left subtree.
- Each dependent table D_{ij} is the right child of some outerjoin node.

Proof: We first show $S \subseteq T$, then show $T \subseteq S$.

It is straightforward to show $S \subseteq T$. First, the default join tree satisfies all three properties. Since the properties are invariant under Equations (4.2), (4.4), and (4.5), any join tree derived by the equations also satisfies the properties. Thus, we have $S \subseteq T$.

We show $T \subseteq S$ using mathematical induction on n , the number of conceptual relations. For $n=1$, a join tree $t \in T$ is always a left-deep tree with A_1 as the first

$$\begin{array}{ll}
(((A_1 \bowtie D_{11}) \bowtie D_{12}) \times A_2) \bowtie D_{21} & ((A_1 \bowtie D_{11}) \bowtie D_{12}) \times (A_2 \bowtie D_{21}) \\
(((A_1 \bowtie D_{11}) \times A_2) \bowtie D_{12}) \bowtie D_{21} & (((A_1 \bowtie D_{11}) \times A_2) \bowtie D_{21}) \bowtie D_{12} \\
((A_1 \bowtie D_{11}) \times (A_2 \bowtie D_{21})) \bowtie D_{12} & (((A_1 \bowtie D_{12}) \times A_2) \bowtie D_{11}) \bowtie D_{21} \\
(((A_1 \bowtie D_{12}) \times A_2) \bowtie D_{21}) \bowtie D_{11} & ((A_1 \bowtie D_{12}) \times (A_2 \bowtie D_{21})) \bowtie D_{11} \\
(((A_1 \bowtie D_{12}) \bowtie D_{11}) \times A_2) \bowtie D_{21} & ((A_1 \bowtie D_{12}) \bowtie D_{11}) \times (A_2 \bowtie D_{21}) \\
(((A_1 \times A_2) \bowtie D_{11}) \bowtie D_{12}) \bowtie D_{21} & (((A_1 \times A_2) \bowtie D_{11}) \bowtie D_{21}) \bowtie D_{12} \\
(((A_1 \times A_2) \bowtie D_{12}) \bowtie D_{11}) \bowtie D_{21} & (((A_1 \times A_2) \bowtie D_{12}) \bowtie D_{21}) \bowtie D_{11} \\
(((A_1 \times A_2) \bowtie D_{21}) \bowtie D_{11}) \bowtie D_{12} & ((A_1 \times (A_2 \bowtie D_{21})) \bowtie D_{11}) \bowtie D_{12} \\
(((A_1 \times A_2) \bowtie D_{21}) \bowtie D_{12}) \bowtie D_{11} & ((A_1 \times (A_2 \bowtie D_{21})) \bowtie D_{12}) \bowtie D_{11} \\
(((A_2 \times A_1) \bowtie D_{11}) \bowtie D_{12}) \bowtie D_{21} & ((A_2 \times (A_1 \bowtie D_{11})) \bowtie D_{12}) \bowtie D_{21} \\
(A_2 \times ((A_1 \bowtie D_{11}) \bowtie D_{12})) \bowtie D_{21} & (((A_2 \times A_1) \bowtie D_{11}) \bowtie D_{21}) \bowtie D_{12} \\
((A_2 \times (A_1 \bowtie D_{11})) \bowtie D_{21}) \bowtie D_{12} & (((A_2 \times A_1) \bowtie D_{12}) \bowtie D_{11}) \bowtie D_{21} \\
((A_2 \times (A_1 \bowtie D_{12})) \bowtie D_{11}) \bowtie D_{21} & (A_2 \times ((A_1 \bowtie D_{12}) \bowtie D_{11})) \bowtie D_{21} \\
(((A_2 \times A_1) \bowtie D_{12}) \bowtie D_{21}) \bowtie D_{11} & ((A_2 \times (A_1 \bowtie D_{12})) \bowtie D_{21}) \bowtie D_{11} \\
(((A_2 \times A_1) \bowtie D_{21}) \bowtie D_{11}) \bowtie D_{12} & (((A_2 \times A_1) \bowtie D_{21}) \bowtie D_{12}) \bowtie D_{11} \\
(((A_2 \bowtie D_{21}) \times A_1) \bowtie D_{11}) \bowtie D_{12} & ((A_2 \bowtie D_{21}) \times (A_1 \bowtie D_{11})) \bowtie D_{12} \\
(A_2 \bowtie D_{21}) \times ((A_1 \bowtie D_{11}) \bowtie D_{12}) & (((A_2 \bowtie D_{21}) \times A_1) \bowtie D_{12}) \bowtie D_{11} \\
((A_2 \bowtie D_{21}) \times (A_1 \bowtie D_{12})) \bowtie D_{11} & (A_2 \bowtie D_{21}) \times ((A_1 \bowtie D_{12}) \bowtie D_{11})
\end{array}$$

Figure 4.3: Example join trees

operand. Based on Equation (4.2), we have $t \in S$, so $T \subseteq S$ holds for $n=1$.

Now suppose $T \subseteq S$ holds for $n=1, \dots, k$ ($k \geq 1$). For $n=k+1$, consider a join tree $t \in T$. Since the three properties imply one-to-one correspondence between outerjoin nodes and dependent tables, t contains $n-1=k$ cross-product nodes. At least one of the k cross-product nodes is on the path from the root operator to the first operand, so we can pick the cross-product node γ closest to the root. (If there is no cross-product on the path, we have $n=1$, which is a contradiction.) Then, we transform t to $t' = t'_L \times t'_R$ by pushing down all outerjoin nodes above γ (if any) using Equation (4.4). Using the induction hypothesis, the left and right subtrees t'_L and t'_R can be derived by applying Equations (4.2), (4.4), or (4.5) to their corresponding default trees. Finally, Equation (4.5) gives us $t' \in S$ and $t \in S$. Therefore, $T \subseteq S$ holds for $n=k+1$. \square

As an example, suppose that a query Q contains two relations Country ($=R_1$) and City ($=R_2$) from Section 2.1.1 in its From clause. Recall that the raw schema for Country and City includes three and two raw tables, respectively. In this case there are 36 possible join trees for Q , shown in Figure 4.3 as algebraic expressions.

So far we defined the entire space of join trees that are equivalent to the default tree. To reduce the search space somewhat, we heuristically prune some join trees when a query contains three or more conceptual relations in its From clause. Specifically, we prohibit a cross-product node from having another cross-product in its right subtree. This restriction is analogous to considering left-deep join trees only in traditional optimizers, and makes sense in our setting of dependent joins: we do not want to invoke fetches based on inner values that do not necessarily pass join predicates.

4.3.2 Algebraic Representation

Based on the set of all join trees for the query Q , we next construct complete algebraic representations for Q by adding Q 's join and local predicates. Because we aim to minimize monetary cost, the only sensible algebraic representation for a given join tree is the one that evaluates all predicates as early as possible. Otherwise there exists at least one join in the join tree that receives more outer tuples than needed, and the corresponding query plan would invoke unnecessary fetches to obtain inner tuples. As an example, given a join tree $(\text{CountryA} \bowtie \text{CountryD1}) \bowtie \text{CountryD2}$ for our example query from Section 4.2, the algebraic representation that evaluates predicate `language='Spanish'` as early as possible is $\sigma_{\text{Spanish}}(\text{CountryA} \bowtie \text{CountryD1}) \bowtie \text{CountryD2}$. Note the other option $\sigma_{\text{Spanish}}((\text{CountryA} \bowtie \text{CountryD1}) \bowtie \text{CountryD2})$ would fetch capital values for non-Spanish speaking countries. Thus, each join tree is converted to exactly one algebraic representation for Q .

4.3.3 Fetch Rule Selection

Given the set of all algebraic representations corresponding to the query Q , the next step is to generate all possible combinations of fetch rules for the raw tables in each algebraic representation. For each algebraic representation, we first compute a set of candidate fetch rules for each raw table, described shortly. Then, selection of one fetch rule each from all candidate sets yields one combination of fetch rules.

Depending on the algebraic representation and set of available fetch rules, some raw tables may have an empty set of candidate fetch rules; in this case there are no possible combinations of fetch rules corresponding to the algebraic representation, so we discard the algebraic representation. (If every algebraic representation is discarded, the database does not include enough fetch rules to execute the query Q .)

Let us consider constructing a Fetch operator F to be a child of Resolve operator R , whose Scan child is associated with raw table T . (We will see in Section 4.3.4 that each raw table in the algebraic representation corresponds to a Resolve-Scan-Fetch operator triple in complete query plans.) Consider the set of candidate fetch rules for Fetch operator F . In Deco's query execution engine, Resolve operator R requests tuples from Fetch operator F to feed its resolution function. To make this mechanism work, the fetch rule $A_1 \Rightarrow A_2$ deployed in Fetch operator F must satisfy the following conditions:

- **Condition 1:** Each attribute in A_1 is instantiated by either a value from an outer tuple passed to Fetch operator F by a join, or a constant value in a local Where clause predicate.
- **Condition 2:** The attributes in $A_1 \cup A_2$ cover all attributes in raw table T , and some attributes in A_2 are attributes of T .

The first condition allows Fetch operator F to invoke the fetch rule by instantiating all attributes in A_1 . The second condition allows new tuples obtained using the fetch rule to be inserted into T and passed up to Resolve operator R .

Note the first condition is specific to the given algebraic representation, while the second one is not. Thus one possible algorithm to compute a candidate set would first find the available fetch rules satisfying the second condition, then discard those violating the first condition.

4.3.4 Complete Query Plan

Given an algebraic representation corresponding to the query Q and a combination of fetch rules for the algebraic representation, we finally construct a complete query plan. First, any Deco query plan contains a MinTuples operator at its root

and a Project operator right below the root. These two operators are constructed based on the MINTUPLES and SELECT clauses of the query. Second, selections, left outerjoins, and inner joins in the algebraic representation are directly mapped to Filter, DLOJoin, and DepJoin operators, placed below the Project operator. Lastly, each raw table T in the algebraic representation is mapped to a Resolve-Scan-Fetch operator triple, with the fetch rule for T from the given combination of fetch rules.

Continuing our example, Figure 4.1 shows three different plans expanded from the same algebraic representation $\sigma_{Spanish}(\text{CountryA} \bowtie \text{CountryD1}) \bowtie \text{CountryD2}$. It can be easily verified (and is intuitive) that the two conditions from Section 4.3.3 are satisfied for the selected fetch rules. The search space for the example query includes many more query plans deploying different combinations of fetch rules, such as a combination of fetch rules $\emptyset \Rightarrow \text{country}$ and $\text{country} \Rightarrow \text{language, capital}$ (Figure 3.3), or a combination of fetch rules $\text{language} \Rightarrow \text{country, capital}$, $\text{country} \Rightarrow \text{language}$ and $\text{country} \Rightarrow \text{capital}$.

4.4 Enumeration Algorithm

We now consider how to efficiently enumerate query plans in the search space and apply the cost estimation algorithm to find the predicted best query plan. As briefly discussed in Sections 4.1.1 and 4.1.3, Deco's specific setting invalidates some key assumptions behind enumeration in traditional query optimizers:

- In Deco, different query plans corresponding to the same algebraic representation may produce different query results: the fetch rules selected in a query plan determine the valid instance over which the query result is evaluated. Thus, estimated cardinality is a property of individual query plans rather than algebraic representations, violating a typical assumption exploited by extensible query optimizers such as Volcano [30] and Cascades [29].
- Deco's cardinality estimation is holistic: the cardinality and cost of a subplan depend in part on the rest of the plan. Thus, a bottom-up enumeration and cost-estimation strategy as in the System R optimizer [51] is not applicable to Deco.

