

# Optimization Techniques for Human Computation-enabled Data Processing Systems

by

Adam Marcus

S.M., Massachusetts Institute of Technology (2008)

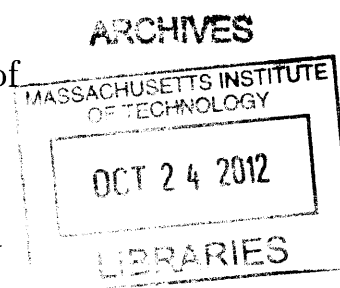
B.Sc., Rensselaer Polytechnic Institute (2006)

Submitted to the Department of  
Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2012



© Massachusetts Institute of Technology 2012. All rights reserved.

Author .....

Department of  
Electrical Engineering and Computer Science  
August, 10 2012

Certified by .....

Samuel R. Madden  
Associate Professor  
Thesis Supervisor

Certified by .....

David R. Karger  
Professor

Thesis Supervisor

Accepted by .....

Leslie A. Kolodziejski

Chair, Department Committee on Graduate Students



# Optimization Techniques for Human Computation-enabled Data Processing Systems

by  
Adam Marcus

Submitted to the Department of  
Electrical Engineering and Computer Science  
on August, 10 2012, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

## Abstract

Crowdsourced labor markets make it possible to recruit large numbers of people to complete small tasks that are difficult to automate on computers. These marketplaces are increasingly widely used, with projections of over \$1 billion being transferred between crowd employers and crowd workers by the end of 2012. While crowdsourcing enables forms of computation that artificial intelligence has not yet achieved, it also presents crowd workflow designers with a series of challenges including describing tasks, pricing tasks, identifying and rewarding worker quality, dealing with incorrect responses, and integrating human computation into traditional programming frameworks.

In this dissertation, we explore the systems-building, operator design, and optimization challenges involved in building a crowd-powered workflow management system. We describe a system called Qurk that utilizes techniques from databases such as declarative workflow definition, high-latency workflow execution, and query optimization to aid crowd-powered workflow developers. We study how crowdsourcing can enhance the capabilities of traditional databases by evaluating how to implement basic database operators such as sorts and joins on datasets that could not have been processed using traditional computation frameworks. Finally, we explore the symbiotic relationship between the crowd and query optimization, enlisting crowd workers to perform selectivity estimation, a key component in optimizing complex crowd-powered workflows.

Thesis Supervisor: Samuel R. Madden  
Title: Professor

Thesis Supervisor: David R. Karger  
Title: Professor





## Acknowledgments

If ever there was a group of people so diverse in their qualities but all consistent in their support throughout the years, it would be my thesis committee. Sam Madden, on the path to helping me think like a systems-builder, taught me most importantly to build something cool. It was with this mindset that he supported the outrageous idea of embedding humans in database systems. Were it not for having him as a constant and dependable advisor with whom I could speak about anything standing in the way of research, graduate school would have been a different world. David Karger got me to think about the big questions around a contribution while at the same time digging deeply into experimental design to make sure we were answering the right questions. He also provided a healthy level of distraction from research with the other aspects of life that matter, be they dancing or journalism. Rob Miller taught me how to answer questions once a human enters the loop, not only adding breadth to my work, but also making it more interesting. I am grateful for how freely he welcomed me into his group, and treated my growth like that of any of his other students.

Whereas a good advisor is supportive and provides direction, it is your peers that are ever-present and inspirational to work with every day. I was lucky to have research collaborators at MIT that taught me as we explored together. Dan Abadi taught Kate Hollenbach and I to be ambitious about research, and relentless once we find a topic. Kate gave me a taste of the creative world outside of building fast systems, a lesson that means more today than it did back then. Michael Bernstein, in addition to meaningfully introducing me to the world of Human-Computer Interaction, is a wonderful person with whom it was a pleasure to explore graduate school. Ted Benson, as an “engineer’s engineer,” introduced me to the mental tool that most adroitly got me through the rest of my research: the faster you can build or write a first version, the faster you can understand your contributions and improve a second version. Osama Badar impressed me with his constant growth, and taught me to teach along the way. And finally, Eugene Wu made for an amazing friend and collaborator on the projects that came to define my dissertation. Sometimes I am unable to tell whether Eugene and I are doing research or learning about some new fact of life together, and I wouldn’t have it any other way.

I am thankful for the participants in our Amazon Mechanical Turk experiments, who put up with many mistakes and taught us a little bit more about how the world works.

Aside from the collaborators with whom I worked directly, many officemates, friends, and members of the Database, Haystack, and User Interface Design Groups made my graduate school experience what it was. In alphabetical order, I am grateful for Sibel Adali, Sihem Amer-Yahia, Eirik Bakke, Leilani Battle, Eugene Brevdo, Daniel Bruckner, Carrie Cai, Harr Chen, Alvin Cheung, James Cowling, Philippe Cudre-Mauroux, Carlo Curino, Dorothy Curtis, Marcia Davidson, Matthew Ezovski, Max Goldman, Theodore Golfinopoulos, Philip Guo, Stavros Harizopoulos, Fabian Howahl, George Huo, David Huynh, Evan Jones, Eunsuk Kang, Harshad Kasure, Juho Kim, Michel Kinsy, Margaret Leibovic, Tom Lieber, Greg Little, Patrice

Macaluso, Nirmesh Malviya, electronic Max, Sheila Marian, Yuan Mei, Barzan Mozafari, Daniel Myers, Neha Narula, Debmalya Panigrahi, Katrina Panovich, Meelap Shah, David Shoudy, Aysha Siddique, Vineet Sinha, Robin Stewart, Mike Stonebraker, Boleslaw Szymanski, Arvind Thiagarajan, Stephen Tu, Andrew Tibbetts, Richard Tibbetts, Ben Vandiver, Manasi Vartak, Jean Yang, Mohammed Zaki, Yang Zhang, Chien-Hsiang Yu, and Sacha Zyto.

Without my parents, none of this would be. My mother and father shaped me as a human being. They planted the seeds for my ability to ask questions and attempt to answer them using science, math, and machines. I hope they are as proud of me as I am grateful to them.

Meredith Blumenstock has made my life increasingly meaningful and bright. I can not imagine a better partner with whom to share the future.

While people are the most important part of my life, two organizations made my and many other researchers' contributions possible. I am grateful to have been supported by a National Science Foundation Graduate Research Fellowship and a National Defense Science and Engineering Graduate Fellowship.

# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
1.1	Crowd Programming Challenges . . . . .	16
1.2	Crowd Workflow Management: The Dark Ages . . . . .	18
1.3	Qurk: A Crowd-powered Workflow System . . . . .	18
1.4	Reusable Operations: Sorts and Joins . . . . .	19
1.5	Query Optimization and Work Quality . . . . .	19
1.6	Ethics and New Employment Models . . . . .	20
1.7	Multiple Authors and Prior Publications . . . . .	20
1.8	The Use of “We” Instead of “I” . . . . .	21
<b>2</b>	<b>Related Work</b>	<b>23</b>
2.1	Studying the Crowds . . . . .	23
2.2	Applications of Crowds . . . . .	24
2.3	Crowdsourced Databases . . . . .	25
2.4	Crowd Programming Models . . . . .	26
2.5	Worker and Result Quality . . . . .	27
<b>3</b>	<b>Qurk Query and Data Model</b>	<b>29</b>
3.1	Data Model . . . . .	29
3.2	Pig Latin-based Query Language . . . . .	30
3.2.1	Generative Tasks . . . . .	31
3.2.2	Sorts . . . . .	32
3.2.3	Joins and Feature Extraction . . . . .	34
3.2.4	Aggregates . . . . .	36
3.3	Workflow Example . . . . .	36
<b>4</b>	<b>Architecture and Query Execution</b>	<b>39</b>
4.1	Architecture . . . . .	39
4.2	Implementation Details . . . . .	40
4.3	HIT Generation . . . . .	41
<b>5</b>	<b>Human-powered Sorts and Joins</b>	<b>43</b>
5.1	Introduction . . . . .	43
5.2	Join Operator . . . . .	45
5.2.1	Implementation . . . . .	45

5.2.2	Feature Filtering Optimization . . . . .	48
5.2.3	Further Exploring the Design Space . . . . .	49
5.2.4	Experiments . . . . .	51
5.2.5	Limitations . . . . .	56
5.2.6	Summary . . . . .	57
5.3	Sort Operator . . . . .	57
5.3.1	Implementations . . . . .	57
5.3.2	Further Exploring the Design Space . . . . .	60
5.3.3	Experiments . . . . .	61
5.3.4	Limitations . . . . .	66
5.3.5	Summary . . . . .	67
5.4	End to End Query . . . . .	67
5.4.1	Experimental Setup . . . . .	67
5.4.2	Results . . . . .	68
5.5	Takeaways and Discussion . . . . .	68
5.6	Conclusion . . . . .	69
<b>6</b>	<b>Counting with the Crowd</b>	<b>71</b>
6.1	Introduction . . . . .	71
6.2	Motivating Examples . . . . .	73
6.2.1	Filtering Photos . . . . .	73
6.2.2	Counting Image Properties . . . . .	74
6.2.3	Coding Tweet Text . . . . .	74
6.3	Counting Approach . . . . .	75
6.3.1	User Interfaces for Count Estimation . . . . .	75
6.3.2	Modifying UDFs to Support Counting . . . . .	78
6.3.3	Estimating Overall Counts From Worker Counts . . . . .	78
6.3.4	Comparing the Approaches . . . . .	79
6.3.5	Estimating Confidence Intervals . . . . .	80
6.4	Identifying Spammers . . . . .	81
6.4.1	Definitions . . . . .	81
6.4.2	A Spammer Model . . . . .	82
6.4.3	An Ineffective Algorithm . . . . .	83
6.4.4	A Spammer Detection Algorithm That Works . . . . .	84
6.5	Protecting Against Coordinated Attacks . . . . .	85
6.5.1	Correcting Worker Responses . . . . .	86
6.5.2	Random Gold Standard Selection . . . . .	88
6.6	Further Exploring the Design Space . . . . .	88
6.7	Experiments . . . . .	89
6.7.1	Datasets . . . . .	89
6.7.2	Estimating Counts . . . . .	90
6.7.3	Sybil Attacks . . . . .	99
6.8	Takeaways and Discussion . . . . .	101
6.9	Conclusion . . . . .	103

<b>7</b>	<b>Discussion and Future Work</b>	<b>105</b>
7.1	How Crowdsourced Labor is Different . . . . .	105
7.2	Breaking out of Microtasks . . . . .	107
7.3	Moving Toward an Employment Model . . . . .	109
7.4	Alternative Optimization Objectives . . . . .	110
7.5	Designing with and for the Crowd . . . . .	111
7.6	More System Support . . . . .	113
7.7	Limitations of Experimental Results . . . . .	115
<b>8</b>	<b>Conclusion</b>	<b>117</b>



# List of Figures

1-1	A sample Amazon Mechanical Turk Task. . . . .	16
3-1	Comparison Sort . . . . .	33
3-2	Rating Sort . . . . .	34
4-1	The Qurk system architecture. . . . .	40
5-1	Simple Join . . . . .	45
5-2	Naive Batching . . . . .	46
5-3	Smart Batching . . . . .	47
5-4	Fraction of correct answers on celebrity join for different batching approaches. Results reported for ten assignments aggregated from two runs of five assignments each. Joins are conducted on two tables with 30 celebrities each, resulting in 30 matches (1 per celebrity) and 870 non-matching join pairs. . . . .	53
5-5	Completion time in hours of the 50 <sup>th</sup> , 95 <sup>th</sup> , and 100 <sup>th</sup> percentile assignment for variants of celebrity join on two tables with 30 celebrities each. . . . .	54
5-6	Comparison Sort . . . . .	58
5-7	Rating Sort . . . . .	59
5-8	$\tau$ and $\kappa$ metrics on 5 queries. . . . .	64
5-9	Hybrid sort algorithms on the 40-square dataset. . . . .	65
6-1	The label-based interface asks workers to label each item explicitly. .	76
6-2	The count-based interface asks workers to estimate the number of items with a property. . . . .	77
6-3	Example shapes in our Shapes and Colors dataset. . . . .	90
6-4	Bar chart of average error by approach, with error bars for minimum and maximum values. Numbers above each bar represent batch size. <i>LabelNoR</i> means labeling with no redundancy. <i>LabelR</i> means labeling with 5-worker redundancy. <i>Thresh</i> means count-based worker input with spammer detection. <i>Avg</i> means count-based worker input with no spammer detection. <i>Thresh</i> and <i>Avg</i> have the same inputs. . . . .	91

6-5	Box-and-whisker plots of the number of seconds workers took to complete different tasks. The top of each box represents the 75th percentile completion time and the bottom represents the 25th percentile. Red lines indicate medians. Whiskers (error bars) represent minimum (bottom) and maximum (top) values. . . . .	93
6-6	As we vary the number of HITs sampled from 1000 (X-axis), we measure the empirical 95% confidence intervals (Y-axis) of two algorithms on samples of that size. Given the same input data with a lot of spammer-provided values, the spammer-detection algorithm ( <i>Thresh</i> ) arrives at the correct estimate while a simple average of inputs ( <i>Avg</i> ) does not. . . . .	94
6-7	As we vary approaches and batch sizes, what 95% confidence interval width (Y-axis) do we achieve in a given number of HITs (X-axis)? . .	95
6-8	Bar chart of average error per approach, broken down by selectivity of males in the dataset. See Figure 6-4 for a description of the graph elements. . . . .	96
6-9	As we vary the selectivity of males, what 95% confidence interval width (Y-axis) do we achieve in a given number of HITs (X-axis)? . . . . .	97
6-10	As we increase the number of coordinated attackers (X-axis), the spam detection algorithm eventually reports the attackers' values (Y-axis). . . . .	99
6-11	How do various amounts of gold standard data reduce the effectiveness of a coordinated attack? . . . . .	100



# List of Tables

5.1	Baseline comparison of three join algorithms with no batching enabled. Each join matches 20 celebrities in two tables, resulting in 20 image matches (1 per celebrity) and 380 pairs with non-matching celebrities. Results reported for ten assignments aggregated from two trials of five assignments each. With no batching enabled, the algorithms have comparable accuracy. . . . .	52
5.2	Feature Filtering Effectiveness. . . . .	55
5.3	Leave One Out Analysis for the first combined trial. Removing hair color maintains low cost and avoids false negatives. . . . .	55
5.4	Inter-rater agreement values ( $\kappa$ ) for features. For each trial, we display $\kappa$ calculated on the entire trial's data and on 50 random samples of responses for 25% of celebrities. We report the average and standard deviation for $\kappa$ from the 50 random samples. . . . .	56
5.5	Number of HITs for each operator optimization. . . . .	67



# Chapter 1

## Introduction

Crowdsourced marketplaces such as Amazon’s Mechanical Turk [4] make it possible to recruit large numbers of people to complete small tasks that are difficult to automate on computers, such as transcribing an audio snippet or finding a person’s phone number on the Internet. Employers submit jobs (Human Intelligence Tasks, or HITs in MTurk parlance) as HTML forms requesting some information or input from workers. Workers (called Turkers on MTurk) perform the tasks, input their answers, and receive a small payment (specified by the employer) in return (typically 1–5 cents).

These marketplaces are increasingly widely used. Crowdfunder [9], a startup that builds tools to help companies use MTurk and other crowdsourcing platforms now claims to process more than 1 million tasks per day to more than 1 million workers and has raised \$17M+ in venture capital. CastingWords [8], a transcription service, uses MTurk to automate audio transcription tasks. Novel academic projects include a word processor with crowdsourced editors [21] and a mobile phone application that enables crowd workers to identify items in images taken by blind users [22]. Other applications include organizing the search through satellite imagery for Jim Gray and Stephen Fossett [1], and translating messages from Creole during the Haiti Earthquake [2].

Given the breadth of crowd-powered applications, most of which are currently hand-built and optimized, it would help to have a platform that makes it easier to build these applications. Workers typically interact with the tasks they must perform through HTML forms like the one in Figure 1-1. Most systems like MTurk offer a form builder interface as well as an API for programmatically managing crowd work. The majority of tasks are managed using the API, but typically the very basic API calls (e.g., create task, retrieve response, pay worker) are a part of higher-order workflow logic that is part of the larger application of the crowdsourced work. For example, Soylent [21] embeds crowdsourced editors in Microsoft Word, and so it requires significant logic to manage the user interface elements it adds to Word, present editing suggestions to users, and manage the multiple forms of questions it asks crowd workers, with the actual MTurk API calls it makes taking up less than 1% of its codebase.

A significant amount of the code behind crowd-powered workflows shepherds requests, vets worker responses for potential mistakes, and connects individual workers’

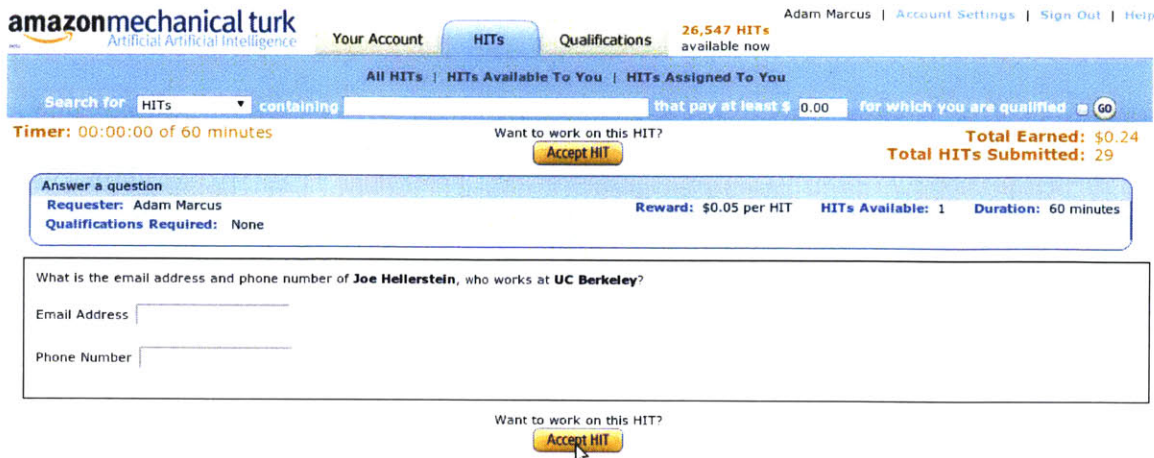


Figure 1-1: A sample Amazon Mechanical Turk Task.

contributions to form a larger whole. Furthermore, much of the code in these workflows is boilerplate, repeated across workflows, and hand-optimized by crowd workflow designers. In this dissertation, I present a system called Qurk that simplifies crowd-powered workflow development along with a set of optimized reusable workflow components that improve on the current hand-coded crowd workflow building regime. Qurk opens up several research questions in the areas of crowd-powered data operator design, crowd-powered workflow optimization, and crowd worker quality estimation.

## 1.1 Crowd Programming Challenges

While there is great promise in the ways in which we can use crowds to extend the capabilities of computation, working with crowd workers has its own set of challenges.

**Code reuse.** Programmers encounter unnecessarily repeated code in two ways as they implement crowd-powered workflows. The first is in implementing the logic behind common functionality, such as filtering or labeling a collection of items in a dataset, or performing entity resolution between two datasets. The second is in the basic plumbing of crowd work: generating HTML tasks, assignment of workers to tasks, asynchronously collecting worker responses, verifying result quality across workers, and compensating the workers with good result quality.

**Complex workflows.** It is common for crowd-powered data processing tasks to require multiple crowd-powered workflow stages. For example, Soylent shortens a paragraph with the help of the crowd in a three-stage workflow called *Find-Fix-Verify*, where one group of workers finds patches of text to shorten, another group of workers fixes up patches by providing shortenings, and yet another group verifies the fixes by voting on the best ones. Workflow manipulation can take thousands of lines of error-prone code, even though the high-level description of the workflow stages can be stated in a sentence.

**Latency.** Sending a task to human workers means a simple imperative program

that blocks on a response from each worker might block for seconds, minutes, or hours at a time. Crowd-powered workflow developers have to implement their own solutions to asynchrony through threading or message passing in order to get around this challenge.

**Cost.** To keep costs low, one can simply pay workers less to perform each task. This has the effect of increasing the time to complete a task, as workers prefer other tasks in the market. It can also reduce quality as workers increase the rate at which they perform the task, which may introduce new errors. Paying workers less also has ethical implications that I address at the end of this dissertation. Another approach to driving down cost, and sometimes even time, is to batch tasks together. Rather than labeling a single image as inappropriate, a worker might label two at a time, reducing the overhead of waiting for their work to be submitted or for the next task to load.

**Interface design.** Because humans are in many ways the most capable but least reliable part of a crowd-powered workflow, designing the interfaces that they interact with is crucial to good workflow design. In the simplest of cases, an interface bug that makes one response button larger than another might cause skew in worker responses. A more complicated case occurs in picking the right interface to elicit user feedback. Imagine sorting a set of books by quality with input from people that have read the books as Amazon does with its five-star rating system. Is a rating-based interface the best way to elicit meaningful feedback from the crowd, or would it be better to ask users who read two books to make a relative comparison between the two?

**Algorithm design.** Some common tasks, like entity resolution, can generate a number of crowd tasks that is superlinear in the size of the datasets they are processing. For example, matching companies in one dataset that are identified by their stock ticker symbols (e.g., IBM) to companies in another dataset identified by their full name (e.g., International Business Machines) might, in an entity resolution implementation, generate crowd tasks for an all-pairs comparison. Since tasks sent to humans cost money, it is desirable to design algorithms that reduce such superlinear task generation.

**Optimization.** Crowd-powered workflows regularly require multiple steps, and deciding how many resources to allocate to each workflow step as well as reordering steps to reduce crowd work is a challenging task. A common workflow optimization technique is known as *selectivity estimation*, where workflow steps that reduce the amount of work required in future steps are executed first. Implementing selectivity estimation is difficult when workflow steps require human feedback, and as we later show, naive crowd-powered selectivity estimation techniques can be an order of magnitude less efficient and two orders of magnitude less accurate than well-designed selectivity estimation approaches.

**Worker and result quality.** Workers can make mistakes or try to perform less accurate work to finish tasks faster. (Workers with consistently low-quality responses are sometimes called “spammers.”) Because of the uncertainty in worker responses, it is impossible to guarantee 100% data accuracy.

There are several approaches to achieving high quality. A simple approach is to build redundancy into tasks. For example, if a worker will, on average, correctly

label an image as inappropriate 90% of the time, we can ask three workers to label each image and take the majority vote response, resulting in around 97% accuracy assuming worker errors are independent. Other approaches involve asking one group of workers to vote on the best answer that another set of workers generated. Such multiple-worker responses tend to increase workflow cost and time as more workers are involved in a workflow. Another common approach is to pose *gold standard questions* with known answers in order to identify workers with poor performance. This approach again requires increased cost and time as each worker must perform gold standard tasks in addition to the yet-incomplete work.

Both of the methods of verifying worker quality require that worker responses either match other workers' answers or match a gold standard dataset. These approaches do not handle scenarios where correctness is not binary. For example, in determining that there are five people in an image, we want to reward a crowd worker who responded 4 more than we reward a crowd worker who responded 10.

## 1.2 Crowd Workflow Management: The Dark Ages

Crowdsourcing platforms are less than a decade old, and the ecosystem for managing them reflects their novelty. Crowdsourced workflow development is currently an ad-hoc process, where developers build imperative programs to manage the crowd. These developers recreate common crowd design patterns, hand-tune their own optimizations, and manage work quality using less than ideal algorithms.

A similar situation existed in the data management field before database management systems reduced data workflow complexity with declarative programming models, reduced program complexity with the introduction of common data operators, and reduced parameter tuning with the introduction of query optimizers. In this dissertation, I explore how analogous approaches in declarative programming, crowd-powered operators, and crowd workflow optimization can improve the ease and efficiency of managing crowd work.

## 1.3 Qurk: A Crowd-powered Workflow System

Qurk is a database workflow engine that is crowd-aware. It supports a Pig Latin-like [5] workflow definition language. While it allows common database operations like sorting and filtering, its novelty is in supporting similar operations that are crowd-powered. For example, Qurk allows a user to sort a collection of images by how interesting they look, an operation that a computer alone can not yet perform.

Qurk is one of the contributions of this dissertation, but it also opens up research opportunities of its own that address the challenges mentioned above:

- To address *code reuse* and *complex workflows*, Qurk's declarative programming language allows workflow developers to specify a high-level description of workflow stages, which the system compiles down into the complex logic that handles data shuffling, asynchrony, result quality, and workflow optimization.

- To avoid *latency*, Qurk’s operators and human task execution layer are non-blocking, with all workflow elements communicating through asynchronous messaging queues that are managed on behalf of the user.
- To reduce *cost*, Qurk applies various forms of batching work to take advantage of humans’ natural ability to batch process certain kinds of data. We also reduce cost by building new *interfaces* and *algorithms* for sorting and joining datasets with the crowd that take human cognition and input mechanisms into account.
- To promote workflow *optimization* we explore different techniques for performing selectivity estimation, and in the process contribute an algorithm for estimating *worker and result quality* in the face of continuous or ordinal response types.

In Chapter 3, we present the data model and query languages that Qurk supports to allow users to build crowd-powered workflows. In Chapter 4, we explore the architecture and implementation that allows Qurk to both handle high-latency worker responses and maintain high result quality across multiple responses.

## 1.4 Reusable Operations: Sorts and Joins

In Chapter 5, we explore how to build reusable crowd-powered operators. We specifically study how to implement crowd-powered sort and join operators.

Building a crowd-powered reusable operator requires attention to traditional considerations such as algorithmic complexity as well as new ones like human attention and user interface usability. We show that with proper design, a join operation between two sets of images (a person-identification task) can be reduced in cost by over an order of magnitude. We also explore the tradeoffs in costs and accuracy between a rating-based sort, which requires a number of tasks linear in the input size, and a comparison-based sort, which may require a number of tasks quadratic in the input size.

## 1.5 Query Optimization and Work Quality

While operator-based optimizations are important for high result quality at reasonable cost, crowd-powered workflows often contain several stages whose parameters and execution must be optimized globally. In Chapter 6, we look at one form of query optimization in the form of crowd-powered selectivity estimation. For example, consider a workflow that requires us to filter a collection of images to those of males with red hair, and consider a scenario in which redheads are less prevalent in our image collection than males. We can reduce workflow cost and time by having the crowd first filter images to those of redheads, and then search among those images for those of males.

The problem of operator ordering through selectivity estimation is one instance of a crowd-powered count-based aggregation operation. Our goal is to, for some dataset,

estimate the number of items in that dataset with a given property (e.g., hair color or gender). We compare a sampled label-based interface, which asks crowd workers to identify item properties on a sample of the dataset, to a sampled count-based interface, which shows a subset of the dataset and asks workers to estimate the number of items with the property in that sample. We show that, for items with high “pop-out,” (e.g., images of people that workers can scan quickly to determine their gender) the count-based approach can provide accurate estimates up to an order of magnitude faster and more cheaply than the label-based approach. For low pop-out items (e.g., snippets of text that require careful attention and are not easily skimmable), we find that the label-based approach is the only way to achieve reasonable result quality.

Through this work, we also explore a method of achieving high result quality by identifying spammers and coordinated attacks by workers (i.e., sybil attacks [35]). This method expands on prior work that determines worker quality by requiring multiple workers to redundantly label each item and measures workers’ overall agreement with each redundant response. Our technique allows us to ask workers questions about a larger fraction of the dataset with less redundancy in our questions, identifying spammers and improving accuracy by up to two orders of magnitude.

## 1.6 Ethics and New Employment Models

The promise of human computation is that it enables new forms of computation and more fluid access to labor than the traditional employee-employer or contract-based arrangement. Aside from the technical challenges that we have to solve in order to enjoy the benefits of crowd computing, there are also ethical ones we must consider.

Since the internet spans time, political borders, degrees of economic development, and culturally accepted norms, many questions arise as to what comprises ethical crowd computing. In Chapter 7, we consider some of these questions and review some literature from the social sciences that addresses the questions in more traditional environments. In addition to ethical questions, we also consider task employment models that bridge traditional employment and the more free-flowing models enabled by crowd work.

## 1.7 Multiple Authors and Prior Publications

The work presented in this dissertation is the result of several research collaborations across three papers and a demo. My advisors, Sam Madden, David Karger, and Rob Miller, who are also co-authors on all of these papers, aided in brainstorming many ideas, defining experiment design, and writing up portions of all three papers. Eugene Wu was instrumental in clarifying the initial ideas for Qurk as well as heavily editing our CIDR paper [56]. He was even more crucial in our study of human-powered sorts and joins [58], where he designed many of the user interfaces, implemented significant portions of the Python backend, and developed and ran many of the sort-based experiments. Eugene was an equal partner in the Scala-based implementation



of Qurk for our SIGMOD demo [57]. For our work on selectivity estimation with the crowd, many discussions with Eugene and Sewoong Oh helped frame my thoughts, though all of the development, analysis, and experiment design are my own.

## 1.8 The Use of “We” Instead of “I”

While I have no experiments to support this hypothesis, I think that in time the research community will find that recognition and attribution, given their importance in the traditional workplace, have significant weight in supporting effective crowd work.

Many crowds influenced this dissertation. My coauthors helped me frame, design, and write up the research that went into the papers behind this dissertation. The software behind Qurk was written collaboratively by Eugene Wu and me. Our human-powered sort and join operator research registered input from at least 952 Turker IDs. The selectivity estimation experiments showed contributions from at least 1062 Turker IDs. An important and oft-unacknowledged crowd includes you, the reader, to whom I write, and without whom I would be unable to frame this document.

It was with all of these crowds in mind that I wrote this dissertation. The contributions, efforts, discoveries, and conclusions are not mine alone, and I find it inappropriate to use the pronoun “I” to describe them. While I take credit for the majority of the intellectual contributions of this dissertation, *we* will explore the rest of this research together.



# Chapter 2

## Related Work

Several communities have taken to studying crowd work and its applications, systems and programming models for crowdsourcing, and worker and result quality metrics. In this chapter, we discuss the various efforts to improve the crowd-powered workflow building process.

The study of crowd work is inseparable from an understanding of human behavior, and so we can take inspiration from the field of psychology. Because the specific psychological concerns of the interfaces we design are problem-specific, we leave discussion of those effects to the individual relevant chapters.

### 2.1 Studying the Crowds

One of the first researchers to study crowd platforms such as MTurk was Ipeirotis, who offers an analysis of the marketplace [44]. From this study, we learn that between January 2009 and April 2010, MTurk saw millions of HITs, hundreds of thousands of dollars exchanged, and that the most popular tasks performed were product reviews, data collection and categorization, and transcription. HIT prices ranges from \$0.01 and \$10.00, with approximately 90% of HITs paying less than \$0.10.