Procedure 1 FindBestPlanNaive(ast)

```

1 bestPlan ← NULL
2 minCost ← ∞
3 for each joinTree do
4   for each fetchRuleSelection do
5     plan ← BuildPlan(ast, joinTree, fetchRuleSelection)
6     if plan.isValid() then
7       plan.EstimateCard(0, ∅)
8       cost ← plan.EstimateCost()
9       if cost < minCost then
10        bestPlan ← plan
11      end if
12    end if
13  end for
14 end for
15 return bestPlan

```

Given these constraints, we needed to devise a plan enumeration algorithm for Deco that generates complete physical plans in the search space while maximizing reuse of common subplans. This strategy makes Deco's query optimizer slower than traditional query optimizers for queries with similar complexity; however, we will see in Section 4.5 that the optimization time tends to be insignificant compared to typical query execution time in Deco.

In this section we first describe a naive exhaustive enumeration, then we describe a more efficient version of the same strategy. In Section 4.5.2 we compare the performance of the more efficient enumeration against the naive one.

4.4.1 Naive Enumeration

In its simplest form, Deco's plan enumeration algorithm generates all query plans in the search space and applies the cost estimation algorithm to each plan to find the predicted best plan. The simple FindBestPlanNaive procedure in Procedure 1 illustrates the entire query optimization process at a high level.

To enumerate all possible join trees (line 3), we use a straightforward recursive algorithm based on the properties of join trees from Section 4.3.1. Given a join

tree, we build a plan with each possible fetch rule selection satisfying Condition 2 in Section 4.3.3 (lines 4–5). In this step, The BuildPlan procedure places the query’s predicates so that they are evaluated as early as possible (recall Section 4.3.2). Finally, we retain those plans satisfying Condition 1 in Section 4.3.3 (line 6), and choose the plan with the least estimated cost (lines 7–11).

4.4.2 Efficient Enumeration

The FindBestPlanNaive procedure in Procedure 1 handles each alternative plan independently, so it may perform some redundant computation across iterations of the inner loop (lines 5–12). Specifically, the EstimateCard procedure in Section 4.2.1 may be called multiple times on the same subplans with the same arguments for target and preds. Since those calls produce the same result (card and cards), we “memo-ize” [30]: the first call computes and stores the result, and subsequent calls reuse it. (Even if the order of predicates in parameter preds in a subsequent call is different from the original order of predicates used to compute the stored result, we can reuse the result by shuffling cards appropriately; recall the definitions of preds and cards in Section 4.2.1.)

It turns out that we can maximize reuse of common subplans by storing only one plan at a time (with its cardinality estimates), if we iterate over the alternative plans in a particular order. This order is determined based on two properties. First, given a left (outer) subplan S , all complete plans containing the subplan S must appear consecutively in the order, so that we can reuse the subplan S and its estimated cardinality. Moreover, we further order those plans containing the subplan S by the first raw table T joining with S so that the Resolve operator corresponding to T can reuse the estimated cardinality of the relevant existing data. Note that these properties apply recursively to different layers of subplans. As a result, we enumerate physical plans by alternately selecting raw tables and their corresponding fetch rules. Note in contrast the naive enumeration first selects a join tree and then selects a fetch rule for each raw table in the tree.

4.5 Experimental Evaluation

In this section we present our experimental evaluation of Deco’s query optimizer. We first evaluate the accuracy of our cost estimation algorithm in a variety of settings. Then, we evaluate the efficiency of our plan enumeration algorithm.

4.5.1 Accuracy of Cost Estimation

To evaluate the accuracy of Deco’s cost model, we compare estimated costs against actual costs for three different scenarios: no existing data in the raw tables (Experiment 1), existing data with fetch prioritization (Experiment 2), and existing data with little effect of fetch prioritization (Experiment 3). We use a variety of queries and plans, explained as we describe each experiment.

Experiment 1: No Existing Data For this experiment we considered the actual costs of executing our most common example query using Amazon Mechanical Turk:

```
SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES 8
```

Recall from the experiment reported in Section 3.6.1 that starting with empty raw tables, the actual costs of the basic plan, the reverse plan, and the hybrid plan (Figure 4.1) were \$12.05, \$2.30, and \$1.35, respectively, when the cost of each fetch was set to \$0.05 for all fetch rules.

For cost estimation, we set the selectivity factor of predicate `language='Spanish'` to 0.1. Also, we set the selectivity factors of resolution functions `dupElim` and `majority-of-3` to 1.0 and 0.4, respectively. Given these settings, Deco’s cost estimation algorithm produces estimated costs of \$15.00, \$2.40, and \$1.40 for the basic, reverse, and hybrid plans, respectively. Figure 4.4 shows the estimated and actual costs for the three query plans. The different portions of the bars show the costs incurred by the different fetch rules in each setting. Even though our overall estimated costs were reasonably close to the actual costs with a mean percentage

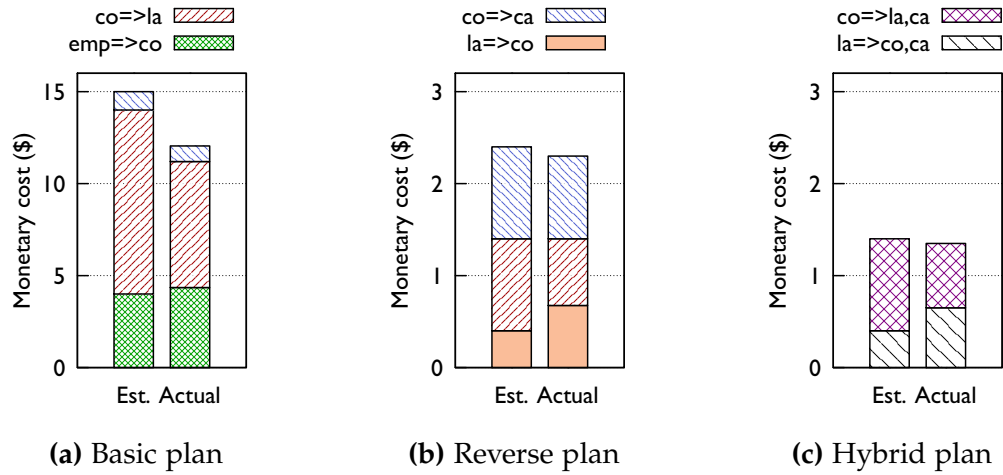


Figure 4.4: Accuracy of cost estimation (Experiment 1)

error of 11%, estimated costs for individual Fetch operators were less accurate. We now explain why.

First, our selectivity factor settings were not accurate enough. Using the basic plan, for example, resolution function *majority-of-3* did not need the third input value as often as expected, and we observed an actual selectivity factor of 0.47. Also, we ended up collecting 64 distinct countries with 87 answers, translating to actual selectivity factors of 0.74 and 0.125 for resolution function *dupElim* and predicate language='Spanish', respectively. In general, selectivity factors for the different operators in a particular plan have different impact on the accuracy of estimated cost. As an example, the cost of the reverse plan is most sensitive to the selectivity of *majority-of-3*: a difference of 0.1 in the selectivity translates to about 20% error in the estimated cost.

Second, our cardinality estimation algorithm makes some simplifying assumptions when handling reverse fetch rules. For example, we assume new countries obtained using language \Rightarrow country satisfy predicate language='Spanish', but in reality workers may return values that do not satisfy this predicate. We could remove this assumption from the Fetch.EstimateCard procedure, if we are given a "selectivity factor" for the fetch rule. In addition, a reverse fetch rule inserts new tuples into some dependent tables, thus effectively decreasing the additional number of required values to resolve the dependent attributes. To address this

problem, we could adjust the selectivity factors of those resolution functions that have interactions with reverse fetch rules.

Experiment 2: Existing Data, Fetch Prioritization For this experiment we considered two queries with different initial states of the raw tables so that fetch prioritization takes effect. First, Figure 4.5 shows the estimated and actual costs of obtaining X result tuples for the following simple query, varying X on the x-axis.

```
SELECT country, language, capital
FROM Country
MINTUPLES X
```

We seeded anchor table CountryA with 100 different country names, and the two dependent tables CountryD1 and CountryD2 with 0, 100, and 200 randomly chosen tuples (across both tables) for Figures 4.5a, 4.5b, and 4.5c, respectively. Missing language and capital values in the partial result were completed using fetch rules $\text{country} \Rightarrow \text{language}$ and $\text{country} \Rightarrow \text{capital}$, respectively.

We used the actual costs reported in Experiment 3 in Section 3.6.3, obtained using our less sophisticated heuristics for fetch prioritization (recall $score_1$ from Section 3.5.2). Note those actual costs were obtained using our crowd simulator, which responds to fetch requests by selecting values from a predetermined set. Since each data point was the average of ten trials, experiments using human workers would have incurred significant latency and dollar cost.

For estimated costs, recall from Section 4.2.1 that our cost model includes a configurable weight α to emulate the effectiveness of fetch prioritization, with a larger α indicating better prioritization. We computed the estimated costs using α values of 0.6, 0.75, and 0.9, to empirically determine a good value for α . In addition, we assume that statistics about the existing data available in the back-end RDBMS are accurate, so that we can evaluate Deco's cost model in isolation. Overall, our cost estimates are reasonably close to the actual costs: With $\alpha=0.75$, we observed mean absolute percentage errors of 6.7%, 12.4%, and 44.4%, for Figures 4.5a, 4.5b, and 4.5c, respectively. Note the large error for Figure 4.5c is mainly due to the 100% error for the $X \leq 30$ data points.

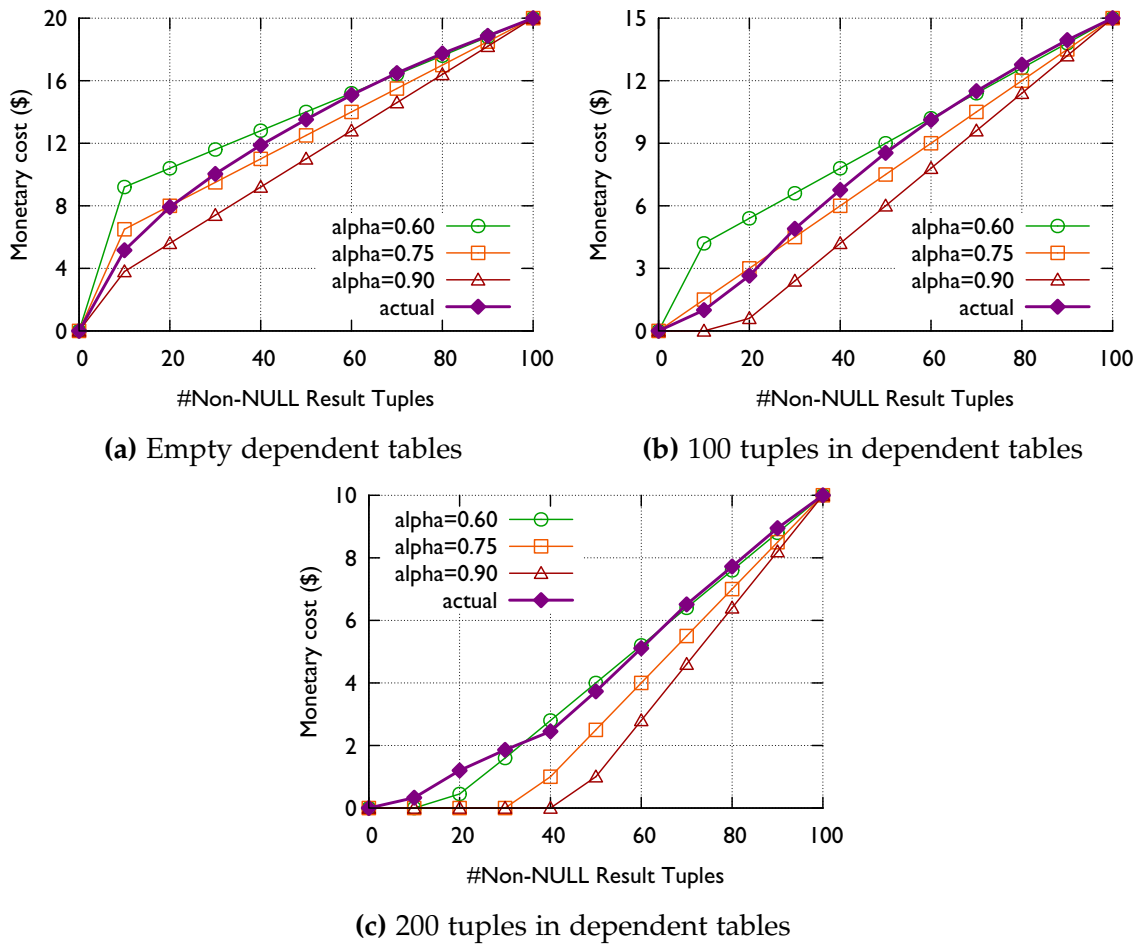


Figure 4.5: Accuracy of cost estimation (Experiment 2, part 1)