Ipeirotis also surveyed Turkers to collect demographic information. Through these surveys, we learn that between 2008 and 2010, the MTurk population demographics resembled the US internet populations', with a bias toward more females (70%), younger workers, lower incomes, and smaller families [47].

As crowdsourcing platforms open up to a more global population, however, we see several demographic shifts. With an increasing crowd worker population from the developing world (in particular, India), Ross et al. show that with changes in demographics come changes in motivations for performing work [66]. Workers from India are more likely to have crowd work as a primary source of income, whereas US workers treat it as an income supplement. This study showed that workers for whom crowd work is their primary source of income, pay becomes a larger incentive for performing tasks.

Mason and Watts [59] studied the effects of price on quantity and quality of work. They find that workers are willing to complete more tasks when paid more per task.

They also find that for a given task difficulty, result accuracy is not improved by increasing worker wages.

Another study of worker incentives by Chandler and Kapelner [25] shows that US workers on MTurk are more likely to perform a task if given context. Specifically, they are more likely to start doing work on a task when told that they will be labeling images of tumor cells than when they will be labeling objects of interest. Indian Turkers do not display this property, favoring both tasks equally. Still, once either worker starts performing a task, their quality is commensurate.

## 2.2 Applications of Crowds

Before we can understand different systems and models for building crowd-powered workflows, we must first understand the types of applications that are built using those workflows.

Crowd-powered tasks have made their way into user interfaces. In Soylent [21], Bernstein et al. add shortening, proofreading, and macro functionality in Word. The authors present a programming paradigm they name *Find-Fix-Verify*, which is designed to elicit several small bits of creative work from the crowd and then have other crowd workers rate the alternative contributions to separate good from bad.

Find-Fix-Verify is one workflow for eliciting work from crowd workers. Little et al. present an iterative approach [54], which is useful in having multiple workers iteratively refine results of prior workers on difficult tasks such as decyphering a hard-to-read text scan. Such novel crowd-powered workflows suggest the need for a programming language and system that make it easy to define and optimize common patterns for directing the crowd.

In a mix of human and machine computation, Branson et al. integrate humans and learning algorithms into a visual 20 questions game [23]. In this game, a coordinating algorithm uses machine vision to select and put an ordering on the questions asked of humans to assist in classifying pictures of birds.

In VizWiz, Bigam et al. [22] describe a mobile phone application that allows blind users to have various crowds, ranging from Turkers to their Facebook friends, to label objects they take pictures of. In such a scenario, latency is important, and the authors present a system called quikTurkit that, for a fixed price, can drastically reduce the latency of results. Latency reduction is accomplished through several techniques, including reposting tasks with regular frequency and posting more tasks than are immediately available to entice Turkers. The existence and effectiveness of such techniques suggests that latency reduction on MTurk is not a fruitful area of research for the database community, since the majority of latency reduction can be achieved by catering to idiosyncracies of the ecosystem.

Noronha et al. show with Platamate [60] that in addition to labeling images with which foods they contain, Turkers are also effective at estimating the nutritional properties of the food, such as how many calories the food contains.

With techniques like Games with a Purpose [73] and CAPTCHA [74], von Ahn et al. show that it is possible to incentivize crowd work with non-monetary rewards.

In Games with a Purpose, the crowd is presented with games that, as a byproduct, label images or music. With CAPTCHA, users who wish to authenticate themselves as human and not a spammer decipher difficult-to-read text or audio clips, thereby transcribing those clips in the process.

Outside of the research world, several commercial applications of crowdsourcing have popped up. CastingWords [8] is an MTurk-powered service that allows users to pay to have their audio clips transcribed, with each worker performing a few minutes of transcription. Captricity [6] uses its own crowd to convert paper forms into digital records, and comes out of research by Chen et al. [26, 27]. CardMunch [7], which was purchased by LinkedIn in 2011, allowed users to take photos of business cards and have them digitized by Turkers. SmartSheet [11] allows crowdsourced tasks to be managed through a spreadsheet interface, which is a powerful way to view tasks to be completed alongside data improvements.

## 2.3 Crowdsourced Databases

Given the parallels between crowd workflow optimization and traditional database query optimization, applying and modifying database technology to crowd work is natural. Several crowd-powered databases including CrowdDB [40], Deco [64, 65], and our own Qurk [56, 58] enable users to issue data-oriented tasks to crowd workers.

While the design details between the various crowd-powered databases are interesting, the majority of the early research contributions in this space have been around database operators, including filters [63], sorts and joins [58], and max-finding [42]. Some of the operator-based work explores proving bounds on the complexity or cost of achieving a certain level of data quality with a certain quality of worker input. Other aspects of this work involve exploring the user interfaces that various algorithms require as input, as we do in our study of crowd-powered sort and join implementations (Chapter 5).

In addition to discussing operator implementations such as sorts and joins in this dissertation, we extend crowd-powered query optimization to consider cross-operator optimizations through selectivity estimation, having the crowd estimate the selectivity of various operators. This has the benefit of serving both crowd-powered query optimizers and implementing crowd-powered *COUNT* aggregates with *GROUP BY* clauses.

In CrowdDB, Franklin et al. present the concept of an *Open World* model of databases that crowd-powered databases allow. Essentially, in this model, all of the tuples relevant to a query are not assumed to be in the database *a priori*, and instead can be retrieved in real time. While this model is desirable for generative open-ended problems, it poses difficulties in optimizing queries for which one does not have all of the data. As an example, consider the following query:

```
SELECT *
FROM professors
WHERE professors.university = "Berkeley";
```

In an open-world model database, we can not be sure that the *professors* table contains all of the professors at Berkeley, and have to decide both how many Berkeley professors exist in the world and have the crowd determine who they are. To address these challenges, Trushkowsky et al. have shown how to both estimate the cardinality of a set, and identify the unique elements in that set with the help of the crowd [72]. This work is related to our selectivity estimation problem: for us to be able to calculate counts of large numbers of groups with the crowd, it is important to enumerate the distinct group elements which we are counting.

The database literature offers some guidance in how to process uncertain data, which Qurk will also generate. Trio [13] handles data with uncertain values and lineage, and presents a language for querying this data. Dalvi and Suciu explore efficient queries over probabilistic databases [32]. BayesStore [75] takes this a step further, adding complex inference functionality to databases with probabilistic data. MauveDB [34] explores generating model-based views over tables in order to model, clean, and perform inference over the data. In Qurk, our workflow and user-defined function language does not handle uncertain data. Instead, the system allows programmers to specify how to convert multiple potentially different worker responses for the same task into a single definitive answer, for example by keeping the response indicated by a majority vote.

## 2.4 Crowd Programming Models

Crowdsourcing platforms such as MTurk offer low-level interfaces for posting HTML forms or iframes around HIT content that is hosted elsewhere. The APIs allow users to generate new HITs, specify a price per HIT assignment, and set a number of assignments per HIT. Users manage their own multi-HIT workflows, poll the API for task completion, and gauge the quality of the responses on their own.

MTurk-style APIs are akin to filesystem and network APIs on which databases and other data and information management platforms are built. Building robust crowdsourced workflows that produce reliable, error-free answers on top of these APIs is not easy. One has to consider how to design the user interface (an HTML form) the crowd worker sees, the price to pay for each task, how to weed out sloppy or intentionally incorrect answers, and how to deal with latency on the order of minutes to hours of various HITs that crowd-powered programs generate. Several startups, such as CrowdFlower [9] and MobileWorks [10] aim to make crowdsourced workflow development easier by offering simplified APIs (CrowdFlower) or task-specific ones (MobileWorks).

Further up the stack are crowdsourced language primitives. Little et al. present TurKit [55], which supports a process in which a single task, such as sorting or editing text, might be implemented as multiple coordinated HITs, and offers a persistence layer that makes it simple to iteratively develop such tasks without incurring excessive HIT costs. Much like low-level parallelization frameworks such as *pthread*s [24] allow developers to fork multiple tasks off to workers in parallel, TurKit offers several parallelization primitives. Like low-level threading primitives, however, low-level crowd

programming libraries require care in correctly using *fork/join*-style parallelization primitives.

Crowdsourcing best-practices are generally relayed as workflows like Soylent’s Find-Fix-Verify. As such, several workflow-oriented systems allow developers to specify at a high level how to process various bits of data, and a system manages HIT creation, aggregation, and, eventually, pricing and quality. Outside of the databases community, systems such as CrowdForge [51] and Jabberwocky [14] help manage such workflows. CrowdForge by Kittur et al. provides a MapReduce-style programming model for task decomposition and verification. Jabberwocky from Ahmad et al. provides a full stack: Dormouse for low-level crowdsourcing primitives, ManReduce for a MapReduce-style task decomposition, and Dog as a Pig-like programming environment [5] for specifying workflows at a high level. While the flavors of Pig Latin in Qurk and Dog are similar in that they allow developers to declaratively specify workflow steps, Dog’s contribution is a declarative specification of how to recruit crowd workers from different platforms, whereas Qurk focuses on how to define tasks encoded in user-defined functions.

The database community has largely rallied around SQL with some crowd-oriented additions to provide a declarative programming interface for the crowd. While we ultimately settled on Pig Latin for Qurk’s programming language, our changes to the core language constructs are the least invasive of the crowd-powered databases. This is because, like traditional databases, we assume a closed-world data model, and provide a method for defining crowd-powered user-defined functions that do not otherwise require modification to the query language. As we tested the limits of the language for building Find-Fix-Verify-style workflows, we realized that a workflow language like Pig Latin might be a better choice than SQL, as it offers the benefits of a declarative programming interface while providing imperative step-by-step descriptions of crowd workflows.

Developers of crowdsourced programming languages are not the first to consider human-in-the-loop programming. Business Process Execution Language (BPEL) [30] is a language for specifying executable business process workflows, like submitting a request to another person, waiting for a response, and taking different actions depending on the response. The crowdsourcing community has for the most part not adopted the BPEL standard, perhaps because other programming models such as MapReduce or SQL are already well-known and used.

## 2.5 Worker and Result Quality

One of the key problems in designing a crowd-powered workflow is providing a desired level of worker response quality. Even the best of workers make mistakes, and building redundancy and verification into a workflow is required.

CrowdFlower requests that users provide gold standard data with which to test worker quality, and disallows workers who perform poorly on the gold standard. For categorical data, an approach such as asking multiple workers each question and selecting a majority vote of responses can improve results. Dawid and Skene presented

an expectation maximization technique for iteratively estimating worker and result quality [33] in the absence of gold standard data. Ipeirotis et al. [46] modified this technique to consider bias between workers in addition to worker quality on categorical data. Karger et al. extend this work, identifying a theoretically better algorithm for quality labels [49].

In this thesis, we explore the effects of using majority vote and iterative worker quality measures when implementing crowd-powered joins in Chapter 5. We also extend the iterative worker quality approach to continuous data without redundant samples in our exploration of estimating counts in Chapter 6. We find that for count estimation, it is better to avoid the redundant labels and instead increase diversity in samples to achieve faster convergence. Our worker quality detection technique allows us to achieve even faster convergence by identifying poor-quality worker output.

Thus far, techniques for measuring worker quality as a proxy for result quality have assumed a strong worker identity system and no cross-worker coordination. These algorithms are susceptible to Sybil attacks [35] by workers with multiple identities or workers who act in concert to avoid being identified as spammers while finishing tasks faster than high-quality workers. Sybil attacks are well-researched in systems design, but have not yet been introduced as research problems with practical solutions in the field of human computation. We propose a Sybil-resistant solution to our count estimation problem, and show that it works in simulations of such attacks.

Since result quality, cost, and latency are often at tension, we require constraint-based optimization to satisfy users' multiple requirements. Dai et al. explore such workflow parameter optimization techniques in a decision-theoretic framework [31].



# Chapter 3

## Qurk Query and Data Model

In this chapter, we present Qurk’s data model and query model. Qurk follows the traditional relational data model. The the original Qurk query language was SQL, but we moved toward a Pig Latin-based language [62]. SQL, while compact, standardized, and feature-rich, has properties that make it less desirable for describing longer workflows of computation. Because high-level human computation pipelines often have multiple steps and branches, we explore a Pig Latin-based query interface to Qurk for defining these workflows. After describing our data and query models with examples of particular features, we close with an end-to-end workflow example from Soylent’s Find-Fix-Verify [21].

### 3.1 Data Model

Qurk’s query executor operates on relational data with no modifications to the traditional relational model [29]. This means that data is stored in tables that contain rows. Each table has several columns to describe row properties, and for any row, a column contains a single value. We integrate crowd input into a query plan by allowing developers to define crowd-powered user-defined functions (UDFs).

Our choice of data model design works well for processing varied datasets, but a wrinkle in the design arises when dealing with multiple crowd worker responses. Because any one crowd worker’s response cannot be counted on to be correct, we often assign multiple workers to a HIT. As we describe in Section 3.2, Qurk’s UDF descriptions specify how to combine multiple worker responses.

Our choice of Pig Latin as the basis for our query language does not suggest we employ a nested relational model as Pig does. Our design goal in borrowing from Pig was to allow simple workflow descriptions while providing developers with the more widely recognized operators of the relational model. In the nested model, tables can contain other tables, allowing set-valued attributes. This table nesting is facilitated by non-relational operator implementations like nested `FOREACH` and `GROUP` operators. As we show in Section 3.2, our language design utilizes Pig Latin’s workflow description logic, and borrows operators that remove set-valued attributes that appear in multiple worker responses, while replacing non-relational operators

such as Pig’s GROUP with relational ones as they appear in SQL.

## 3.2 Pig Latin-based Query Language

We now describe the Qurk query language, and focus on how operators such as filters, joins, and sorts are expressed through a series of queries. Users utilize the language to describe crowd-powered workflows to process their datasets, and periodically call out to crowd workers in these workflows by defining templates for crowd-powered UDFs. By embedding crowd-powered UDFs in a Pig Latin-like workflow description language, we base our language design on a set of data processing primitives that are already vetted by the declarative workflow-building community.

Our examples have workers provide us with information about various images. We use image examples for consistency of explanation, and because databases typically do not perform processing over images. While image processing makes for a good example of the benefit of crowd-powered databases, Qurk’s use cases are not limited to processing images. Franklin et al. [40] show how human computation-aware joins can be used for entity disambiguation, and we explore using workers to rate video content in Section 5.4. In Section 6, we also explore labeling and counting textual data such as tweets.

Qurk supports user-defined scalar and table functions (UDFs) to retrieve data from the crowd. Rather than requiring users to implement these UDFs in terms of raw HTML forms or low-level code, most UDFs are implemented using one of several pre-defined Task templates that specify information about how Qurk should present questions to the crowd. Qurk compiles task templates into an HTML form that displays an item to be processed, a prompt describing how to process it, and form elements to collect worker output.

To illustrate a simple example, suppose we have a table of celebrities, with schema `celeb(name text, img url)`.

We want the crowd to filter this table and find celebrities that are female. We would write:

```
female_celebs =  
  FILTER celeb BY isFemale(c.url)
```

This is an unadulterated Pig Latin query, with `isFemale` defined as follows:

```
TASK isFemale(field) TYPE Filter:  
  Prompt: "<table><tr> \  
    <td><img src='%s'></td> \  
    <td>Is the person in the image a woman?</td> \  
  </tr></table>", tuple[field]  
  YesText: "Yes"  
  NoText: "No"  
  BatchPrompt: "There are %d people below. \  
    Please specify whether each is a \  
    Yes No"
```

```
woman.", BATCHSIZE
Combiner: MajorityVote
```

Tasks have types (e.g., `Filter` for yes/no questions or `Rank` for sort-based comparators) that define the low-level implementation and interface to be generated. Filter tasks take tuples as input, and produce tuples that users indicate satisfy the question specified in the `Prompt` field. Here, `Prompt` is simply an HTML block into which the programmer can substitute fields from the tuple being filtered. This tuple is available via the `tuple` variable; its fields are accessed via the use of field names in square brackets. In this case, the question shows an image of the celebrity and asks if they are female. The `YesText` and `NoText` fields allow developers to specify the titles of buttons to answer the question.

Filters describe how to ask a worker about one tuple. The query compiler and optimizer can choose to repeat the `Prompts` for several tuples at once. This allows workers to perform several filter operations on records from the same table in a single HIT. The `BatchPrompt` allows a user to specify to workers that multiple answers are expected of them in a single HIT.

Workers sometimes make mistakes, generate unusual answers, or, in an attempt to make money quickly, submit tasks without following directions. Because any one worker might provide an unreliable response, it is valuable to ask multiple workers for answers. We allow users to specify how many responses are desired; by default we send jobs to 5 workers. Certain worker response combination logic (described next in our discussion of `Combiners`) will ask progressively more workers to provide an answer if a task is particularly difficult, and so the number of workers per task is not always fixed. In our experiments we measure the effect of the number of workers on answer quality. We also discuss algorithms for adaptively deciding whether another answer is needed in Section 5.5.

The `Combiner` field specifies a function that determines how to combine multiple responses into one answer. In addition to providing a `MajorityVote` combiner, which returns the most popular answer, we have implemented the method described by Ipeirotis et al. [46]. This method, which we call `QualityAdjust`, identifies spammers and worker bias, and iteratively adjusts answer confidence accordingly in an Expectation Maximization-like fashion. Developers can implement their own combiners by building a custom user-defined aggregate (UDA), that, in addition to providing an aggregate result, can also request more worker responses if answer certainty is low.

Advanced users of Qurk can define their own tasks that, for example, generate specialized UIs. However, these custom UIs require additional effort if one wishes to take advantage of optimizations such as batching.

### 3.2.1 Generative Tasks

Filter tasks have a constrained user interface for providing a response. Often, a task requires workers to generate unconstrained input, such as producing a label for an image or finding a phone number. In these situations, we must normalize worker responses to better take advantage of multiple worker responses. Since generative

tasks can have workers generate data for multiple fields and return tuples, this is a way to generate tables of data.

For example, say we have a table of animal photos: `animals(id integer, img url)`. We wish to ask workers to provide us with the common name and species of each animal:

```
common_species =  
  FOREACH animals  
  GENERATE id, animalInfo(img).common  
           animalInfo(img).species
```

In this case, `animalInfo` is a generative UDF which returns two fields, `common` with the common name, and `species` with the species name.

```
TASK animalInfo(field) TYPE Generative:  
  Prompt: "<table><tr> \  
          <td><img src='%s'> \  
          <td>What is the common name \  
          and species of this animal? \  
        </table>", tuple[field]  
  Fields: {  
    common: { Response: Text("Common name"),  
              Combiner: MajorityVote,  
              Normalizer: LowercaseSingleSpace },  
    species: { Response: Text("Species"),  
              Combiner: MajorityVote,  
              Normalizer: LowercaseSingleSpace }  
  }
```

A generative task provides a `Prompt` for asking a question, much like a filter task. It can return a tuple with fields specified in the `Fields` parameter. Just like the filter task, we can combine the work with a `Combiner`. We also introduce a `Normalizer`, which takes the text input from workers and normalizes it by lower-casing and single-spacing it, which makes the combiner more effective at aggregating responses. While one is not present in this example, a user can also provide a `BatchPrompt` to precede multiple batched questions.

Pig Latin provides a method of nesting results inside other results with statements nested inside `FOREACH` operations. We do not allow this language feature to avoid generating set-valued attributes.

### 3.2.2 Sorts

Sorts are implemented through UDFs specified in the `ORDER BY` clause. Suppose, for example, we have a table of images of squares of different sizes, with schema `squares(label text, img url)`. To order these by the area of the square, we write:

```
sorted =
  ORDER squares BY squareSorter(img)
```

where the task definition for `squareSorter` is as follows.

```
TASK squareSorter(field) TYPE Rank:
  SingularName: "square"
  PluralName: "squares"
  OrderDimensionName: "area"
  LeastName: "smallest"
  MostName: "largest"
  Html: "<img src='%s' class=lgImg>",tuple[field]
```

As we discuss in Section 5.3, Quirk uses one of several different interfaces for ordering elements. One version asks workers to order small subsets of elements. Another version asks users to provide a numerical ranking for each element. The `Rank` task asks the developer to specify a set of labels that are used to populate these different interfaces. In the case of comparing several squares, the above text will generate an interface like the ones shown in Figure 3-1 and 3-2.

## There are 2 groups of squares. We want to order the squares in each group from smallest to largest.

- Each group is surrounded by a dotted line. Only compare the squares within a group.
- Within each group, assign a number from 1 to 7 to each square, so that:
  - 1 represents the smallest square, and 7 represents the largest.
  - We do not care about the specific value of each square, only the relative order of the squares.
  - Some groups may have less than 7 squares. That is OK: use less than 7 numbers, and make sure they are ordered according to size.
  - If two squares in a group are the same size, you should assign them the same number.

The interface consists of two groups of squares, each enclosed in a dotted rectangular border. The top group contains five squares of different sizes, with a small input field below each. The bottom group also contains five squares of different sizes, with a small input field below each. Below the bottom group is a green button with the word "Submit" in white text.

Figure 3-1: Comparison Sort

As with filters, tasks like `Rank` specified in the `ORDER BY` clause can ask users to provide ordering information about several records from the input relation in a single HIT. This allows our interface to batch together several tuples for a worker to process.

**There are 2 squares below. We want to rate squares by their size.**

- For each square, assign it a number from 1 (smallest) to 7 (largest) indicating its size.
- For perspective, here is a small number of other randomly picked squares:



smallest

1

2

3

4

5

6

7

largest

smallest

1

2

3

4

5

6

7

largest

Submit

Figure 3-2: Rating Sort

### 3.2.3 Joins and Feature Extraction

The basic implementation of joins is similar to that for sorts and filters. Suppose we want to join a table of images with schema `photos(img url)` with the celebrities table defined above:

```
combined =
  JOIN celeb, photos
  ON samePerson(celeb.img, photos.img)
```

This join syntax borrows from SQL, since Pig Latin's join logic assumes join predicates are defined as equality between two fields. Qurk instead allows more rich human-powered join predicates like `samePerson`, which is an equijoin task that is defined as follows:

```
TASK samePerson(f1, f2) TYPE EquiJoin:
  SingularName: "celebrity"
  PluralName: "celebrities"
  LeftPreview: "<img src='%s' class=smImg>",tuple1[f1]
  LeftNormal: "<img src='%s' class=lgImg>",tuple1[f1]
  RightPreview: "<img src='%s' class=smImg>",tuple2[f2]
  RightNormal: "<img src='%s' class=lgImg>",tuple2[f2]
  Combiner: MajorityVote
```

The fields in this task are used to generate one of several different join interfaces that is presented to the user. The basic idea with these interfaces is to ask users to compare pairs of elements from the two tables (accessed through the `tuple1` and `tuple2` variables); these pairs are used to generate join results. As with sorting and filter, Qurk can automatically batch together several join tasks into one HIT. A sample interface is shown in Figure 5-1.

As we discuss in Section 5.2.2, we often wish to extract features of items being joined together to filter potential join candidates down, and allow us to avoid computing a cross product. Some features may not be useful for accurately trimming the cross product, and so we introduce a syntax for users to suggest features for filtering that may or may not be used (as we discuss in Section 5.2.2, the system automatically selects which features to apply.)

We supplement traditional join syntax with an `OPTIONALLY` keyword that indicates the features that may help filter the join. For example, the query:

```
combined =
  JOIN celeb, photos
  ON samePerson(celeb.img, photos.img)
  AND OPTIONALLY gender(celeb.img) == gender(photos.img)
  AND OPTIONALLY hairColor(celeb.img) == hairColor(photos.img)
  AND OPTIONALLY skinColor(celeb.img) == skinColor(photos.img)
```

joins the `celeb` and `photos` table as above. The additional `OPTIONALLY` clause filters extract gender, hair color, and skin color from images being joined and are used to reduce the number of join candidates that the join considers. Specifically, the system only asks users to join elements from the two tables if all of the predicates in the `OPTIONALLY` clauses that Qurk uses are satisfied. The predicates that Qurk does use to narrow down join candidates can be applied in a linear scan of the tables, avoiding a cross product that might otherwise result. In Section 5.2.2, we discuss scenarios where not all of the predicates proposed by a user are used for feature filtering. Here, `gender`, `hairColor`, and `skinColor` are UDFs that return one of several possible values. For example:

TASK `gender(field)` TYPE Generative:

```
Prompt: "<table><tr> \
        <td><img src='%s'> \
        <td>What is this person's gender? \
      </table>", tuple[field]
Response: Choice("Gender",
                 ["Male", "Female", UNKNOWN])
Combiner: MajorityVote
```

In contrast to the `animalInfo` generative task, note that this generative task only has one field, so it omits the `Fields` parameter. Additionally, the field does not require a `Normalizer` because it has a constrained input space.

It is possible for feature extraction interfaces to generate a special value `UNKNOWN`, which indicates a worker could not tell its value. This special value is equal to any other value, so that an `UNKNOWN` value does not remove potential join candidates.

### 3.2.4 Aggregates

Finally, we turn to grouping and aggregating data. Imagine a table `shapes(id, picture)` that contains several pictures of shapes that vary in fill color. If one wanted to generate an interface to navigate this collection of images, it would help to summarize all colors of images and their frequency in the dataset, as in the following query:

```
counted_shapes =  
  FOREACH shapes  
  GENERATE fillColor(picture), COUNT(id)  
  GROUP BY fillColor(picture)
```

Here, `fillColor` is a crowd-powered UDF that asks a user to specify the color of a shape from a drop-down list of possible colors. Our grouping and aggregation language design is a diversion from the design for Pig Latin. In Pig Latin, grouping and aggregating are performed in separate `GROUP` and nested `FOREACH` steps. Since we do not want to use a grouping operator that generates set-valued attributes, we instead borrow from SQL's syntax, which performs grouping and aggregation in the same query.

## 3.3 Workflow Example

The language features we have described until now allow a user to compactly specify a single crowd-powered data processing step. If this were the only requirement of our query language, SQL would be an effective choice of query language. In practice, crowd-powered pipelines require multiple steps that flow into one-another. As our running example, we will consider Soylent's Find-Fix-Verify [21] algorithm as it is used to shorten the length of a long paper.

To avoid giving any one crowd worker too much power to change significant portions of a paper, the Find-Fix-Verify workflow creates microtasks that put workers' responses at tension with one-another. Given a section of text to shorten, the algorithm guides through the text paragraph-by-paragraph. In a given paragraph, the Find stage asks multiple crowd workers to identify patches of text that could be shortened. In the Fix stage, each patch is sent to a different crowd of workers who recommend shorter versions of the text. Finally, the multiple patches and shortened recommendations are then sent to a third group of crowd workers that Verify, or vote, on the best shortenings.

To implement such a workflow in a SQL-like language, users would have two choices. A user could utilize temporary tables to capture the output of each workflow step, for example creating a `find_temp` table with the output of the Find stage. This



has the undesirable effect of separating query plan elements by tables, across which most query optimizers do not optimize or re-order plan elements. Alternatively, the worker could nest the SQL queries inside one-another. This would have the effect of requiring the workflow to be stated in reverse, with the `Verify` stage embedding a nested `Fix` stage as a subquery and so on. Such an nested/inverted workflow description is undesirable as it makes the workflow difficult to understand.

Languages such as Pig Latin allow declarative workflow specifications with named stages that feed into one-another. We can succinctly describe the Find-Fix-Verify workflow in Qurk’s Pig Latin-inspired language. Given a relation `paragraphs(paragraph_id, text)` with paragraphs of a paper (or, alternatively, a previous process that splits a paper into paragraphs), one can issue the following Pig Latin-like query:

```
find = FOREACH paragraphs
      GENERATE paragraph_id, text,
                FLATTEN(suggestPatch(text)) AS patch;
fix = FOREACH find
      GENERATE paragraph_id, text, patch,
                FLATTEN(suggestCut(text, patch)) AS fixed;
verify = SORT fix
        BY paragraph_id, compareFixes(text, patch, fixed);
```

This query has three phases. In the `find` phase, we emit the patches of text that the crowd identifies as candidates for shortening. To understand what purpose `FLATTEN` serves, we must first look at the definition of `suggestPatch`.

```
TASK suggestPatch(field) TYPE Generative:
  Prompt: "In the paragraph below, please highlight \
          a patch of text that can be shortened: \
          <div id='paragraph_container'>%s</div>", tuple[field]
  Fields: {
    common: { Response: HighlightPatch("paragraph_container"),
              Combiner: ListGenerator
            }
  }
```

The `suggestPatch` UDF presents the paragraph to a crowd worker, who is prompted to highlight a patch of text in the paragraph that can be shortened. The response is encoded by `HighlightPatch`, a javascript-powered highlight-detection module that returns the start and end offsets of the highlighted patch in the paragraph. The `Combiner`, which until now has been `MajorityVote` or `QualityAdjust`, is instead `ListGenerator`. While previous combiners assumed answers would be similar, and some form of overlap amongst worker responses could provide the correct one, there is no good way to combine the highlighted patches from different workers until after

cuts are recommended for them in the `fix` stage. Instead, the `ListGenerator` returns a list of all of the patches suggested by the crowd<sup>1</sup>.

In order to keep with the relational model and avoid set-valued attributes, we wrap the list returned by `suggestPatches` in `FLATTEN`, which creates one tuple per patch. As an example, consider some paragraph (e.g., `paragraph_id = 3`) with some text (e.g., "There are several ways to..."). If `suggestPatches` returns a list with highlighted patches from two crowd workers (e.g., [`start: 5, end: 20, start: 10, end: 36`]), the `find` stage will emit:

```
3 | "There are several ways to..." | {start: 5, end: 20}
3 | "There are several ways to..." | {start: 10, end: 36}
```

Once candidate patches are emitted from `find`, the `fix` stage will prompt workers with another `Generative UDF` called `suggestCut` that highlights a patch in the paragraph and asks the crowd to recommend shorter versions of the highlighted text. These potential cuts are again not easily combined, and are thus flattened into one row per potential cut. In the `verify` stage, we sort the worker responses by paragraph ID (a comparison that is easily performed by `Qurk`), and then, per paragraph, prompt the crowd to put a ranking on the shortened patches using a `UDF` called `compareFixes` that has `UDF` type `Rank`.

We see that, for Find-Fix-Verify, the workflow can be described in roughly three lines of `Qurk` workflow logic and less than 100 lines of `Qurk UDF` logic specifications. By contrast, the `TurKit`-based `Soylent` implementation of the workflow logic behind Find-Fix-Verify requires more than 1000 lines of javascript. While not a scientific comparison, it's clear that a declarative approach reduces the amount of code required to implement a crowd-powered query.