Now we consider a second query. Figure 4.6 shows the actual and estimated costs of obtaining X result tuples for the following join query:

```
SELECT city, country, population, language
FROM Country, City
WHERE City.country = Country.country
MINTUPLES X
```

For Figures 4.6a and 4.6b, we seeded anchor tables CountryA and CityA with two real datasets: the 100 largest European cities and the 200 largest cities in the world. For Figure 4.6c, we seeded the anchor tables with a synthetic dataset

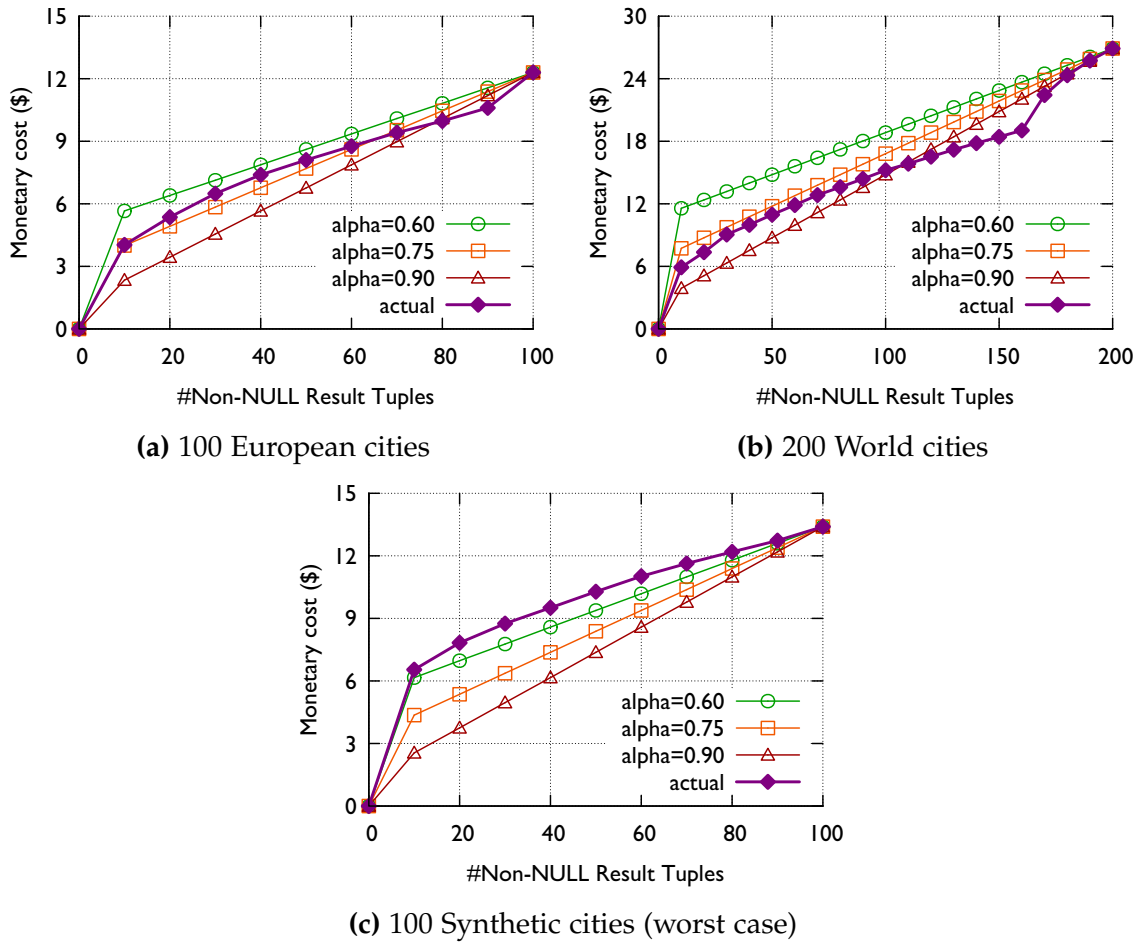


Figure 4.6: Accuracy of cost estimation (Experiment 2, part 2)

to demonstrate the worst case. All dependent tables were initially empty and populated on-demand using fetch rules $\text{country} \Rightarrow \text{language}$ (for relation Country) and $\text{city, country} \Rightarrow \text{population}$ (for relation City). Again we used the actual costs reported in Experiment 4 in Section 3.6.3 that were obtained using our less sophisticated heuristics, from the crowd simulator.

In Figures 4.6a and 4.6b, our cost estimates are reasonably accurate as in Figure 4.5: With $\alpha=0.75$, the mean absolute percentage errors are 4.8% and 11.1% for Figures 4.6a and 4.6b, respectively. In Figure 4.6c, we deliberately generated a synthetic dataset to make fetch prioritization work as poorly as possible. Hence, the estimated costs (even with α of 0.6) are smaller than the actual costs for the

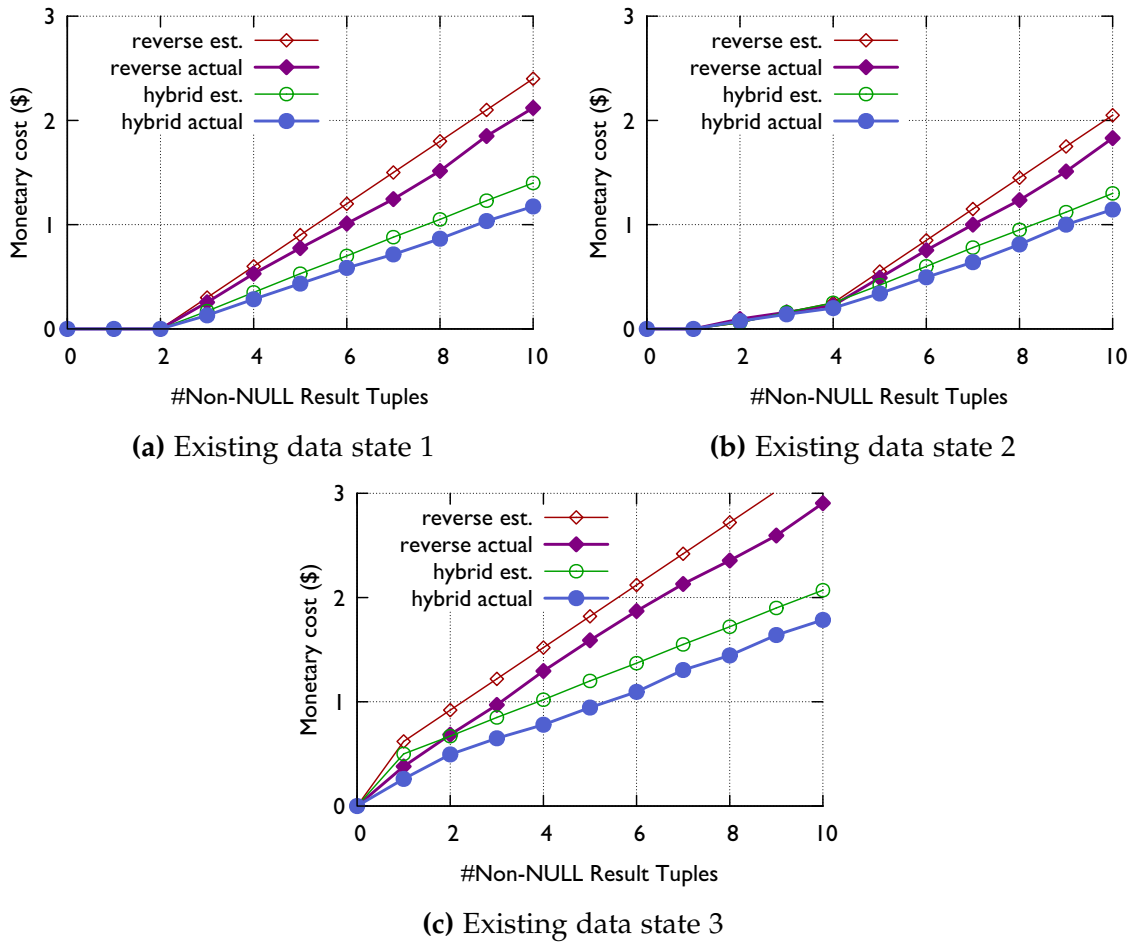


Figure 4.7: Accuracy of cost estimation (Experiment 3, part 1)

entire range of X . This result is consistent with the definition of α in Section 4.2.1: a smaller α means less effective prioritization.

Based on our experiments, we believe by setting $\alpha=0.75$ as a rule of thumb, our cost model will produce acceptable estimates for many cases. Note that the best value for α depends on the heuristic approach for fetch prioritization; we would need to use a larger α for some more sophisticated heuristics described in Section 3.5.

Experiment 3: Existing Data, No Fetch Prioritization Now we consider a similar scenario but without fetch prioritization. Figure 4.7 shows the actual and estimated costs of obtaining X result tuples for the following query.

```

SELECT country, capital
FROM Country
WHERE language='Spanish'
MINTUPLES X

```

We started the query with three different initial states of the raw tables, resulting in the three graphs in Figure 4.7. For each initial state, we measured the actual costs of executing the reverse and hybrid plans, with our crowd simulator set to correspond to the selectivities observed using Amazon Mechanical Turk in Experiment 1. (Note the simulator flips a coin to produce data, and we report the average of ten trials.)

For cost estimation, we used the same selectivity setting as in Experiment 1. Overall, our estimated costs were reasonably accurate across all three initial states and both plans, with a mean absolute percentage error of 18.6%. This result is comparable to Experiment 1 (no existing data) and implies that our cost model is able to distinguish between existing data versus new data.