Furthermore, the `Pig Latin`-based implementation is more approachable to developers and readers than a `SQL`-based implementation, as there are no nested queries or temporary tables. While the `SQL`- and `Pig Latin`-based approaches are equivalent in terms of optimization opportunities and functionality, we opted to implement a `Pig Latin`-like query language for our `Qurk` prototype because we found `Pig Latin` to suit real-world use cases that are more workflow/stage-oriented. This reasoning is echoed by other crowd language designs like `Jabberwocky's Dog` [14].

---

<sup>1</sup>The actual solution proposed in `Soylent` is to generate patches of text based on high-overlap regions identified by multiple workers. Using `ListGenerator` as a combiner has the effect of asking for fixes across all highlighted patches, whereas a combiner that creates a list of several high-overlap patches of text would be more true to the solution proposed by the authors of `Soylent`.

# Chapter 4

## Architecture and Query Execution

In this chapter, we describe the architecture of Qurk, provide some implementation details, and discuss the query execution process.

### 4.1 Architecture

Qurk is architected to handle an atypical database workload. The crowdsourced workloads that Qurk handles rarely encounter hundreds of thousands of tuples, and an individual operation, encoded in a HIT, takes several minutes. Components of the system operate asynchronously, and the results of almost all operations are saved to avoid re-running unnecessary crowd work. Qurk’s architecture is shown in Figure 4-1.

The **Query Executor** takes as input query plans from the query optimizer and executes the plan, possibly generating a set of tasks for humans to perform according to the rules in Section 4.3. Due to the latency in processing HITs, each operator runs in its own thread, asynchronously consuming results from input queues and sending tasks to the **Task Manager**. This asynchronous operator and input queue design is similar to that of the Volcano parallel query executor [41].

The Task Manager maintains a global queue of tasks to perform that have been enqueued by all operators, and builds an internal representation of the HIT required to fulfill a task. The manager takes data that the **Statistics Manager** has collected to determine the number of workers and the cost to charge per task, which can differ per operator. Additionally, the manager can collapse several tasks generated by operators into a single HIT. These optimizations collect several tuples from a single operator (e.g., collecting multiple tuples to sort) or from a set of operators (e.g., sending multiple filter operations over the same tuple to a single worker).

Qurk first looks up each HIT that is generated from the task manager in the **Task Cache** to determine if the the result has been cached (if an identical HIT was executed already) and probes the **Task Model** to determine if a learning model has been primed with enough training data from other HITs to provide a sufficient result without calling out to a crowd worker. The Task Model currently serves as an architectural placeholder, and is not explored or implemented in this dissertation. We discuss the open questions around implementing the Task Model in Chapter 7.

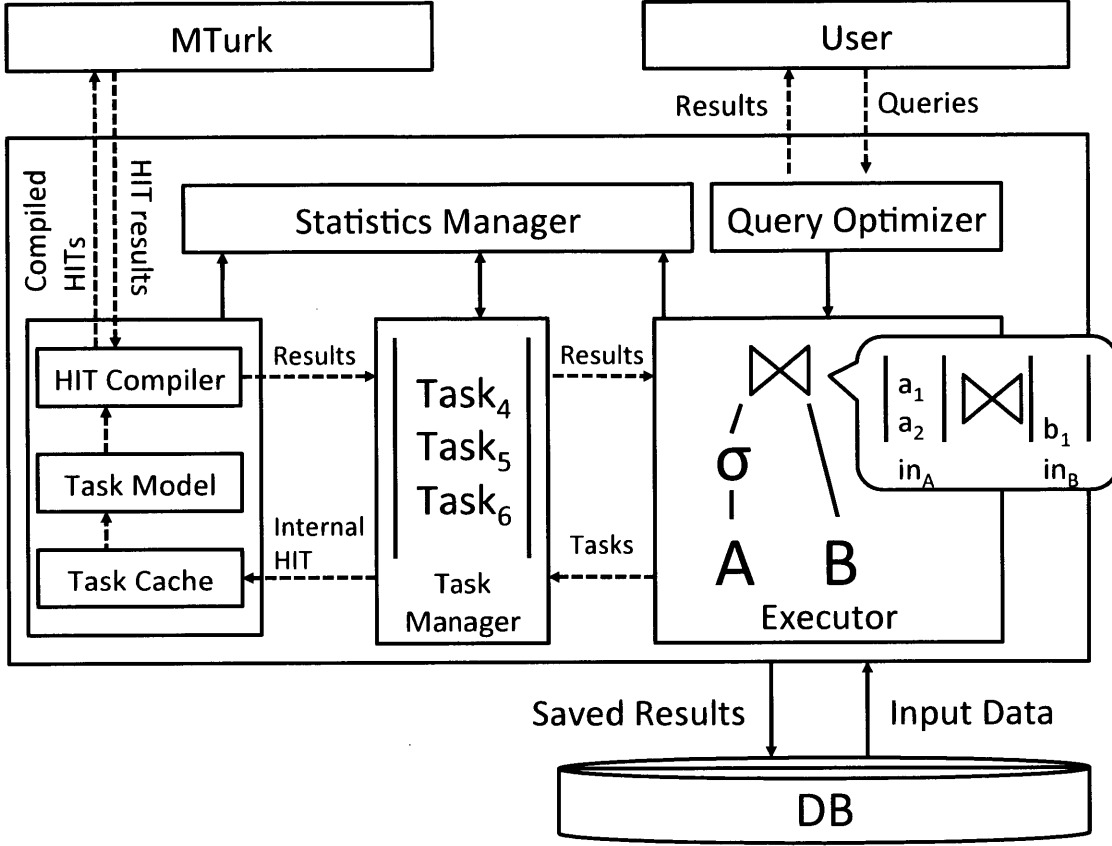


Figure 4-1: The Qurk system architecture.

If the HIT cannot be satisfied by the Task Cache or Task Model, then it is passed along to the **HIT Compiler**, which generates the HTML form that a workers will see when they accept the HIT, as well as other information required by the crowdsourcing platform (e.g., MTurk). The compiled HIT is then passed to the crowdsourcing platform. Upon completion of a HIT, the Task Model and Task Cache are updated with the newly learned data, and the Task Manager enqueues the resulting data in the next operator of the query plan. Once results are emitted from the topmost operator, they are inserted into the database. The user can periodically check for new records, or wait until a callback alerts them that the query has completed.

## 4.2 Implementation Details

Qurk is implemented as a Scala workflow engine with several input types including relational databases and tab-delimited text files. To quickly iterate on many of the interfaces for this dissertation, we created several interface prototypes and experiments in Python using the Django web framework.

**Pricing Tasks.** Our current Qurk implementation runs on top of Mechanical Turk. We pay a fixed value per HIT (\$0.01 in our experiments). Research by Mason and Watts has suggested that workers on Mechanical Turk do not do particularly higher

quality work for higher priced tasks [59]. Mason and Watts also find that workers are willing to perform more work in each task with an increase in pay per task, suggesting that Turkers have an internal model of how much money their work is worth. In all of our experiments, the basic tasks we perform are quick enough that users will do several of them for \$0.01, which means we can batch several tasks into a single HIT. Paying more per HIT would allow us to perform more batching, but the degree of additional batching would scale linearly with the additional money we pay, which wouldn’t save us money.

**Objective Function.** Because we pay a fixed value per HIT, Qurk uses a simple objective function: minimize the total number of HITs required to fully process a query subject to the constraint that query answers are actually produced<sup>1</sup>. The constraint arises because certain optimizations we apply, like batching, will eventually lead to HITs that are too time-consuming for users to be willing to do for \$0.01.

**Batching.** Qurk automatically applies two types of batching to tasks: *merging*, where we generate a single HIT that applies a given task (operator) to multiple tuples, and *combining*, where we generate a single HIT that applies several tasks (generally only filters and generative tasks) to the same tuple. Both of these optimizations have the effect of reducing the total number of HITs<sup>2</sup>. For filters and generative tasks, batches are generated by concatenating multiple **Prompt** sections (e.g., “Is the person in this image female?”) for multiple tasks together onto the single web page presented to the user. These batched **Prompts** are preceded with a single **BatchPrompt** (e.g., “Please answer questions about the gender of people in the following 5 pictures. If you are unsure about a person’s gender, click **unsure**”) as described in Section 3.2. Different user interfaces require different levels of batching as we will see when we explore sorts, joins, and counts. We leave it to future work to estimate the best batch size per interface.

**HIT Groups.** In addition to batching several tasks into a single HIT, Qurk groups together (batched) HITs from the same operator into groups that are sent to Mechanical Turk as a single HIT group. This is done because Turkers tend to gravitate toward HIT groups with more tasks available in them, as they can more quickly perform work once they are familiar with the interface. In CrowdDB [40], the authors show the effect of HIT group size on task completion rate.

## 4.3 HIT Generation

Qurk queries are translated into HITs that are issued to the underlying crowd. It is important to generate tasks in a way that keeps the total number of HITs generated

---

<sup>1</sup>Other objective functions include maximizing answer quality or minimizing latency. Unfortunately, answer quality is hard to define (the correct answer to many human computation tasks cannot be known). Latency is highly variable, and probably better optimized through low-level optimizations like those used in quikTurkit [22] or Adrenaline [20].

<sup>2</sup>For sequences of conjunctive predicates, combining actually does more “work” on people than not combining, since tuples that may have been discarded by the first filter are run through the second filter as well. Still, as long as the first filter does not have 0 selectivity, this will reduce the total number of HITs that have to be run.

low. For example, as in a traditional database, it's better to filter tables before joining them. Query planning in Qurk is done in a way similar to conventional logical-to-physical query plan generation. A query or workflow is translated into a query plan/tree that processes input tables in a bottom-up fashion. Relational operations that can be performed by a computer rather than humans are pushed down the query plan as far as possible—including pushing non-HIT joins below HIT-based filters when possible—as the goal is to reduce the number of tasks performed by humans rather than reducing query/CPU time or disk utilization.

The system generates HITs for all non-join WHERE clause expressions first, and then as those expressions produce results, feeds them into join operators, which in turn produce HITs that are fed to successive operators. As described in Chapter 6, Qurk utilizes selectivity estimation to order filters and joins in the query plan.

# Chapter 5

## Human-powered Sorts and Joins

### 5.1 Introduction

In studying human computation workflows, one runs into repeated use of common data processing operations. Workflow developers often implement tasks that involve familiar database operations such as filtering, sorting, and joining datasets. For example, it is common for MTurk workflows to filter datasets to find images or audio on a specific subject, or rank such data based on workers' subjective opinion. Programmers currently waste considerable effort re-implementing these operations because reusable implementations do not exist. Furthermore, existing database implementations of these operators cannot be reused, because they are not designed to execute and optimize over crowd workers.

Reusable crowd-powered operators naturally open up a second opportunity for database researchers: operator-level optimization. Human workers periodically introduce mistakes, require compensation or incentives, and take longer than traditional silicon-based operators. Currently, workflow designers perform ad-hoc parameter tuning when deciding how many assignments of each HIT to post in order to increase answer confidence, how much to pay per task, and how to combine several human-powered operators (e.g., multiple filters) together into one HIT. These parameters are amenable to cost-based optimization, and introduce an exciting new landscape for query optimization and execution research.

In this chapter, we focus on the implementation of two of the most important database operators: joins and sorts. The human-powered versions of these operators are important because they appear in multiple use cases. For example, information integration and deduplication can be stated as a join between two datasets, one with canonical identifiers for entities, and the other with alternate identifiers. Human-powered sorts are widespread as well. Each time a user provides a rating, product review, or votes on a user-generated content website, they are contributing to a human-powered ORDER BY.

Sorts and joins are challenging to implement because there are a variety of ways they can be implemented as HITs. For example, to sort a list of images, we might ask users to compare groups of images. Alternatively, we might ask users for numerical

ratings for each image. We would then use the comparisons or scores to compute the order. The interfaces for sorting are quite different, require a different number of total HITs and result in different answers. Similarly, there are a variety of ways to issue HITs that compare objects for computing a join, and we study answer quality generated by different interfaces on a range of datasets.

In this chapter, we will explore how Qurk’s executor can choose the best implementation or user interface for different operators depending on the type of question or properties of the data. The Qurk executor combines human computation and traditional relational processing (e.g., filtering images by date before presenting them to the crowd). Finally, Qurk automatically translates queries into HITs and collects the answers in tabular form as they are completed by workers.

Besides describing implementation alternatives, we also explore optimizations to compute a result in a smaller number of HITs, which reduces query cost and sometimes latency. Specifically, we look at:

- *Batching*: We can issue HITs that ask users to process a variable number of records. Larger batches reduce the number of HITs, but may negatively impact answer quality or latency.
- *Worker agreement*: Workers make mistakes, disagree, and attempt to game the marketplace by doing a minimal amount of work. We evaluate several metrics to compute answer and worker quality, and inter-worker agreement.
- *Join pre-filtering*: There are often preconditions that must be true before two items can be joined. For example, two people are the same if and only if they have the same gender. We introduce a way for users to specify such filters (essentially human-powered classifiers), which require a linear pass by the crowd over each table being joined, but allow us to avoid a full cross-product when computing the join.
- *Hybrid sort*: When sorting, asking users to rate items requires fewer tasks than directly comparing pairs of objects, but produces a less accurate ordering. We introduce a hybrid algorithm that uses rating to roughly order items, and iteratively improves that ordering by using comparisons to improve the order of objects with similar ratings.

Our join optimizations result in more than an order-of-magnitude cost reduction, from \$67 to \$3, on a join of 30 photos with 30 other photos while maintaining result accuracy and latency. For sorts, we show that ranking (which requires a number of HITs linear in the input size) costs dramatically less than ordering (which requires a number of HITs quadratic in the input size), and produces comparable results in many cases. Finally, in an end-to-end test, we show that our optimizations can reduce by a factor of 14 the number of HITs required to join images of actors and rank-order them.



## 5.2 Join Operator

This section describes several implementations of the join operator, and the details of our feature filtering approach for reducing join complexity. We present a series of experiments to show the quality and performance of different join approaches.

### 5.2.1 Implementation

The join HIT interface asks a worker to compare elements from two joined relations. Qurk implements a block nested loop join, and uses the results of the HIT comparisons to evaluate whether two elements satisfy the join condition. We do not implement more efficient join algorithms (e.g., hash join or sort-merge join) because we do not have a way to compute item (e.g., picture) hashes for hash joins or item order for sort-merge joins.

The following screenshots and descriptions center around evaluating join predicates on images, but Qurk is not limited to image data types. The implementations generalize to any field type that can be displayed in HTML. In this section, we assume the two tables being joined are  $R$  and  $S$ , with cardinalities  $|R|$  and  $|S|$ , respectively.

**Is the same celebrity in the image on the left and the image on the right?**




Figure 5-1: Simple Join

## Simple Join

Figure 5-1 shows an example of a simple join predicate interface called SimpleJoin. In this interface, a single pair of items to be joined is displayed in each HIT along with the join predicate question, and two buttons (*Yes*, *No*) for whether the predicate evaluates to true or false. This simplest form of a join between tables  $R$  and  $S$  requires  $|R||S|$  HITs to be evaluated.

**Is the same celebrity in the image on the left and the image on the right?**

☐ Yes ☐ No



☐ Yes ☐ No




Figure 5-2: Naive Batching

## Naive Batching

Figure 5-2 shows the simplest form of join batching, called NaiveBatch. In NaiveBatch, we display several pairs vertically. *Yes*, *No* radio buttons are shown with each pair that is displayed. A *Submit* button at the bottom of the interface allows the worker to submit all of the pairs evaluated in the HIT. If the worker clicks *Submit* without having selected one of *Yes* or *No* for each pair, they are asked to select an option for each unselected pair.