We consider a second query. Figure 4.8 shows the actual and estimated costs of obtaining X result tuples for the following query:

```

SELECT capital, population, language
FROM Country, City
WHERE City.country = Country.country AND City.city = Country.capital
MINTUPLES X

```

Again we started the query with three different initial states of the raw tables, resulting in the three graphs in Figure 4.8. For each initial state, we executed two query plans corresponding to the following join trees:

- Plan 1: (((CountryA \bowtie CountryD2) \bowtie CityA) \times CountryD1) \bowtie CityD1
- Plan 2: (((CityA \times CountryA) \bowtie CountryD2) \bowtie CountryD1) \bowtie CityD1

Both plans uses the following set of fetch rules:

- [Country] $\emptyset \Rightarrow$ country, country \Rightarrow language, and country \Rightarrow capital
- [City] $\emptyset \Rightarrow$ city,country and city,country \Rightarrow population

Note despite the simplicity of the original query, these plans are 5-way joins over the raw tables. For Plan 2, we set the crowd simulator to produce capital cities

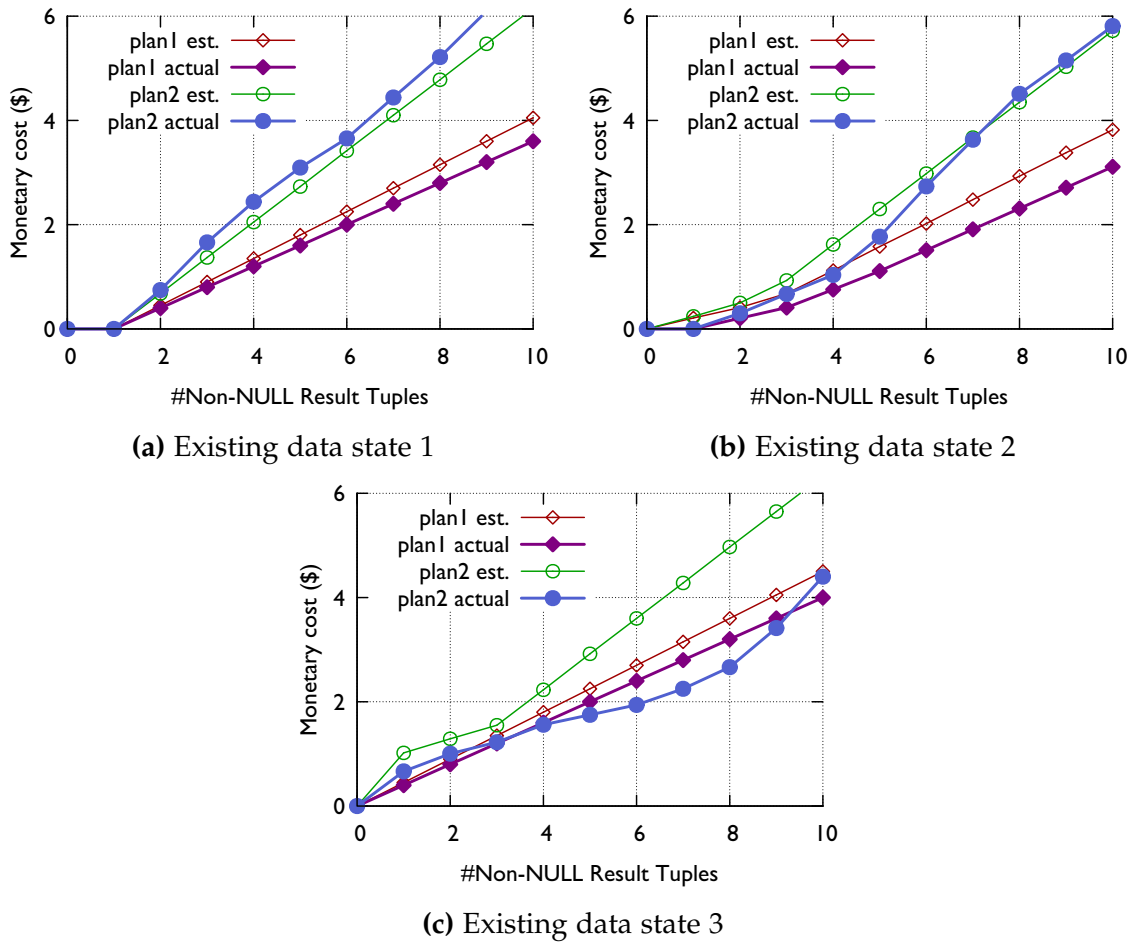


Figure 4.8: Accuracy of cost estimation (Experiment 3, part 2)

with a probability of 0.3 (for fetch rule $\emptyset \Rightarrow \text{city, country}$), and also used the corresponding selectivity setting for cost estimation.

In Figures 4.8a and 4.8b, our cost estimates are reasonably close to the actual costs with a mean absolute percentage error of 20.3%. Figure 4.8c illustrates a case where our cost model often fails to predict the better plan: We populated relation City only with capital cities, resulting in a large discrepancy between the actual selectivity of join predicate $\text{City.city}=\text{Country.capital}$ and the provided selectivity setting.

Summary In Experiments 1–3 we evaluated the accuracy of Deco’s cost estimation by comparing estimated costs against actual costs for a wide variety of

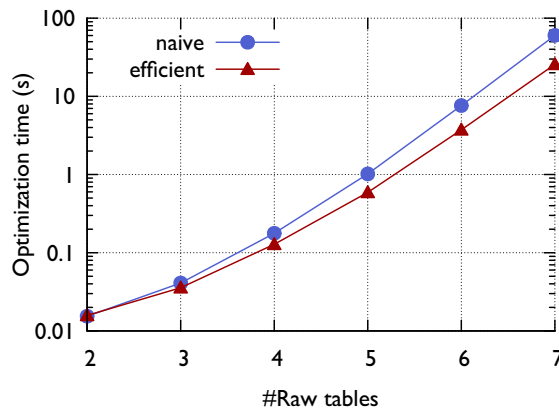


Figure 4.9: Efficiency of plan enumeration

scenarios. Overall we observed that the estimated costs were fairly close to the actual costs, with mean absolute percentage errors of 20% or less in most cases we considered. We can safely conclude that a query plan chosen by Deco’s query optimizer will be relatively inexpensive, if not the cheapest plan, for a given query.

4.5.2 Efficiency of Plan Enumeration

To evaluate the efficiency of plan enumeration, we compare the efficient enumeration from Section 4.4.2 against the naive enumeration from Section 4.4.1, in terms of the overall optimization time, i.e., time taken to find the predicted best plan given a query.

Experiment 4 Because the size of the search space depends heavily on the number of raw tables m , we generated queries with a varying number of raw tables based on their From clauses. For each query, we created a set of fetch rules so that each raw table has exactly one candidate fetch rule. (Here we used a conservative setting: more fetch rules make the efficient enumeration even faster than the naive one.) Figure 4.9 shows the optimization times for the naive and efficient enumeration in logarithmic scale, for $m=2..7$. Not surprisingly the efficient enumeration performs better than the naive enumeration for the entire range of m values. Moreover, the percent improvement tends to increase as m increases, because the amount of redundant computation also increases. With $m=7$, the

efficient enumeration is 2.35 times faster than the naive enumeration.

4.6 Related Work

There has been a large body of previous work addressing query optimization in traditional database systems [15]. Sections 4.1 and 4.4 elaborated on several key differences between traditional query optimizers and Deco’s query optimizer in plan costing and enumeration. Since Deco’s cost model exploits the available statistics in the back-end RDBMS to estimate cardinality of relevant existing data, Deco can of course benefit from any sophisticated techniques for improved statistics management and cardinality estimation [14, 16, 48].

Also related is prior work on query optimization over diverse data sources in the context of heterogeneous or federated database systems [18, 23, 32]. In some sense, Deco’s overall architecture is analogous to federated database systems: Deco’s query processor, fetch procedures, and the crowd correspond to a mediator, wrappers, and data sources, respectively. However, as far as the query optimization problem is concerned, all of the fundamental differences between Deco and traditional database systems that we have described in this chapter also apply when comparing Deco and federated database systems.

4.7 Conclusion

We presented Deco’s query optimizer that finds the best plan to answer a query in terms of estimated monetary cost. To reflect Deco’s query semantics and plan execution strategies, we incorporated several novel techniques into the query optimizer:

- Estimating the cost of executing a given query plan requires us to distinguish between free existing data versus paid new data. Our cost model takes into account the amount of relevant existing data to estimate the amount of new data required to produce the result.
- Deco’s cardinality estimation needs to cope with changes to the database

state during query execution. To do so, our cardinality estimation algorithm simultaneously estimates cardinality and the final state of the database using a top-down recursive process.

- To find the best plan efficiently, plan enumeration must avoid redundant computation across different query plans in our search space, which is challenging in Deco's specific setting. Our plan enumeration algorithm maximizes reuse of common subplans to the extent possible.

Experiments showed that estimated costs produced by our cost estimation algorithm are accurate enough to successfully find inexpensive query plans for a wide variety of settings. We also validated that our more sophisticated plan enumeration algorithm performs much better than naive enumeration.

Coupled with Deco's query execution engine described in Chapter 3, Deco's query processor as a whole provides a complete solution for answering a Deco query while minimizing monetary cost and reducing latency.

Chapter 5

Design and Implementation of CrowdFill

We now present our *CrowdFill* system, a separate system from Deco (Chapters 2–4) that implements the table-filling approach introduced in Chapter 1. Recall that the table-filling approach enables all participating workers to collaboratively complete a partially-filled table by filling in empty cells and voting on data entered by other workers. Our goal is to collect high-quality data from the crowd, while adhering to constraints on the collected data, capping total monetary cost, and keeping latency low. CrowdFill addresses the data quality, monetary cost, and latency objectives primarily with its voting scheme, compensation scheme, and real-time synchronization scheme, respectively, all of which are covered in this chapter.¹

In Section 5.1, we describe a formal model for CrowdFill that defines table states, primitive operations on tables, and constraints on the collected data. In Section 5.2, we describe CrowdFill’s overall architecture, then explain its components. In Section 5.3, we describe how CrowdFill guides workers toward a final table satisfying prespecified constraints while providing an intuitive data-entry interface. In Section 5.4, we describe how CrowdFill distributes total monetary

¹The material presented in this chapter first appeared in [46, 47].

budget across the workers who participated in data collection. Section 5.5 describes our experimental evaluation, Section 5.6 covers related work, and we conclude in Section 5.7.

5.1 Formal Model

We begin in Sections 5.1.1 and 5.1.2 by defining CrowdFill’s model for tables and primitive operations on tables. We specify the notion of a partially-filled *candidate table* during data collection, the operations that can be performed on candidate tables, and how a *final table* is computed from a candidate table. In Section 5.1.3 we add *constraints* to the model, so CrowdFill users can provide a minimum required number of rows, and can specify restrictions on the collected data values. Finally, in Section 5.1.4 we cover issues related to the fact that workers operate on a table concurrently. We explain how worker actions are transmitted via messages to a central server and other workers, and how conflicts are handled. We prove a strong *convergence* theorem, stating that when all work ceases, all copies of the candidate and final tables are guaranteed to have the same value.

5.1.1 Table Specification

Recall from Section 1.1 that we consider the relational data model, which represents structured data as a two-dimensional table: each row in the table typically corresponds to a data element or a fact, while each column is a property or attribute of the element or fact. Also, throughout the chapter we say “user” to denote the entity (human or application) wishing to perform data collection, as distinct from a crowdsourced “worker”, who performs tasks contributing to the data collection goal.

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

- **Column definitions:** A column name, data type, and optionally a domain (set of allowed values) for each column.

- **Primary key:** One or more key columns that together should uniquely identify each row in the final table. By default, all columns together are a key, i.e., there should be no duplicate rows in the final table.

As a running example, suppose we are interested in collecting information about soccer players who appeared at least 80 times in international matches (“caps”). We use the following schema:

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

Columns name and nationality together are the primary key.

Scoring function: To ensure the quality of collected data, our model allows workers to provide upvotes and downvotes on data in each row. To aggregate votes, the user provides a *scoring function* $f(u_r, d_r)$ where u_r and d_r denote an upvote count and a downvote count, respectively, for a given row r . The intention is for a higher score to indicate that the row is more likely to be correct. Specifically, we take the meaning of score ranges as follows:

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

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

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

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

5.1.2 Table State and Primitive Operations

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

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

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

Candidate table: A *candidate* table R is a set of rows, where each row r is annotated with its upvote count u_r and downvote count d_r . The candidate table can be modified by performing one of the following *primitive operations*:

- $\text{insert}(r)$: Insert a new empty row r into R , with $u_r = d_r = 0$.
- $\text{fill}(r, A, v)$: Fill in an empty column A in row $r \in R$ to have value v .
- $\text{upvote}(r)$: Upvote a complete row $r \in R$. Increment u_q for each row $q \in R$ whose value is equal to the value of row r .
- $\text{downvote}(r)$: Downvote a partial row $r \in R$. Increment d_q for each row $q \in R$ whose value is equal to or a superset of the value of row r .

Note upvoting an incomplete row r is not allowed: Filling in more columns in row r could invalidate the intention of the upvote, since we do not know whether the new combination of values is acceptable. On the other hand, downvoting an incomplete row r is allowed: Filling in more columns does not invalidate that the subset combination of values was unacceptable. Since upvote and downvote operations semantically approve or disapprove of the values in row r (or a part of the row) rather than row r itself, we automatically propagate each vote to all other rows as specified above. We will see in Sections 5.2 and 5.3 that worker actions correspond to fill, upvote, and downvote operations, while insert operations are issued only by the system, to control the number of empty rows in the table.

In our example SoccerPlayer table, here is one possible candidate table. Symbols \uparrow and \downarrow indicate upvote and downvote counts, respectively. Note in particular that candidate tables need not have unique rows with a given primary key; keys are enforced in the final table, defined next.

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

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

Based on our example candidate table above and majority-of-three-or-more scoring function (Section 5.1.1), we obtain the following final table:

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

Note the five incomplete rows in the candidate table are omitted, while Beckham is omitted because the score for the row is zero.

5.1.3 Constraints

We now describe *constraints*, which enable CrowdFill users to specify restrictions on the final table of collected data. *Cardinality constraints* specify that the final table must contain a minimum number of rows. *Values constraints* specify that rows with certain values or combinations of values must be present in the final table. *Predicates constraints*, not yet implemented in the CrowdFill system, specify that values in the final table must satisfy certain conditions. We will see that

cardinality constraints can be considered a special case of values constraints, and values constraints are a special case of predicates constraints. Nevertheless, from both a user and system perspective, it is useful to distinguish the three concepts.

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

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

Values constraint: In many cases a user or application may wish to start with a partially-filled table, using the crowd to complete the missing data. A common example is to have a set of values for the keys (soccer player names and nationalities, for example), and use the crowd to fill in missing values (position, caps, goals). Another case is when the table has a set of rows, but the user wishes to crowdsource additional rows. For this type of scenario, the user can specify a set T of “initial” rows, which we refer as *template* rows. Template rows can be complete, meaning they should also be present in the final table; they can be partial, with workers expected to fill in missing values; and they can be empty, in which case they are specifying how many additional rows are needed. Given the latter case, we see that cardinality constraints are in fact a special case of values constraints.

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

For each row $t \in T$, there exists a unique row $s \in S$ such that $\bar{s} \supseteq \bar{t}$, i.e., the values in row s are equal to or a superset of the values in row t .

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

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

name	nationality	position	caps	goals
		FW		
	Brazil			
	Spain			

Note the final table in Section 5.1.2 satisfies the values constraint with this template.

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

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

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

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

name	nationality	position	caps	goals
		=‘FW’		≥ 30
	=‘Brazil’			≥ 30
	=‘Spain’		≥ 100	

Note the final table in Section 5.1.2 satisfies the predicates constraint with this template.

5.1.4 Concurrent Operations

As briefly described in Section 1.1.2, CrowdFill shows an up-to-date version of the evolving table to each participating worker. The workers make changes to the table using the primitive operations defined in Section 5.1.2. In this section, we first explain the client-server structure of the system, and specify a “vote history” that is needed to maintain consistency. Then we explain how operations at one client are propagated to other versions of the table, and how they are applied when they arrive. Lastly, we show formally that our execution model handles concurrency elegantly: We prove a theorem stating that when the system quiesces, all copies of the table are identical.

Execution overview: The CrowdFill system consists of *clients* (the workers’ web browsers) and a *server* to which all clients are connected. We assume that message deliveries between the server and clients are reliable and in-order. The server has a *master* copy of the candidate table, and each client has its own copy, which is initially identical to the master copy. Suppose the worker at client C performs an operation op . Client C applies op to its own copy of the table, then sends a corresponding *message* m to the server. Once the server receives message m from client C , it first processes m on the master table, then forwards m to all clients except C . Finally all clients except C receive m and process m on their copies of the table. Details of message generation, and the application of operations and messages to a table, are covered momentarily.

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

Applying locally-generated operations: Suppose the worker at client C performs a primitive operation op . Let R_C denote C ’s local copy of the candidate table. Also, let UH_C and DH_C denote C ’s upvote and downvote histories, respectively.

For each operation type, client C applies op locally and sends a corresponding message m to the server as follows:

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

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

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

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

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

How the Model Supports Concurrency

Our model was designed carefully so workers can operate independently, performing operations on copies of the same table with conflicts resolved in an intuitive and seamless fashion. It is easy to see (and formally proven below) that insert and voting operations can be performed on independent copies and propagate to the other clients without conflict. Thus, the only source of potential conflict is when two different workers fill in empty values in the same row at the same time—either for the same column or for different columns.

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

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

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

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

name	nationality	position	caps	goals	↑	↓
		FW			0	0

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

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

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

Convergence

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

When clients concurrently generate and process messages according to the specifications above, the server and all clients process each generated message once and only once. However, the server and each client may process the messages in a different order. Despite this difference, our convergence theorem guarantees that the server and all clients always have the same candidate table whenever the system “quiesces” (i.e., all generated messages are propagated and processed).

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

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

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

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

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

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

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

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

Let R_A and R_B denote the sets of rows in candidate tables at two different locations (either two clients, or the server and one client), after processing all messages in M on the initial table R_0 . By Lemma 1, to prove $R_A = R_B$, it suffices to

show that there is no $r \in R_A$ such that $r \notin R_B$. Suppose, for sake of contradiction, that there exists $r \in R_A$ such that $r \notin R_B$.

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

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

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

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

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

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

Proof of convergence theorem (part 2): Since the server and all clients process the same set of upvote and downvote messages in M , they all end up with the same UH and DH . Therefore, by Lemma 3, they all have the same upvote and downvote counts after all messages in M are processed. \square

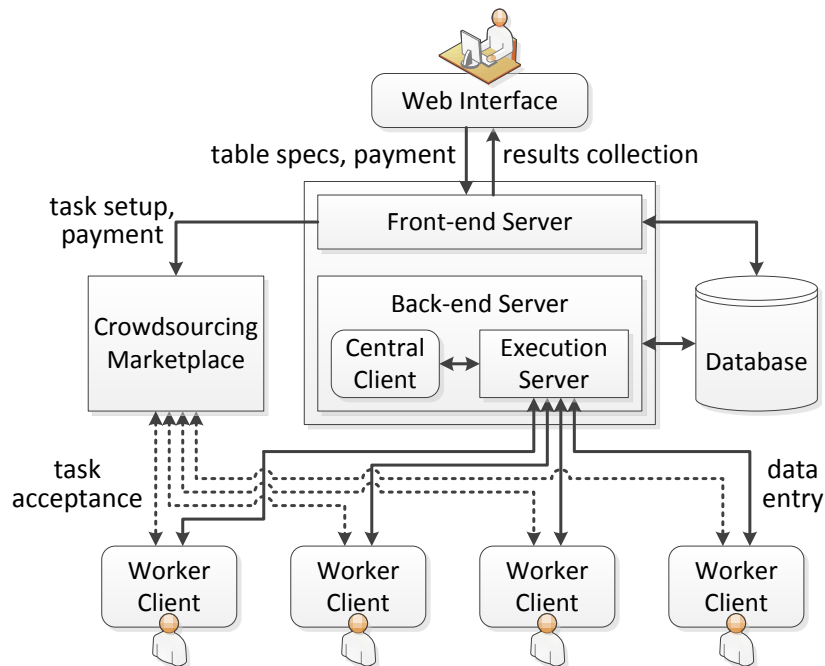


Figure 5.1: CrowdFill architecture

5.2 System Overview

We have implemented a fully-functional prototype of the CrowdFill system. The system is based on the formal model of Section 5.1, although as expected a number of additional system-oriented details were needed. This section provides an overview of the CrowdFill system. Some of the most complicated aspects—constraint-satisfaction and worker compensation—are covered in detail in Sections 5.3 and 5.4.

5.2.1 Architecture

Figure 5.1 shows the overall architecture of the CrowdFill system. It consists of several major components: a web interface for users, a front-end server, a back-end server, and one or more worker clients. It also connects with one or more crowdsourcing marketplaces (only one is shown in our diagram), and a database. In the course of data collection, these components interact with each other as

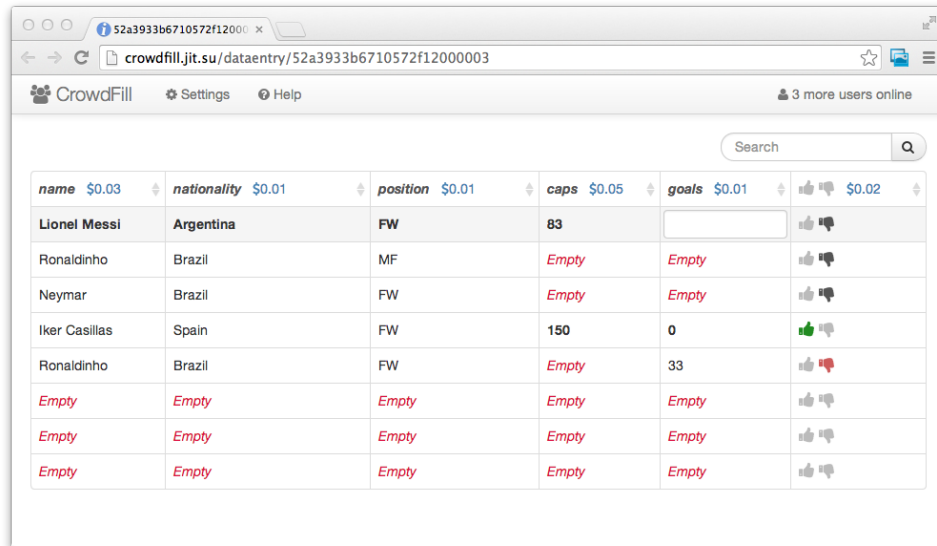
key	column name	type	possible values
<input checked="" type="checkbox"/>	name	Text	regular expression (optional)
<input checked="" type="checkbox"/>	nationality	Text	regular expression (optional)
<input type="checkbox"/>	position	Choice	<input type="checkbox"/> FW <input type="checkbox"/> MF <input type="checkbox"/> DF <input type="checkbox"/> GK
<input type="checkbox"/>	caps	Number	0 <input type="text"/> maximum value (optional)
<input type="checkbox"/>	goals	Number	0 <input type="text"/> maximum value (optional)

Figure 5.2: Table schema editor

follows:

1. Using the web interface, a user sends a table specification to the front-end server to launch data collection. Figure 5.2 shows CrowdFill's table schema editor.
2. The front-end server creates one or more tasks in the crowdsourcing marketplace. (The number of tasks is marketplace dependent. So far we have used Amazon Mechanical Turk [1] exclusively.)
3. Each worker accepting a task is redirected to the back-end server and establishes a bidirectional persistent connection to the back-end server.
4. Workers perform actions through their data entry interfaces (see Figure 5.3), until the back-end server determines that enough data has been collected.
5. Using the web interface, the user retrieves collected data from the front-end server and pays workers through the crowdsourcing marketplace.