For a batch size of  $b$ , where  $b$  pairs are displayed in each HIT, we can reduce the number of HITs to  $\frac{|R||S|}{b}$ .

## Find pairs of images with the same celebrity

- To select pairs, click on an image on the left and an image on the right. Selected pairs will appear in the **Matched Celebrities** list on the left.
- To magnify a picture, hover your pointer above it.
- To unselect a selected pair, click on the pair in the list on the left.
- If none of the celebrities match, check the **I did not find any pairs** checkbox.
- There may be multiple matches per page.

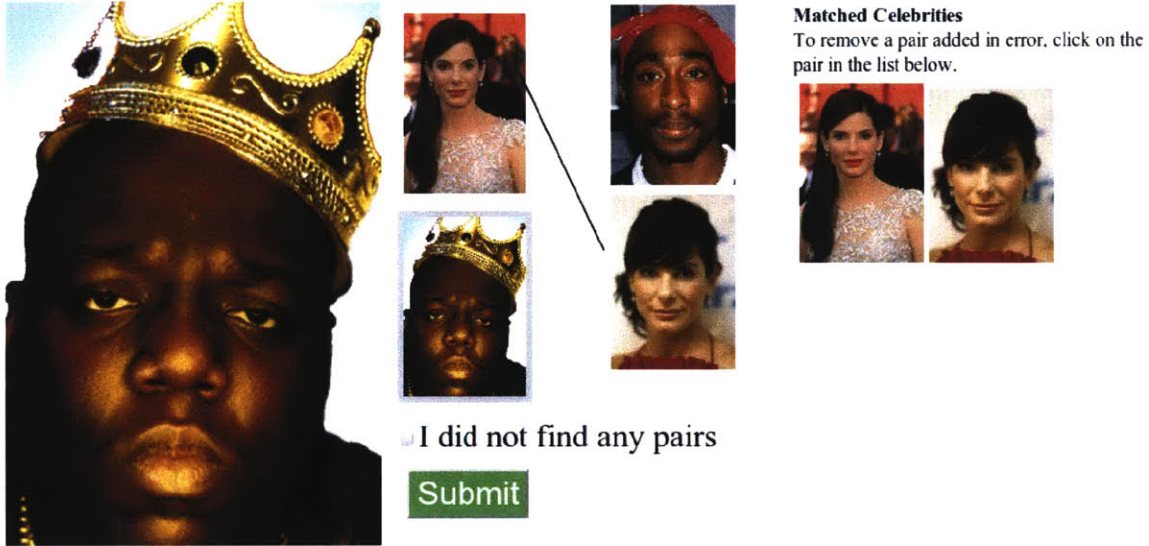


Figure 5-3: Smart Batching

## Smart Batching

Figure 5-3 shows a more complex join batching interface called SmartBatch. Two columns of images are displayed, and workers are asked to click on pairs of images that match the join predicate. The first column contains images from table  $R$  and the second contains images from table  $S$ .

Once a worker selects a pair, it is displayed in a list to the right of the columns, and can be removed (if added by mistake) by clicking on the pair. All selected pairs are connected by a line. A worker may match multiple images on the right that match with an image on the left or vice versa. If none of the images match the join predicate, the worker is asked to click a checkbox indicating no matches. In order to submit the HIT, the box must be checked or at least one pair must be selected.

To conserve vertical space, images are not displayed at full size. If a user hovers over an image, it is displayed at the size used in SimpleJoin and NaiveJoin (e.g., in Figure 5-3, the mouse is hovering over Notorious B.I.G, who is displayed at full size).

For  $r$  images in the first column and  $s$  in the second column, we must evaluate  $\frac{|R||S|}{rs}$  HITs.



## Alternative Join Algorithms

There are a number of alternative join algorithms we have not discussed. For example, we could ask workers to label each tuple with a unique identifier of the entity that it represents, and perform a traditional join on the identifier. Our goal is to understand the accuracy-cost tradeoffs of batching and combining, so these alternatives are outside this dissertation’s scope. Our results can still be used to benefit other join algorithms, and we use the idea of labeling tuples for the feature filtering optimization described in Section 5.2.2.

### 5.2.2 Feature Filtering Optimization

In Section 3.2, we introduced the `OPTIONALLY` clause to joins for identifying feature-based filters that may reduce the size of a join cross product. This clause allows the developer to specify that some features must be true for the join predicate to evaluate to true. For example, two profile images shouldn’t join unless they have the same gender, hair color, and skin color. These predicates allow us to only consider join pairs which match the extracted features.

We now explain the benefit of this filtering. To simplify our analysis, we assume that all filter features are uncorrelated, and that the filters do not emit the value `UNKNOWN`.

Suppose there are  $N$  `OPTIONALLY` clauses added to a join. Let  $F = \{F_1, \dots, F_N\}$ , where  $F_i$  is a set that contains the possible values for the feature being compared in `OPTIONALLY` clause  $i$ . For example, if the  $i$ th feature is *hairColor*,  $F_i = \{\text{black, brown, blond, white}\}$ . Let the probability that feature  $i$  (e.g., hair color) has value  $j$  (e.g., brown) in table  $X$  to be  $\rho_{Xij}$ . Then, for two tables,  $R$  and  $S$ , the probability that two random records from both tables match on feature  $i$  is:

$$\sigma_i = \sum_{j \in F_i} \rho_{Sij} \times \rho_{Rij}$$

In other words,  $\sigma_i$  is the *selectivity* of feature  $i$ . For example  $\sigma_{gender}$  across tables  $R$  and  $S$  is  $\rho_{Sfemale} \times \rho_{Rfemale} + \rho_{Smale} \times \rho_{Rmale}$ .

The selectivity of all expressions in the `OPTIONALLY` clauses of a join (assuming the features are independent) is:

$$Sel = \prod_{i \in [1 \dots N]} \sigma_i$$

Feature filtering causes the total number of join HITs that are executed to be a fraction  $Sel$  of what would be executed by a join algorithm alone. For example, if two features *gender* and *hairColor* appear in `OPTIONALLY` clauses for a join,  $Sel = \sigma_{gender} \times \sigma_{hairColor}$ . This benefit comes at the cost of running one linear pass over each table for each feature filter. Of course, the HITs in the linear pass can be batched through merging and combining.

In general, feature filtering is helpful, but there are three possible cases where we

may not want to apply a filter: 1) if the additional cost of applying the filter does not justify the reduction in selectivity it offers (e.g., if all of the people in two join tables of images have brown hair); 2) if the feature doesn't actually guarantee that two entities will not join (e.g., because a person has different hair color in two different images); or 3) if the feature is ambiguous (i.e., workers do not agree on its value).

To detect 1), we run the feature filter on a small sample of the data set and estimate selectivity, discarding filters that are not effective. We discuss how selectivity estimation is performed for filters in Chapter 6. To evaluate 2) for a feature  $f$ , we also use a sample of both tables, computing the join result  $j_{f-}$  with all feature filters except  $f$ , as well as the join result with  $f$ ,  $j_{f+}$ . We then measure the fraction  $\frac{|j_{f-} - j_{f+}|}{|j_{f-}|}$  and if it is below some threshold, we discard that feature filter clause from the join<sup>1</sup>.

For case 3) (feature ambiguity), we use a measure called inter-rater reliability (IRR), which measures the extent to which workers agree. As a quantitative measure of IRR, we utilize Fleiss'  $\kappa$  [39] (defined in Footnote 5). Fleiss'  $\kappa$  is used for measuring agreement between two or more raters on labeling a set of records with categorical labels (e.g., true or false). It is a number between -1 and 1, where a higher number indicates greater agreement. A  $\kappa$  of 0 roughly means that the ratings are what would be expected if the ratings had been sampled randomly from a weighted distribution, where the weights for a category are proportional to the frequency of that category across all records. For feature filters, if we measure  $\kappa$  to be below some small positive threshold for a given filter, we discard it from our filter set. In this research, we only evaluate whether  $\kappa$  is a good signal of worker agreement, but do not identify a particular cutoff for discarding features, as this depends on the users' level of comfort with worker response diversity for their application. Due to our use of Fleiss'  $\kappa$ , Qurk currently only supports detecting ambiguity for categorical features, although in some cases, range-valued features may be binned into categories.

### 5.2.3 Further Exploring the Design Space

In our experiments, we will explore the various interfaces and algorithms we have outlined. In order to fully understand where in the design space our experiments lie, it helps to identify other representative solutions that we have not explored but could make for interesting directions for future work.

**Alternative user interfaces.** In our experiments, we explore three user interfaces for eliciting pairs of matched items from crowd workers. While these designs help us explore workers' ability to batch-process data, they are not an exhaustive exploration of the design space.

A design that extends our batched interfaces would be one that displays items in a two-dimensional array, as one does with cards in memory games. So far, we have only batched images vertically, providing two columns of images that are to be matched. Adding more columns to this display might increase the batch size while

---

<sup>1</sup>This fraction is an accuracy measure that identifies the amount of join results that are eliminated by using feature  $f$ . We leave other error scores to future work, but have found this measure to work well in practice.

not significantly affecting accuracy. For this design to work, we need a pair selection mechanism that avoids drawing lines across the screen as we do in smart batching. One such design might involve adding selected pairs to a gutter to the side of the screen, or only showing lines between matched items when a user hovers over one of the items.

Another interface and algorithm design would involve asking users to find similar items rather than precise matches. As the system learns a “crowd kernel” [70] for similarity between items, it could potentially identify nearby items as strong candidates for matches.

**Game designs.** While the interfaces and interactions we design are designed for use on paid crowd platforms, we might design different interfaces if building for Games with a Purpose [73]-style interactions. The benefit of casting the entity resolution problem as a game is that one might be able to perform the matching for reduced or eliminated payments to workers, who are now incentivized by gameplay.

One gamelike design would cast the pair matching problem as a game of Memory. As players explore an obscured set of items to identify matching items two at a time, they reveal the matches to the system. Scoring the game would be difficult, as *a priori* we would not know whether a pair identified by a player was truly a pair or a mistake. To handle this, we could turn the game into a multiplayer experience, with players taking turns identifying pairs while the other players vote to verify each pair.

Another aspect of entity resolution that can be made into a game is feature extraction. Image tagging has already been cast as a two-player game in which players earn points by tagging an image with the same terms [73]. We could identify join candidates by looking for high-overlap images to use as candidates for join pairs.

**Machine learning.** In our study of human-powered joins, we have restricted our exploration to functionality that is strictly human-powered. Specifically, we are looking at the power of different user interfaces for identifying matched pairs, and exploring how to elicit hints about features that are necessary conditions for a pair of items to meet a join condition. In practice, we want to further reduce the amount of pairwise comparisons workers have to perform to reduce cost, and a natural method of doing this is to pre-process the two datasets with machine learning algorithms that identify high-probability candidates for matches. A recent paper by Wang et al. [76] continues our line of research by combining crowd matching and machine learning algorithms. In that work, the authors also show that learning algorithms can generate batches of similar items to increase the likelihood of matches in a batch that a worker is looking at.

The machine learning approach is relatively domain-specific. If performing entity resolution on text fields, one might look for fields that have high text overlap or sound like one-another when pronounced. For images, one could apply a suite of computer vision algorithms to identify similar items. Vision algorithms might further be domain-specific, as is the case with face detection algorithms.

The approaches we explore in this dissertation can layer with machine learning techniques for reducing crowd work. It is not always the case that an off-the-shelf automated matching algorithm for the problem a user is trying to solve will exist, whereas describing the matching problem to another human is a simpler task. Even if

an algorithm is identified, it might require training data before it can reach accuracy levels that are acceptable to the user. Finally, even after pairs are eliminated through a learning algorithm, the low-confidence negative or positive matches are still likely to need vetting by the crowd, and our join interfaces can power those scenarios.

## 5.2.4 Experiments

We now assess the various join implementations and the effects of batching and feature filtering. We also explore the quality of worker output as they perform more tasks.

### Dataset

In order to test join implementations and feature filtering, we created a *celebrity join dataset*. This dataset contains two tables. The first is `celeb(name text, img url)`, a table of known celebrities, each with a profile photo from IMDB<sup>2</sup>. The second table is `photos(id int, img url)`, with images of celebrities collected from People Magazine’s collection of photos from the 2011 Oscar awards.

Each table contains one image of each celebrity, so joining  $N$  corresponding rows from each table naively takes  $N^2$  comparisons, and has selectivity  $\frac{1}{N}$ .

### Join Implementations

In this section, we study the accuracy, price, and latency of the celebrity join query described in Section 3.2.3.

We run each of the join implementations twice (Trial #1 and #2) with five assignments for each comparison. This results in ten comparisons per pair. For each pair of trials, We ran one trial in the morning before 11 AM EST, and one in the evening after 7 PM EST, to measure variance in latency at different times of day. All assignments are priced at \$0.01, which costs \$0.015 per assignment due to Amazon’s half-cent HIT commission.

We use the two methods described in Section 3.2 to combine the join responses from each assignment. For **MajorityVote**, we identify a join pair if the number of positive votes outweighs the negative votes. For **QualityAdjust**, we generate a corpus that contains each pair’s *Yes*, *No* votes along with the Amazon-specified Turker ID for each vote. We execute Ipeirotis et al.’s worker quality algorithm [46] for five iterations on the corpus, and parametrize the algorithm to penalize false negatives twice as heavily as false positives (this gave us good results on smaller test runs, but the results are not terribly sensitive to these parameters).

**Baseline Join Algorithm Comparison:** We first verify that the three join implementations achieve similar accuracy in unbatched form. Table 5.1 contains the results of the joins of a sample of 20 celebrities and matching photos. The ideal algorithm results in 20 positive matches and 380 negative matches (pairs which do not join). The true positives and negatives for **MajorityVote** and **QualityAdjust** on all ten assignments per pair are reported with the prefixes MV and QA, respectively. From

---

<sup>2</sup><http://www.imdb.com>

Implementation	True Pos. (MV)	True Pos. (QA)	True Neg (MV)	True Neg (QA)
IDEAL	20	20	380	380
Simple	19	20	379	376
Naive	19	19	380	379
Smart	20	20	380	379

Table 5.1: Baseline comparison of three join algorithms with no batching enabled. Each join matches 20 celebrities in two tables, resulting in 20 image matches (1 per celebrity) and 380 pairs with non-matching celebrities. Results reported for ten assignments aggregated from two trials of five assignments each. With no batching enabled, the algorithms have comparable accuracy.

these results, it is evident that all approaches work fairly well, with at most 1 photo which was not correctly matched (missing true positive). We show in the next section that using QA and MV is better than trusting any one worker’s result.

**Effect of Batching:** In our next experiment, we look at the effect of batching on join quality and price. We compared our simple algorithm to naive batching with 3, 5, and 10 pairs per HIT and smart batching with a  $2 \times 2$  and  $3 \times 3$  grid, running a celebrity join between two images tables with 30 celebrity photos in each table. The answer quality results are shown in Figure 5-4. There are several takeaways from this graph.

First, all batching schemes except Smart 2x2, which performs as well as the Simple Join, do have some negative effect on the overall total number of true positives. When using QA, the effect is relatively minor with 1–5 additional false negatives on each of the batching schemes. There is not a significant difference between naive and smart batching. Batching does not significantly affect the overall true negative rate.