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



The screenshot shows a web browser window with the URL `crowdfill.jit.su/dataentry/52a3933b6710572f12000003`. The interface includes a search bar and a table with the following data:

name \$0.03	nationality \$0.01	position \$0.01	caps \$0.05	goals \$0.01	thumbs up/down \$0.02
Lionel Messi	Argentina	FW	83	<input type="text"/>	thumbs up/down
Ronaldinho	Brazil	MF	Empty	Empty	thumbs up/down
Neymar	Brazil	FW	Empty	Empty	thumbs up/down
Iker Casillas	Spain	FW	150	0	thumbs up/down
Ronaldinho	Brazil	FW	Empty	33	thumbs up/down
Empty	Empty	Empty	Empty	Empty	thumbs up/down
Empty	Empty	Empty	Empty	Empty	thumbs up/down
Empty	Empty	Empty	Empty	Empty	thumbs up/down

Figure 5.3: Data entry interface

5.2.2 Front-end Server

The CrowdFill server logically consists of a front-end server interacting with applications, and a back-end server interacting with workers. The front-end server provides applications with the CrowdFill REST API, which supports creating, updating, and deleting table specifications (including table schemas, scoring functions, and constraint templates), controlling the actual data collection, and retrieving collected data. Using the CrowdFill API, we built a web-based graphical user interface. The server stores all metadata and collected structured data in a MongoDB [5] database.

The front-end server communicates with one or more crowdsourcing marketplaces to attract workers (to the back-end server) and to pay those workers once enough data is collected. Although we have only used Amazon Mechanical Turk for the marketplace, additional marketplaces can easily be added, as long as those marketplaces allow us to host questions “externally” and make “bonus” payments to workers, similar to Amazon Mechanical Turk [1].

5.2.3 Back-end Server

The back-end server corresponds to the server in CrowdFill’s formal model described in Section 5.1.4: It maintains the master copy of the candidate table and broadcasts each incoming message to all clients except the one originating the message. We built the server using Node.js [8]; for connections between the back-end server and clients, we chose the Socket.IO library [9] so that most web browsers could participate as worker clients.

In addition to the basic functionality described as part of the formal model, the back-end server is responsible for populating a candidate table, determining the amount of monetary compensation for each worker, and storing a complete trace of worker actions for bookkeeping. We will discuss these aspects in Sections 5.3 and 5.4.

5.2.4 Worker Client

Each worker client provides its worker with a data entry interface running in a web browser. Through this interface, workers can perform three kinds of actions: fill, upvote, and downvote. These actions correspond to the primitive operations from Section 5.1 with the same names, with some restrictions on vote operations mentioned below. Note worker clients never generate insert operations. For now suppose that there are enough incomplete rows in the candidate table; we will discuss this issue further in Section 5.3.

Fill action: As shown in Figure 5.3, the main part of this interface is an HTML table. This table shows an up-to-date local copy of the candidate table, and it allows workers to fill in empty cells in-place. (In this regard, this interface bears much similarity to online spreadsheets such as Google Docs [4] spreadsheet.) Filling in an empty cell generates a fill operation as described in Section 5.1.4. To encourage workers to fill in different parts of the table, each client randomizes the order of rows in the local copy of the table presented to the worker.

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

Although the formal model in Section 5.1 does not prevent a single worker from contributing multiple upvotes and/or downvotes to the same row, the CrowdFill data entry interface intentionally prohibits this behavior: each worker may provide, directly or indirectly, at most one vote for each row. (Implementing this behavior requires maintaining a log of worker identifiers and votes.) Thus, upvote and downvote counts represent the number of different workers who approve or disapprove of a given set of values. To further enforce this semantics, when a worker provides the last value that completes a row, that worker automatically upvotes the row, without additional payment. Also, a single worker may not upvote more than one row with the same primary key. Finally, CrowdFill provides a feature allowing users to set a maximum number of votes per row, to prevent excessive voting.

5.3 Satisfying the Constraints

Recall from Section 5.1.3 that our overall goal is to obtain a final table satisfying the *cardinality* and *values constraints*. Also recall that cardinality constraints are a special case of values constraints: we can include in the values constraint template T additional empty rows when the original template rows are fewer than the cardinality constraint. In the rest of this section, we assume cardinality constraints are absorbed by the values constraint in this fashion. As a reminder, a values constraint with template T says: for each template row $t \in T$, there exists a unique row s in the final table such that $\bar{s} \supseteq \bar{t}$.

To guide the final table towards the template, and to minimize wasted work, the CrowdFill system only allows new rows to be inserted into the candidate table by a special client, which we call *CC* (for “Central Client”), in the back-end server (Figure 5.1). With this approach, workers never need to add rows, and they need not be aware of the constraints, allowing them to simply fill in empty values in existing rows, and cast votes.

5.3.1 Probable Rows Invariant

The overall objective of the special client CC, as it adds rows to the candidate table, is to keep the table in a state where filling in empty values might produce a final table satisfying the constraint. We first define the notion of a row being *probable*—informally, given the current state of the candidate table, a probable row may eventually contribute to the final table. Based on the derivation of a final table from a candidate table as defined in Section 5.1.2, we say that row r is probable if it satisfies one of the following conditions:

1. Row r contains empty values for some primary key columns and has a zero score from its upvote and downvote counts. (Recall from Section 5.1.1 the score is computed by $f(u_r, d_r)$.)
2. Row r contains no empty values for the primary key columns and has a zero score, but no other row with the same primary key has a positive score.
3. Row r is a complete row with a positive score, and no other row with the same primary key has a greater score. If there are other rows with the same primary key and an equal score, only one row in the group is probable, and we assume ties are broken deterministically.

Through special client CC, the CrowdFill system maintains the following invariant at all times, based on the values constraint.

Probable Rows Invariant (PRI): Each template row t corresponds to a unique probable row r in the candidate table such that $\bar{r} \supseteq \bar{t}$.

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

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

Proof: By the third condition in the definition of probable rows above, along with the final table derivation from Section 5.1.2, every complete probable row in R is

in S . Since every $p \in P'$ is a complete row, we have $P' \subseteq S$. By PRI on R and T , each $t \in T$ corresponds to a unique $p \in P'$ such that $\bar{p} \supseteq \bar{t}$. From $P' \subseteq S$, each $t \in T$ corresponds to a unique $s \in S$ such that $\bar{s} \supseteq \bar{t}$. Therefore, the final table satisfies the values constraint. \square

5.3.2 Maintaining the Invariant

We now explain how the CrowdFill system maintains the PRI. Initially client CC populates its candidate table R with the set T of template rows. In addition, client CC upvotes all complete template rows, as if those rows were completed by workers (recall Section 5.2.4). CC's initialization operations—as well as later operations—are propagated to the server and all other clients, as if CC were a worker client.

After initialization, all rows in R have nonnegative scores since no downvotes have been cast. In addition, recall from Section 5.1.3 that there does exist a final table satisfying the values constraint; this assumption implies that no two rows in T with complete values for the primary key columns have the same key. Thus, all rows in R are probable, and the PRI trivially holds.

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

As workers perform actions, graph G changes as the set P of probable rows changes. Client CC incrementally computes a maximum bipartite matching for G after each change in G . Whenever a change in G makes the size of a maximum bipartite matching for G smaller than $|T|$, client CC inserts probable rows into R so that the maximum bipartite matching contains exactly $|T|$ edges, as follows. When a template row $t \in T$ becomes “free” (i.e., does not appear in the matching), client

CC performs a breadth-first search from t to a free probable row in P to find an augmenting path (i.e., a path between two free nodes where edges are alternately in and out of the matching). In the worst case, this BFS takes $O(|P||T|)$ time; however, if no probable rows can be connected to two or more template rows with distinct values (e.g., the template has only empty rows, or all primary keys are completely filled in), it takes $O(|P|)$ time.

By Berge's theorem [11], if an augmenting path is found, we can increase the size of the bipartite matching back to $|T|$. Otherwise, maintaining the PRI requires inserting another row q into R . Usually we can use \bar{t} for the value of the new row q , in which case an edge between t and q is added to the matching. However, inserting row q with value \bar{t} does not always make q probable. Let us see what can go wrong. First, value \bar{t} might have been downvoted by several workers, giving potential row q a negative score. This case usually indicates the template row t has incorrect values. Second, value \bar{t} may have all primary key columns filled in, while there is already a probable row with the same primary key and a higher score. In either case, client CC first attempts to shuffle the matching so that another template row $t' \in T$ becomes free. If CC cannot find another probable template row t' , to maintain the PRI CC has no option but to remove t from T , perhaps violating the user's original intention. When this case occurs, our system continues data collection with the reduced template; it might instead be appropriate to raise an error and abort data collection, depending on the user's preference.

5.3.3 Probable Rows Invariant Maintenance Example

Suppose we have a values constraint with the template T from Section 5.1.3, which we repeat here labeling the template rows a , b , and c :

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

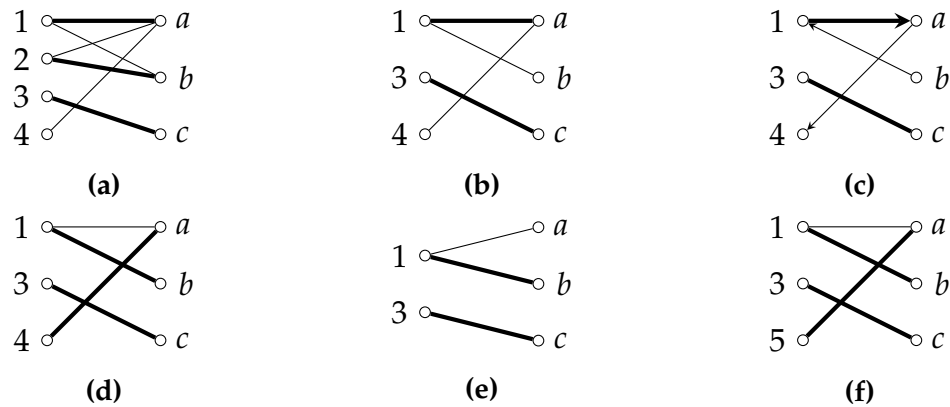


Figure 5.4: Bipartite graph representation of the PRI maintenance

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

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

Recall from Section 5.1.1 that the scoring function $f(u_r, d_r)$ for our example is $u_r - d_r$ if $u_r + d_r \geq 2$, and 0 otherwise. Thus all four rows above are probable, i.e., $P = \{1, 2, 3, 4\}$. Figure 5.4a shows the bipartite graph G corresponding to P and T . The maximum bipartite matching incrementally maintained by client CC is denoted by the thick edges in the graph—three edges in Figure 5.4a.

Now suppose row 2 is downvoted by one more worker, so its score becomes -2. Figure 5.4b shows graph G after row 2 is removed from P . Since the template row b is now free, client CC starts a breadth-first search from row b and finds an augmenting path $b-1-a-4$ (Figure 5.4c). Based on this path, CC updates the matching to include $4-a$, $1-b$, and $3-c$ (Figure 5.4d). In this case, the PRI is maintained without inserting an additional row into R .

Now suppose a worker fills in Messi’s caps as 82, replacing row 4 with row 4’ shown below. Further suppose row 4’ is downvoted by two workers, so it is removed from P (Figure 5.4e). Although the template row a is free, there is no

augmenting path starting from row a . Thus CC inserts the value of row a into R as row 5, adding $5-a$ to the maximum bipartite matching (Figure 5.4f). The resulting candidate table is now:

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

5.4 Compensating Workers

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

5.4.1 Challenges and Approach

In paid crowdsourcing, workers are motivated by monetary compensation, and typically have a desire to maximize their total earnings. On the user side, our challenge is an example of the *cost-latency-quality* tradeoffs discussed in [58]: We want to exploit workers’ desire to earn money in order to obtain a final table of high quality, without too much cost or latency. We describe how CrowdFill’s compensation scheme addresses this challenge.

Roughly, our overall scheme is based on compensating workers for those data

entries that actually contribute to the final table, directly or indirectly.² Under this scheme (combined with row-wise voting), a worker entering correct value has a much better chance of getting paid than one entering incorrect data. Since populating a candidate table as described in Section 5.3 minimizes wasted work, this scheme does not penalize “good” workers. Moreover, this scheme makes the overall monetary cost more predictable.

CrowdFill allows a user to simply specify a total monetary budget. When the table is complete, the system calculates the final compensation for each worker based on how and when they contributed to the table. Our approach also can take into account variability in the difficulty of providing values for different columns, and the fact that entering new key values can get progressively more difficult as the table fills up. In addition to final payment, it is necessary during data collection to provide estimated monetary value for each action. Although we use a relatively simple approach, experimentally our simple approach produces estimates that are reasonably close to the actual compensation for each worker; see Section 5.5.

We first detail how final compensation is calculated after the table is complete in Section 5.4.2. Compensation estimation during data collection is covered in Section 5.4.3.

5.4.2 Allocating Total Budget to Workers

Suppose the user specified a total monetary budget B , and CrowdFill has obtained a final table S satisfying the constraints. The back-end server stores a complete trace of worker actions in terms of a set M of messages received from all worker clients (as in Section 5.1.4), where each message in M is uniquely timestamped and annotated with the worker identifier originating the message. Note messages from special client CC (Section 5.3) are not included in M . Our goal is to determine overall compensation for each worker who participated in data collection,

²Note that rejecting “incorrect” answers in Amazon Mechanical Turk has been problematic, since doing so lowers workers’ approval rates, which may affect their qualifications to work on other tasks. Since CrowdFill makes “bonus” payments to workers rather than using the default per-task payments, workers’ approval rates are not affected by our scheme.

given B and M .

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

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

We elaborate Steps 1–5 below. Step 6 is straightforward.

Defining the Notion of Contribution

We specify which messages in M contributed to the final table S . Recall from Sections 5.1.4 and 5.2.4 that worker clients send three types of messages to the back-end server:

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

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

Direct contribution of replace: Consider a cell $c \in C$ that corresponds to column A of row $s \in S$, hereafter called $s.A$. The $\text{replace}(r, q, \bar{q})$ message in M that directly

contributes to cell c is the one that filled in the A value in the row that eventually became row s . Formally, $\text{replace}(r, q, \bar{q})$, generated by a $\text{fill}(r, A, v)$ operation, directly contributes to a cell $c \in C$ for $s.A$ if there exists a series of $k \geq 1$ messages, $\text{replace}(r_0, r_1, \bar{r}_1)$, $\text{replace}(r_1, r_2, \bar{r}_2), \dots, \text{replace}(r_{k-1}, r_k, \bar{r}_k)$ in M , such that $r_0 = r$, $r_1 = q$, and $r_k = s$.

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

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

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

Contribution of upvote: An $\text{upvote}(\bar{r})$ message in M directly contributes to a given row $s \in S$ if it increased the upvote count of row s , i.e., $\bar{r} = \bar{s}$. One exception is the case when the upvote message was automatically sent by a fill action that completed a row by providing the last value (Section 5.2.4), which we do not count as a separate contribution. Note each upvote message can contribute to at most one row in S , since S does not have duplicate rows due to the primary key constraint.

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

Allocation Schemes

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

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

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

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

There are many possible ways to determine the weights, including asking the user to designate weights as part of the table specification. In our system, we automatically choose the weights using the trace of worker actions stored in M . Our goal is to set y_i ($1 \leq i \leq m$), y_\uparrow , and y_\downarrow to the median times taken for workers to generate replace messages for filling in A_i , upvote messages, and downvote messages, respectively, that contribute to the final table. (Using medians rather than

averages makes the weights more resilient to outliers.) We use the difference of timestamps in two consecutive messages from the same worker as the time taken for generating the second message, but we are aware of its flaws; to address this limitation, we need to incorporate a more sophisticated mechanism into the client side.

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

The dual-weighted allocation scheme addresses this problem by assigning progressively higher weights to the cells for the primary key columns. Specifically, for each primary key column A_i , we assign linearly increasing weights from $(1 - z_i)y_i$ to $(1 + z_i)y_i$, to all cells in C_i in the order that their values first appeared in column A_i of the candidate table: Compensation for the cell containing the k -th value in key column A_i is $b_k = (1 + \frac{2z_i}{|C_i|-1}(k - \frac{|C_i|+1}{2}))y_iB/Y$. Compensation for the other cells, upvotes, and downvotes remains the same as in column-weighted allocation.

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

Splitting Cell Compensation

We allocate the portion of B assigned to each cell $c \in C$ to the one or two replace messages contributing to c . Given a cell $c \in C$, let b_c denote the portion of B assigned to c . Let h_c be a “splitting factor” between 0 and 1, discussed momentarily. We allocate $h_c b_c$ to the message directly contributing to c , and $(1 - h_c)b_c$ to the message indirectly contributing to c , if any. (Note as discussed above some

cells may not have any indirectly contributing messages, so this scheme does not necessarily exhaust B .)

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

5.4.3 Estimating Compensation

As described in Section 5.4.2, CrowdFill pays workers who participate in data collection based on their contribution to the final table S . Now consider the worker perspective. To keep workers engaged during data collection, it is necessary to provide estimated monetary compensation for each action. This estimation problem is quite challenging on its own, and a complete exploration is beyond the scope of this thesis. Here we describe our intuitive initial approach, which we evaluate empirically in Section 5.5. Note our approach treats all workers as equally likely to perform useful actions; if we kept track of worker's past performance we could adjust our estimates accordingly.

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

Now we describe how CrowdFill estimates compensation for each action, for

the three allocation schemes specified in Section 5.4.2.

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

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

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

5.5 Experimental Evaluation

In this section we present our experimental evaluation of the CrowdFill system. First, we briefly describe and assess the overall effectiveness of CrowdFill’s table-filling approach for a data-collection task. Then we focus on our compensation schemes, comparing the three allocation schemes from Section 5.4.2, as well as measuring the accuracy of estimated compensation from Section 5.4.3.

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

We deployed the CrowdFill front-end and back-end servers on Nodejitsu [7], a Node.js [8] hosting platform in the cloud; CrowdFill ran on a single “drone” (an individual unit of computing power). CrowdFill’s database was hosted on MongoLab [6], a database-as-a-service provider for MongoDB [5].

Schema and constraints: We used the running example SoccerPlayer schema from Section 5.1, with an additional date-of-birth (dob) column:

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

Recall from Section 5.1.1 that our scoring function implements a “majority of three or more” voting scheme.

Our goal was to collect information about 20 soccer players with caps between 80 and 99 inclusive, starting from an empty table. (Players with caps ≥ 100 were excluded because their information is readily available from the FIFA Century Club [3].)

Overall effectiveness: We report results for a representative run that collected data using five human workers. Note that results may vary significantly based on the workers participating in an experiment; drawing meaningful conclusions

would require a large number of trials using different sets of workers. In our representative run it took 10 minutes 44 seconds to obtain a final SoccerPlayer table with 20 complete rows. At completion, the candidate table contained 23 rows; two rows were downvoted twice or more, and one extra row was added by a conflict. In this run, all 20 final rows were accurate, although we occasionally observed inaccurate rows in other runs.

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

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

Accuracy of estimated compensation: Now we evaluate whether workers earned what they expected, based on the estimates CrowdFill provided during data collection. Figure 5.5 shows actual and estimated compensation in our representative run for each of the five workers. The middle bar for each worker represents the sum of estimates shown when actions were performed. These raw estimates were reasonably close to the actual compensation across all five workers, with a mean absolute percentage error of 16.1%.

As discussed in Section 5.4.3, the estimated compensation for each action is calculated assuming the action will eventually contribute to the final table. If a worker consistently provides incorrect values, the estimation error can be arbitrarily large. The rightmost bar for each worker shows the sum of estimates only for

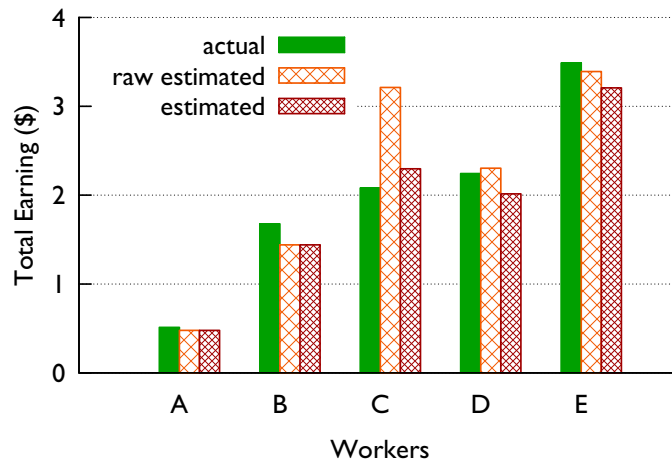


Figure 5.5: Accuracy of estimated compensation

those actions that contributed to the final table. Using these corrected estimates, we observed a mean absolute percentage error of 9.9%.

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

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

Another comparison we can make across compensation schemes is “stability,” i.e., whether workers earn at a steady rate throughout the table-filling task. Figure

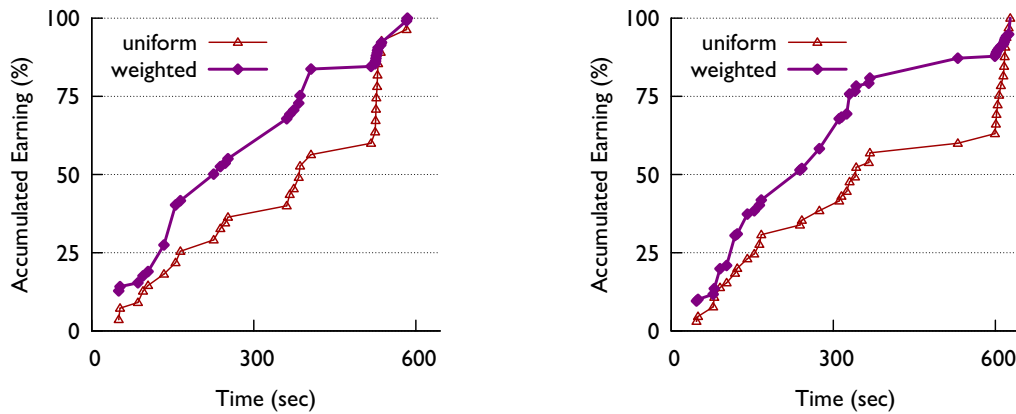


Figure 5.6: Earning rates for uniform and weighted allocation, two workers

5.6 shows earning rates of two representative workers during our representative run. The x-axis represents elapsed time from the start of data collection, while the y-axis represents accumulated earning as percentage of the eventual total. Thus, the slope of a line in the graph corresponds to the earning rate. We plot the actual earning rates using dual-weighted allocation (which, recall, is equivalent to column-weighted allocation in this run), along with what the earning rates would have been under uniform allocation. We can see in this experiment that weighted allocation appears to be somewhat more stable than uniform allocation in terms of earning rate. More extensive experiments would be needed to fully understand earning rates in a wide variety of settings.

5.6 Related Work

Real-time cooperative editing systems [4, 24, 57] allow multiple clients to concurrently edit the same evolving document. Like CrowdFill, these systems have a notion of convergence, and of intention preservation; they incorporate a technique called “operational transformation” to maintain those properties. CrowdFill’s specific setting of structured data, and potentially inconsistent data entries, led us to the formal model in Section 5.1, which handles potentially-conflicting operations intuitively and seamlessly.

There has been a large body of prior work on various aspects of monetary compensation in crowdsourcing environments. Many papers, e.g., [39, 50, 52], have studied specific compensation schemes to elicit desired behaviors from workers. Reference [52] compared several different compensation schemes in terms of accuracy of worker answers; CrowdFill’s compensation scheme is most analogous to “reward-agreement”, which rewards workers for answers agreeing with the majority. Another important problem in crowdsourcing marketplaces is task pricing. While reference [35] describes a method for estimating a worker’s reservation wage (i.e., the lowest wage a worker is willing to accept), more recent studies [53, 54] propose an alternative approach where workers indicate their desired wage by submitting a bid. Pursuing these directions may allow CrowdFill to improve its allocation scheme, with an aim of minimizing total monetary cost without a prespecified budget.