Second, QA does better than MV in improving true positive result quality on batched schemes. This is likely because QA includes filters for identifying spammers and sloppy workers, and these larger, batched schemes are more attractive to spammers that quickly and inaccurately complete tasks. The overall error rate between two trials of 5 assignments per pair was approximately the same. However, individual trials are more vulnerable to a small number of spammers, which results in higher variance in accuracy.

Third, MV and QA often achieve far higher accuracy as compared to the expected accuracy from asking a single worker for each HIT. In the Simple experiments, the expected true positive rate of an average worker was  $235/300 = 78\%$ , whereas MV was 93%. MV performed the worst in the Smart 3x3 experiments, yet still performed as well as the expected true positive rate of  $158/300 = 53\%$ . In all cases, QA performed significantly better.

We also determined the cost (in dollars) of running the complete join (900 comparisons) for the two trials (with 10 assignments per pair) at a cost of \$0.015 per assignment (\$0.01 to the worker, \$0.005 to Amazon). The cost of a naive join is thus  $900 \times \$0.015 \times 10 = \$135.00$ . The cost falls proportionally with the degree of batching



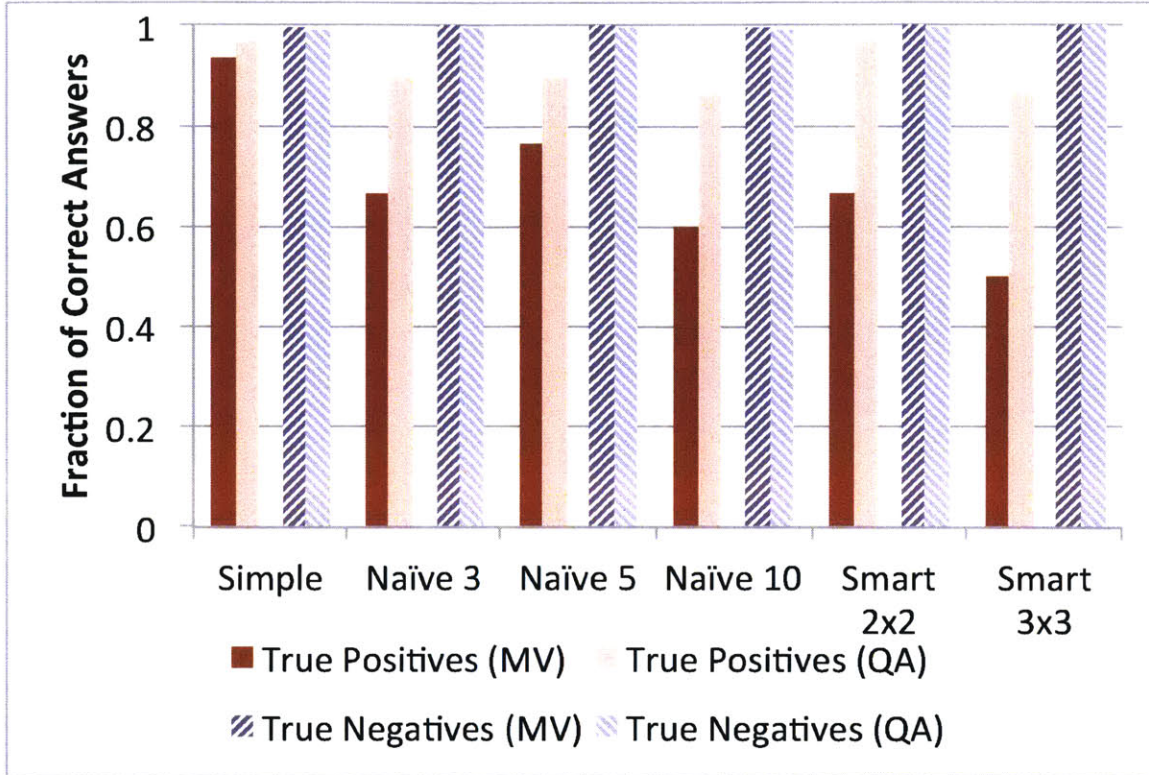


Figure 5-4: Fraction of correct answers on celebrity join for different batching approaches. Results reported for ten assignments aggregated from two runs of five assignments each. Joins are conducted on two tables with 30 celebrities each, resulting in 30 matches (1 per celebrity) and 870 non-matching join pairs.

(e.g., naive 10 reduces cost by a factor of 10, and a 3x3 join reduces cost by a factor of 9), resulting in a cost of around \$13.50.

Figure 5-5 shows end-to-end latency values for the different join implementations, broken down by the time for 50%, 95%, and 100% percent of the assignments to complete. We observe that a reduction in HITs with batching reduces latency, even though fewer HITs are posted and each HIT contains more work<sup>3</sup>. Both SimpleJoin trials were slower than all other runs, but the second SimpleJoin trial was particularly slow. This illustrates the difficulty of predicting latency in a system as dynamic as MTurk. Finally, note that in several cases, the last 50% of wait time is spent completing the last 5% of tasks. This occurs because the small number of remaining tasks are less appealing to Turkers looking for long batches of work. Additionally, some Turkers pick up and then abandon tasks, which temporarily blocks other Turkers from starting them. Many of these latency-based issues can be solved by applying the techniques used by quikTurkit [22].

<sup>3</sup>There are limits to this trend, but the latency generally goes down until workers are unwilling to perform tasks because they are too unfairly priced.

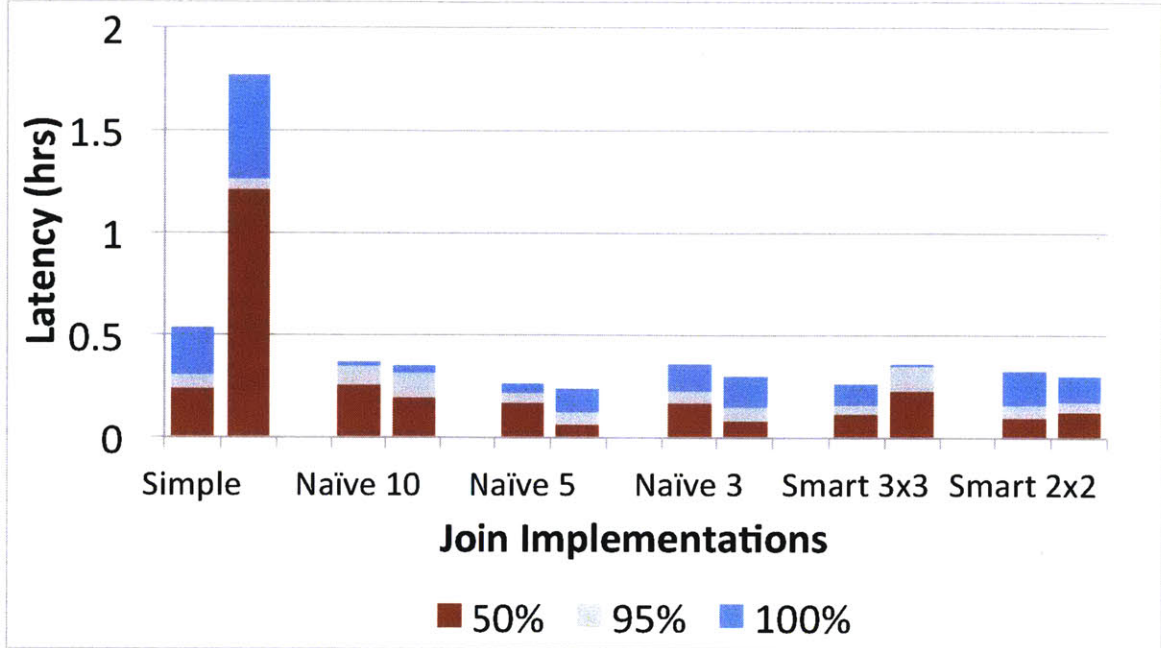


Figure 5-5: Completion time in hours of the 50<sup>th</sup>, 95<sup>th</sup>, and 100<sup>th</sup> percentile assignment for variants of celebrity join on two tables with 30 celebrities each.

### Assignments vs. Accuracy

One concern is that a worker’s performance will degrade as they execute more tasks and become bored or less cautious. This is a concern as our results (and those of CrowdDB [40]) suggest that a small number of workers complete a large fraction of tasks.

To test if the amount of work done by a worker is negatively correlated with work quality, we performed a linear regression. For a combination of responses to the two simple  $30 \times 30$  join tasks, we fit the number of tasks each worker did with their accuracy ( $\frac{\text{correct tasks}}{\text{tasks completed}}$ ), and found  $R^2 = 0.028$ ,  $p < .05$ . Accuracy and number of tasks are positively correlated (the slope,  $\beta$ , is positive), and the correlation explains less than 3% of variance in accuracy. This suggests no strong effect between work done and accuracy.

### Feature Filtering

Finally, we ran an experiment to measure the effectiveness of feature filtering. In this experiment, we asked workers to choose the hair color, skin color, and gender of each of the 60 images in our two tables. For each feature, we ran two trials with 5 votes per image in each trial, combining answers using majority vote. We also ran two trials with a combined interface where we asked workers to provide all three features at once.

Table 5.2 shows the effectiveness of applying feature filters in the four trials. We report the number of errors, which is the number of pairs that should have joined (of

Trial #	Combined?	Errors	Saved Comparisons	Join Cost
1	Y	1	592	\$27.52
2	Y	3	623	\$25.05
1	N	5	633	\$33.15
2	N	5	646	\$32.18

Table 5.2: Feature Filtering Effectiveness.

30) that didn’t pass all three feature filters. We then report the saved comparisons, which is the number of comparisons (of 870) that feature filtering helped avoid. We also report the total join cost with feature filtering. Without feature filters the cost would be \$67.50 for 5 assignments per HIT.

From these results, we can see that feature filters substantially reduce the overall cost (by more than a factor of two), and that combining features reduces both cost and error rate. The reason that combining reduces error rate is that in the batched interface, workers were much more likely to get hair color correct than in the non-batched interface. We hypothesize that this is because when asked about all three attributes at once, workers felt that they were doing a simple demographic survey, while when asked solely any one feature (in this case hair color), they may have overanalyzed the task and made more errors.

We now look at the error rate, saved comparisons, and total cost when we omit one feature from the three. The goal of this analysis is to understand whether omitting one of these features might improve join quality by looking at their effectiveness on a small sample of the data as proposed in Section 5.2.2. The results from this analysis on the first combined trial are shown in Table 5.3 (all of the trials had the same result). From this table, we can see that omitting features reduces the error rate, and that gender is by far the most effective feature to filter on. From this result, we conclude that hair color should potentially be left out. In fact, hair color was responsible for all of the errors in filtering across all trials.

Omitted Feature	Errors	Saved Comparisons	Join Cost
Gender	1	356	\$45.30
Hair Color	0	502	\$34.35
Skin Color	1	542	\$31.28

Table 5.3: Leave One Out Analysis for the first combined trial. Removing hair color maintains low cost and avoids false negatives.

To see if we can use inter-rater reliability as a method for determining which attributes are ambiguous, we compute the value of  $\kappa$  (as described in Section 5.2.2) for each of the attributes and trials. The results are shown in Table 5.4. From the table, we can see that the  $\kappa$  value for gender is quite high, indicating the workers generally agree on the gender of photos. The  $\kappa$  value for hair is much lower, because many of the celebrities in our photos have dyed hair, and because workers sometimes disagree about blond vs. white hair. Finally, workers agree more about skin color when it is

presented in the combined interface, perhaps because they may feel uncomfortable answering questions about skin color in isolation.

Table 5.4 displays the average and standard deviation of  $\kappa$  for 50 random samples with 25% of the celebrities in each trial. We see that these  $\kappa$  value approximations are near the true  $\kappa$  value in each trial, showing that Qurk can use early  $\kappa$  values to accurately estimate worker agreement on features without exploring the entire dataset. As the dataset size increases, we suspect a smaller percentage of samples will be required to achieve commensurate accuracy.

From this analysis, we can see that  $\kappa$  is a promising metric for automatically determining that hair color (and possibly skin color) should not be used as a feature filter.

Trial	Sample Size	Combined?	Gender $\kappa$ (std)	Hair $\kappa$ (std)	Skin $\kappa$ (std)
1	100%	Y	0.93	0.29	0.73
2	100%	Y	0.89	0.42	0.95
1	100%	N	0.85	0.43	0.45
2	100%	N	0.94	0.40	0.47
1	25%	Y	0.93 (0.04)	0.26 (0.09)	0.61 (0.37)
2	25%	Y	0.89 (0.06)	0.40 (0.11)	0.95 (0.20)
1	25%	N	0.85 (0.07)	0.45 (0.10)	0.39 (0.29)
2	25%	N	0.93 (0.06)	0.38 (0.08)	0.47 (0.24)

Table 5.4: Inter-rater agreement values ( $\kappa$ ) for features. For each trial, we display  $\kappa$  calculated on the entire trial’s data and on 50 random samples of responses for 25% of celebrities. We report the average and standard deviation for  $\kappa$  from the 50 random samples.

### 5.2.5 Limitations

One might wonder how well our findings generalize to other join applications. Our experiments so far have been run on a single image dataset, and so we caution against assuming that particular variable values (e.g., batch size) will be the same in experiments on other datasets. In fact, when we run a complete end-to-end experiment on a different dataset in Section 5.4, we see similar trends to the ones we have just discussed (i.e., batching reduces work, but eventually sees quality degradation and workers refusing to do complex tasks for \$0.01), but find that we are able to batch more images into the Smart Join implementation without losing quality.

We are unable to make any statements about the statistical significance of particular experiments, such as the ones behind Figure 5-4. While we re-ran experiments that generated the chart and found similar results, it is too expensive to run the experiments enough times to identify error bars for the chart. Still, once we apply `QualityAdjust` to remove spammers from the results, we do notice trends. The true negative rate never changes across tests, so all we focus on is the true positive rate.

After quality-adjusting, the true positive rate of any of the approaches sees a steady decrease as the batch size for that approach increases. These are takeaways that we expect will generalize to other experiments.

Similar caution should be taken with interpreting the feature filtering results. Worker agreement and leave-one-out analysis can serve as good fail-fast and effectiveness measures respectively, but significant future work remains in identifying the particular level of these signals that indicates desirable and undesirable results.

### 5.2.6 Summary

We found that for joins, batching is an effective technique that has small effect on result quality and latency, offering an order-of-magnitude reduction in overall cost. Naive and smart batching perform similarly, with smart 2x2 batching and QA achieving the best accuracy. In Section 5.4 we show an example of a smart batch run where a 5x5 smart batch interface was acceptable, resulting in a 25x cost reduction. We found that the QA scheme in [46] significantly improves result quality, particularly when combined with batching, because it effectively filters spammers. Finally, feature filtering offers significant cost savings when a good set of features can be identified. Putting these techniques together, we can see that for celebrity join, feature filtering reduces the join cost from \$67.50 to \$27.00. Adding batching can further reduce the cost by up to a factor of 10 on both feature filtering and joins, yielding a final cost for celebrity join of \$2.70.

## 5.3 Sort Operator

Users often want to perform a crowd-powered sort of a dataset, such as “order these variants of a sentence by quality,” or “order the images of animals by adult size.”

As with joins, the HITs issued by Qurk for sorting do not actually implement the sort algorithm, but provide an algorithm with information it needs by either: 1) comparing pairs of items to each other, or 2) assigning a rating to each item. The Qurk engine then sorts items using pairwise comparisons or their ratings.

### 5.3.1 Implementations

We now describe our two basic implementations of these ideas, as well as a hybrid algorithm that combines them. We also compare the accuracy and total number of HITs required for each approach.