5.7 Conclusion

We presented CrowdFill, a system for collecting structured data from the crowd, using the table-filling approach. CrowdFill explores a quite different approach from Deco, which is microtask-based and was covered in Chapters 2–4. CrowdFill’s formal model defines a concise set of primitive operations as workers contribute values and votes on a shared table, and it seamlessly resolves conflicts caused by concurrent operations while providing data consistency across clients. CrowdFill allows the specification of value and cardinality constraints on the collected data, and guides data collection towards the constraints while providing an intuitive data-entry interface. CrowdFill’s compensation scheme distributes a specified monetary budget to workers in a way that rewards those worker operations that contribute to the final result, while encouraging useful work along the way by providing estimated compensation. Our experiments show that CrowdFill’s table-filling approach provides an effective means of collecting structured data from the crowd. We also validated that CrowdFill’s compensation scheme

provides final compensation commensurate with workers' efforts, and offers reasonably accurate estimated compensation during data collection. In the next chapter, we will summarize key differences and tradeoffs between CrowdFill's table-filling approach and the microtask-based approach of Deco and other systems [26, 38].

Chapter 6

Summary and Future Work

6.1 Summary of Contributions

This thesis studied the problem of crowdsourcing structured data. We explored two complimentary approaches to the problem, and for each approach we developed a complete prototype system designed and implemented in a principled manner. We summarize the main contributions of the thesis:

- **Foundations of Deco.** In Chapter 2 we provided an overview of Deco, a system for “declarative crowdsourcing” that uses the microtask-based approach for data collection. After describing the Deco data model and the syntax and semantics of Deco’s query language, we explained Deco’s overall architecture and major components.
- **Query Execution in Deco.** In Chapter 3 we presented Deco’s query execution engine, explaining in detail how our approach to query execution minimizes monetary cost and reduces latency when executing a given query plan. To answer Deco queries correctly and efficiently, we developed a hybrid execution model, which respects Deco semantics while enabling parallel access to the crowd. Query execution using this hybrid model bears as much similarity to incremental view maintenance as to a traditional iterator model. The query execution engine also uses a sophisticated prioritization scheme for fetching data from the crowd, to minimize monetary cost.

- **Query Optimization in Deco.** In Chapter 4 we presented Deco’s query optimizer, which finds the best plan to answer a query in terms of estimated monetary cost. To reflect Deco’s query semantics and plan execution strategies, we developed a cost model distinguishing between free existing data versus paid new data, a cardinality estimation algorithm coping with changes to the database state during query execution, and a plan enumeration algorithm maximizing reuse of common subplans. Coupled with Deco’s query execution engine, Deco’s query processor as a whole provides a complete solution for answering a Deco query that minimizes monetary cost, then reduces latency.
- **Design and Implementation of CrowdFill.** In Chapter 5 we presented CrowdFill, an alternative system for crowdsourcing structured data that uses the table-filling approach. CrowdFill implements a formal model that enables real-time collaboration among workers, and that resolves conflicts in an intuitive fashion. CrowdFill allows the specification of constraints on the collected data, and guides data collection towards the constraints while providing an intuitive data-entry interface. CrowdFill’s compensation scheme distributes a specified monetary budget to workers in a way that rewards those worker operations that contribute to the final result, while encouraging useful work along the way.

6.2 Contrasting Our Two Approaches

As discussed in Section 1.1.3, each of our two approaches has advantages and disadvantages. The microtask-based approach is more naturally scalable, and more suitable for combining humans and computers, than the table-filling approach. On the other hand, the table-filling approach enables workers to enter data in a more transparent, active, and flexible manner than the microtask-based approach.

In addition to the general differences discussed above and in Section 1.1.3, there are a number of practical differences between Deco and CrowdFill:

- To express data collection goals, Deco users pose SQL queries, while CrowdFill users rely on constraint templates.
- In terms of compensation, Deco users specify a fixed monetary cost for each fetch rule, and Deco attempts to minimize total monetary cost; CrowdFill users specify total monetary budget, which the system distributes across workers.
- To keep latency low, Deco relies on maximizing parallelism to the extent possible, while CrowdFill supports real-time collaboration that bypasses crowdsourcing marketplaces.
- Although both Deco and CrowdFill specify minimum thresholds for data quality (using resolution functions and scoring functions, respectively), Deco primarily relies on asking redundant questions while CrowdFill uses its row-wise voting scheme.

6.3 Future Work

We first discuss some areas of future work for Deco and CrowdFill in Sections 6.3.1 and 6.3.2, respectively. Then in Section 6.3.3 we conclude with future directions in the general area of crowdsourcing structured data.

6.3.1 Deco

We see the following three major directions for Deco:

- **More general SQL constructs.** Although Deco's data model and query language as defined in Chapter 2 are general enough to support any SQL queries, the current Deco system only supports Select-Project-Join queries. A natural next step is to extend the system to support more general SQL queries beyond Select-Project-Join; for example, order-by and group-by are certainly useful SQL constructs in any environment.
- **Alternatives to MinTuples.** Deco's functionality would be enhanced by supporting alternatives to MinTuples, such as MaxCost and MaxTime, as described

in Section 2.2. These alternatives enable end-users to maximize the number of result tuples subject to the specified monetary or time budget. Since MaxCost and MaxTime do not specify a desired property of valid instances explicitly, our query execution and optimization strategies developed around MinTuples would need to be revisited.

- **Adaptive query processing techniques.** Another important avenue of future work is to incorporate adaptive query processing techniques [37] into the Deco prototype. Due to the simplified statistical information about crowd-sourced data and the long-running nature of Deco queries, the “optimize-then-execute” paradigm [19] may not always yield the best possible execution strategy in our setting.

6.3.2 CrowdFill

We see the following four major directions for CrowdFill:

- **More general constraints.** While the cardinality and values constraints in the current CrowdFill system are sufficient for many scenarios, the predicates constraint defined in Section 5.1.3 would add significant power in specifying restrictions on final tables.
- **More sophisticated compensation schemes.** Although we have shown empirically that our current compensation scheme works reasonably well, much of it is ad-hoc based on intuition; further study is needed to fully understand and refine the current scheme.
- **Protection against spammers.** An extremely important area of investigation is the potential effect of spammers in our system, i.e., workers trying to obtain payment for insincere work. Our compensation scheme discourages incorrect answers, but the transparent nature of our table-filling approach may enable spammers to hinder data collection, both individually and collectively, and to steal credit by copying potentially correct answers from other workers. A full understanding of the potential for such actions in CrowdFill, and how to inhibit or eliminate them through modifications to the compensation scheme

and/or data-entry interface, is a significant challenge for the future.

- **More comprehensive experimental evaluation.** Our experimental evaluation reported in Section 5.5 is preliminary; more comprehensive evaluation might provide valuable insights into the CrowdFill system and the table-filling approach in general. As a first step, larger-scale evaluations would be valuable, including larger table sizes, more concurrent workers, and a variety of data domains. Unfortunately, as with much work in crowdsourcing, the process of conducting large-scale empirical evaluations in a realistic setting can be expensive and time-consuming.

6.3.3 Crowdsourcing Structured Data

In this thesis we focused on the microtask-based approach and the table-filling approach individually, and provided one specific implementation for each approach. Given that we now have fully-functional prototypes of Deco and CrowdFill, we can envision a thorough comparison of the two approaches, largely through empirical studies, considering all three axes of cost, latency, and quality. Finally, as crowdsourcing marketplaces and platforms mature and offer more functionality, we anticipate that other complementary or improved approaches and implementations for crowdsourcing structured data may emerge in the future.

Bibliography

- [1] Amazon Mechanical Turk. <http://mturk.com/>.
- [2] Amazon Mechanical Turk Developer Sandbox. <https://requestersandbox.mturk.com/>.
- [3] FIFA century club (men). http://www.fifa.com/mm/document/fifafacts/stats-centclub/52/00/59/centuryclub100314_neutral.pdf.
- [4] Google Docs. <http://docs.google.com/>.
- [5] MongoDB. <http://www.mongodb.org/>.
- [6] Mongolab. <http://www.mongolab.com/>.
- [7] Nodejitsu. <http://www.nodejitsu.com/>.
- [8] Node.js. <http://www.nodejs.org/>.
- [9] Socket.IO. <http://socket.io/>.
- [10] O. Alonso, D. E. Rose, and B. Stewart. Crowdsourcing for relevance evaluation. *SIGIR Forum*, 42, 2008.
- [11] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842–844, 1957.
- [12] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

- [13] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowd-searcher. In *WWW*, 2012.
- [14] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 2002.
- [15] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.
- [16] S. Chaudhuri and V. R. Narasayya. Automating statistics management for query optimizers. In *ICDE*, 2000.
- [17] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [18] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, 2002.
- [19] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
- [20] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Declarative platform for data sourcing games. In *WWW*, 2012.
- [21] D. J. Dewitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.
- [22] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4):86–96, 2011.
- [23] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in a heterogeneous dbms. In *VLDB*, 1992.
- [24] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.
- [25] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.

- [26] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, 2011.
- [27] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems: The Complete Book*. Pearson Education, 2009.
- [28] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.
- [29] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.
- [30] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.
- [31] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won? dynamic max discovery with the crowd. In *SIGMOD*, 2012.
- [32] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.
- [33] P. Hansen. Methods of nonlinear 0-1 programming. *Annals of Discrete Mathematics*, 5:53–70, 1979.
- [34] J. Hellerstein and J. Naughton. Query execution techniques for caching expensive methods. In *SIGMOD*, 1996.
- [35] J. Horton and L. Chilton. The labor economics of paid crowdsourcing. In *EC*, 2010.
- [36] S. R. Jeffery, L. Sun, M. DeLand, N. Pendar, R. Barber, and A. Galdi. Arnold: Declarative crowd-machine data integration. In *CIDR*, 2013.
- [37] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
- [38] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

- [39] W. Mason and D. J. Watts. Financial incentives and the “performance of crowds”. In *HCOMP*, 2009.
- [40] A. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *SIGMOD*, 2012.
- [41] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: Declarative crowdsourcing. In *CIKM*, 2012.
- [42] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.
- [43] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. An overview of the deco system: data model and query language; query processing and optimization. *SIGMOD Record*, 41(4):22–27, 2012.
- [44] H. Park, A. Parameswaran, and J. Widom. Query processing over crowd-sourced data, <http://ilpubs.stanford.edu:8090/1052/>. Technical report, Stanford InfoLab, 2012.
- [45] H. Park and J. Widom. Query optimization over crowdsourced data. *PVLDB*, 6(10):781–792, 2013.
- [46] H. Park and J. Widom. CrowdFill: A system for collecting structured data from the crowd. In *WWW*, 2014.
- [47] H. Park and J. Widom. CrowdFill: Collecting structured data from the crowd. In *SIGMOD*, 2014.
- [48] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.
- [49] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD*, 1990.

- [50] O. Scekic, H. L. Truong, and S. Dustdar. Incentives and rewarding in social computing. *Communications of the ACM*, 56(6):72–82, 2013.
- [51] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.
- [52] A. D. Shaw, J. J. Horton, and D. L. Chen. Designing incentives for inexperienced human raters. In *CSCW*, 2011.
- [53] Y. Singer and M. Mittal. Pricing mechanisms for crowdsourcing markets. In *WWW*, 2013.
- [54] A. Singla and A. Krause. Truthful incentives in crowdsourcing tasks using regret minimization mechanisms. In *WWW*, 2013.
- [55] R. Snow, B. O’Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast - but is it good? evaluating non-expert annotations for natural language tasks. In *EMNLP*, 2008.
- [56] A. Sorokin and D. Forsyth. Utility data annotation with amazon mechanical turk. In *CVPRW*, 2008.
- [57] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.
- [58] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, 2012.
- [59] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *CHI*, 2004.
- [60] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

- [61] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.
- [62] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.