#### Comparison-based

The comparison-based approach (Figure 5-6) asks workers to directly specify the ordering of items in the dataset. Our naive but accurate implementation of this approach uses up to  $\binom{N}{2}$  tasks per sort assignment, which is expensive for large datasets. While the worst-case number of comparisons is  $O(N \log N)$  for traditional sort algorithms, we now explain why we require more comparison tasks.



## There are 2 groups of squares. We want to order the squares in each group from smallest to largest.

- Each group is surrounded by a dotted line. Only compare the squares within a group.
- Within each group, assign a number from 1 to 7 to each square, so that:
  - 1 represents the smallest square, and 7 represents the largest.
  - We do not care about the specific value of each square, only the relative order of the squares.
  - Some groups may have less than 7 squares. That is OK: use less than 7 numbers, and make sure they are ordered according to size.
  - If two squares in a group are the same size, you should assign them the same number.

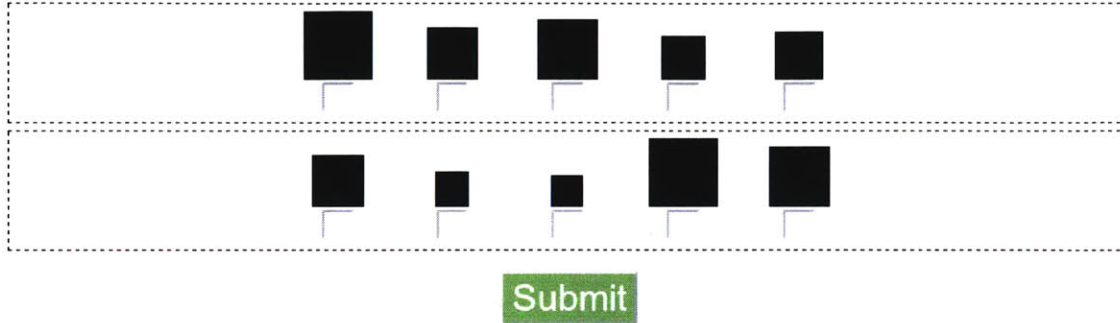


Figure 5-6: Comparison Sort

In practice, because these individual sort HITs are done by different workers, and because tasks themselves may be ambiguous, it can be the case that transitivity of pairwise orderings may not hold. For example, worker 1 may decide that  $A > B$  and  $B > C$ , and worker 2 may decide that  $C > A$ . One way to resolve such ambiguities is to build a directed graph of items, where there is an edge from item  $i$  to item  $j$  if  $i > j$ . We can run a cycle-breaking algorithm on the graph, and perform a topological sort to compute an approximate order. Alternatively, as we do in our implementation, we can compute the number of HITs in which each item was ranked higher than other items<sup>4</sup>. This approach, which we call “head-to-head,” provides an intuitively correct ordering on the data, which is identical to the true ordering when there are no cycles and workers provide correct responses.

Cycles also mean that we can not use algorithms like Quicksort that only perform  $O(N \log N)$  comparisons. These algorithms do not compare all elements, and yield unpredictable results in ambiguous situations (which we found while running our experiments). There has been a lot of theoretical work on pushing faulty comparator-based sorts toward  $O(N \log N)$  [18, 16, 67, 52]. It would be interesting to see how these techniques can be adapted to a human-in-the-loop framework.

Instead of comparing a single pair at a time, our interface, shown in Figure 5-6, displays groups of  $S$  items, and asks the worker to rank items within a group relative to one-another. When generating groups of size  $S$ , we ensure that a pairwise comparison appears in at least one group, but it is possible that in order to complete

<sup>4</sup>This approach breaks cycles by picking the most strongly agreed-upon comparisons. There are pathological scenarios, such as all workers providing the same cyclic ordering, which it can not solve.

**There are 2 squares below. We want to rate squares by their size.**

- For each square, assign it a number from 1 (smallest) to 7 (largest) indicating its size.
- For perspective, here is a small number of other randomly picked squares:

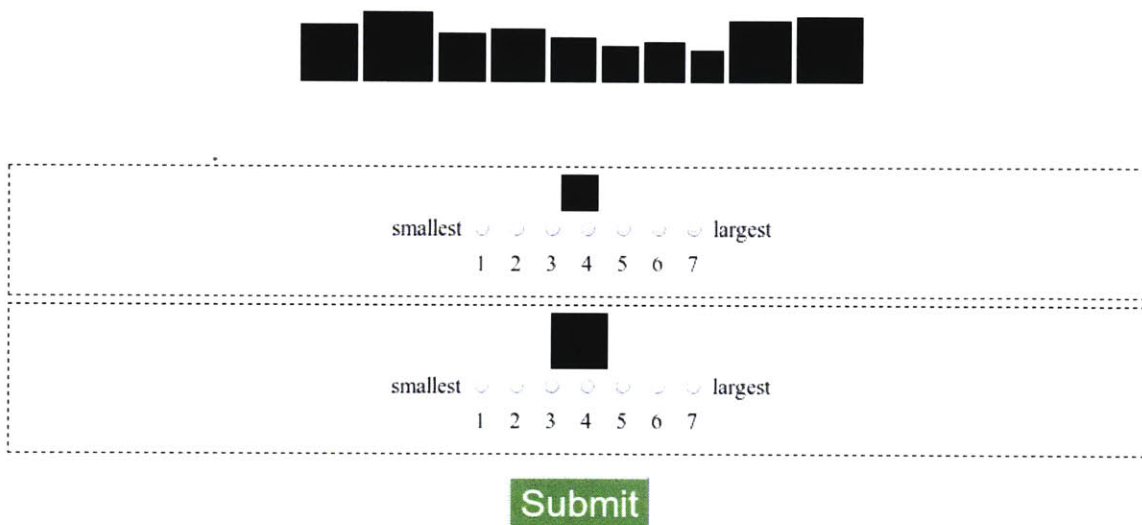


Figure 5-7: Rating Sort

a yet-unevaluated pair, we add other pairs to the grouping to reach an  $S$  items in the group. The result of each task is  $\binom{S}{2}$  pairwise comparisons, which reduces the number of HITs to  $\frac{N \times (N-1)}{S \times (S-1)}$ . Although the number of HITs is large, they can be executed in parallel. We can batch  $b$  such groups in a HIT to reduce the number of hits by a factor of  $b$ .

## Rating-based

The second approach is to ask workers to rate each item in the dataset along a numerical scale. We then compute the mean of all ratings for each item, and sort the dataset using these means.

Figure 5-7 illustrates the interface for a single rating task. The worker is shown a single item and asked to rate it along a seven-point Likert Scale [53], which is commonly used for subjective survey data. In order to provide context to assign a rating, we show ten randomly sampled images along the top of the interface. Showing a random selection allows us to give the worker a sense for the dataset without knowing its distribution *a priori*.

The advantage of this approach is that it requires closer to  $O(N)$  HITs. We can batch  $b$  ratings in a HIT to reduce the number of HITs by a factor of  $b$ . The variance of the rating can be reduced by asking more workers to rate the item. The drawback is that each item is rated independently of other items, and the relative ordering of an item pair's mean ratings may not be fully consistent with the ordering that would

result if workers directly compared the pair.

## Hybrid Algorithm

We now propose a hybrid approach that initially orders the data using the rating-based sort and generates a list  $L$ . Each item  $l_i \in L$  has an average rating  $\mu_i$ , as well as a standard deviation  $\sigma_i$  computed from votes used to derive the rating. The idea of our hybrid approach is to iteratively improve  $L$  by identifying subsets of  $S$  items that may not be accurately ordered and using the comparison-based operator to order them. The user can control the resulting accuracy and cost by specifying the number of iterations (where each iteration requires one additional HIT) to perform.

We explored several techniques for selecting size- $S$  windows for comparisons. We outline three representative approaches:

**Random:** In each iteration, pick  $S$  items randomly from  $L$ .

**Confidence-based:** Let  $w_i = \{l_i, \dots, l_{i+S-1}\}$ , meaning  $w_i$  contains the  $S$  consecutive items  $l_j \in L$  starting from item  $l_i$ . For each pair of items  $a, b \in w_i$ , we have their rating summary statistics  $(\mu_a, \sigma_a)$  and  $(\mu_b, \sigma_b)$ . For  $\mu_a < \mu_b$ , we compute  $\Delta_{a,b}$ , the difference between one standard deviation above  $\mu_a$  and one standard deviation below  $\mu_b$ , where  $\Delta_{a,b} = \max(\mu_a + \sigma_a - \mu_b - \sigma_b, 0)$ . Note that  $\Delta_{a,b}$  is only greater than 0 when there is at least a one standard deviation overlap between items  $a$  and  $b$ . For all windows  $w_i$ , we then compute  $R_i = \sum_{(a,b) \in w_i} \Delta_{a,b}$  and order windows in decreasing order of  $R_i$ , such that windows with the most standard deviation overlap, and thus least confidence in their ratings, are reordered first.

**Sliding window:** The algorithm picks a window  $w_i = \{l_{i \bmod |L|}, \dots, l_{(i+S) \bmod |L|}\}$  with  $i$  starting at 1. In successive iterations,  $i$  is incremented by  $t$  (e.g.,  $i = (i + t)$ ), which the  $\bmod$  operation keeps the range in  $[1, |L|]$ . If  $t$  is not a divisor of  $L$ , when successive windows wrap around  $L$ , they will be offset from the previous passes.

### 5.3.2 Further Exploring the Design Space

In our experiments, we will explore the various interfaces and algorithms we have outlined. In order to fully understand where in the design space our experiments lie, it helps to identify other representative solutions that we have not explored but could make for interesting directions for future work.

**Interface details.** While we presented a single interface for each of comparison- and rating-based sorts, the details of each interface have several variations.

The rating-based interface is a combination of several design choices. While rating on a seven-point Likert scale is common for subjective measures, it would be useful to see if adding granularity increases accuracy or overwhelms workers. Traditional Likert scales are designed to evaluate declarative statements (e.g., “This animal has a large adult size”), whereas our design was less strict (e.g., “Rate this animal by its adult size”). Testing variations of the rating question might make for interesting future work. Another variation involves the ten randomized example images we display to



workers at the top of each rating task. We can explore displaying different amounts of these images in addition to biasing the sample toward images that early on seem to be extremely low- or high-ranking.

The comparison-based interface has fewer variations than the rating-based one. The most interesting of the design variations are around the interaction technique by which workers order the items being compared. Our interface has users assign numbers indicating the relative ranking of the items in a batch, but a draggable interface might make it easier for workers to interact with more items at a time.

**Alternative algorithmic approaches.** There are fundamentally two inputs to a ranking algorithm: ratings and comparisons. We proposed one hybrid algorithm that compares nearby items after ranking by worker-provided ratings, but other hybrids exist. For example, we could recursively rate items, applying more ratings to items that are more uncertain in order to better discern the differences between them. Similarly, we could design a comparison-based interface that has workers broadly group items quickly (e.g., “generally large animals,” “medium-sized animals,” and “generally small animals”) before applying finer-granularity comparisons within those groupings.

**Blocking approaches.** Whenever a crowd-powered algorithm goes to the crowd for input that it must then process before generating more crowd work, the algorithm creates an artificial barrier to parallelism, and thus sees increased latency. Our algorithms have no such blocking steps, as they generate rating and all-pairs comparison tasks for all items at once. The hybrid algorithms we explore require blocking steps between the rating stage and the comparison stage, and potentially require several comparison stages. While quicksort-like algorithms might reduce the generated crowd work required to sort  $N$  items to  $O(N \log N)$ , they might introduce  $O(\log N)$  blocking stages if implemented naively.

### 5.3.3 Experiments

We now describe experiments that compare the performance and accuracy effects of the **Compare** and **Rate** sort implementations, as well as the improvements of our **Hybrid** optimizations.

The experiments compare the relative similarity of sorted lists using Kendall’s Tau ( $\tau$ ), which is a measure used to compute rank-correlation [50]. We use the  $\tau$ -b variant, which allows two items to have the same rank order. The value varies between -1 (inverse correlation), 0 (no correlation), and 1 (perfect correlation).

For each pair in **Compare**, we obtain at least 5 comparisons and take the majority vote of those comparisons. For each item in **Rate**, we obtain 5 scores, and take the mean of those scores. We ran two trials of each experiment.

### Datasets

The *squares dataset* contains a synthetically generated set of squares. Each square is  $n \times n$  pixels, and the smallest is  $20 \times 20$ . A dataset of size  $N$  contains squares of sizes

$\{(20 + 3 * i) \times (20 + 3 * i) | i \in [0, N)\}$ . This dataset is designed so that the sort metric (square area) is clearly defined, and we know the correct ordering.

The *animals dataset* contains 25 images of randomly chosen animals ranging from ants to humpback whales. In addition, we added an image of a rock and a dandelion to introduce uncertainty. This is a dataset on which comparisons are less certain, and is designed to show relative accuracies of comparison and rating-based operators. We will test various sort orders on this dataset, including adult size and dangerousness.

## Square Sort Microbenchmarks

In this section, we compare the accuracy, latency, and price for the query described in Section 3.2.2, in which workers sort squares by their size.

**Comparison batching.** In our first experiment, we sort a dataset with 40 squares by size. We first measure the accuracy of **Compare** as the group size  $S$  varies between 5, 10, and 20. Batches are generated so that every pair of items has at least 5 assignments. Our batch-generation algorithm may generate overlapping groups, so some pairs may be shown more than 5 times. The accuracy is perfect when  $S = 5$  and  $S = 10$  ( $\tau = 1.0$  with respect to a perfect ordering). The rate of workers accepting the tasks dramatically decreases when the group size is above 10 (e.g., the task takes 0.3 hours with group size 5, but more than 1 hour with group size 10.) We stopped the group size 20 experiment after several hours of uncompleted HITs. We discuss this effect in more detail, and ways to deal with it, in Section 5.5.

**Rating batching.** We then measure the accuracy of the **Rate** implementation. The interface shows 10 sample squares, sampled randomly from the 40, and varies the batch size from 1 to 10, requiring 40 to 4 HITs, respectively. In all cases, the accuracy is lower than **Compare**, with an average  $\tau$  of 0.78 (strong but not perfect ranking correlation) and standard deviation of 0.058. While increasing batch size to large amounts made HITs less desirable for Turkers and eventually increased latency, it did not have a noticeable effect on accuracy. We also found that 5 assignments per HIT resulted in similar accuracy to 10 assignments per HIT, suggesting diminishing returns for this task.

**Rating granularity.** Our next experiment is designed to measure if the granularity of the seven-point Likert scale affects the accuracy of the ordering as the number of distinct items increases. We fix the batch size at 5, and vary the size of the dataset from 20 to 50 in increments of 5. The number of HITs vary from 4 to 10, respectively. As with varying batch size, the dataset size does not significantly impact accuracy (avg  $\tau$  0.798, std 0.042), suggesting that rating granularity is stable with increasing dataset size. While combining 10 assignments from two trials did reduce  $\tau$  variance, it did not significantly affect the average.

## Query Ambiguity: Sort vs. Rank

The square sort microbenchmarks indicate that **Compare** is more accurate than **Rate**. In our next experiment, we compare how increasing the ambiguity of sorting tasks affects the accuracy of **Rate** relative to **Compare**. The goal is to test the utility of

metrics that help predict 1) if the sort task is feasible at all, and 2) how closely **Rate** corresponds to **Compare**. The metric we use to answer 1) is a modified<sup>5</sup> version of Fleiss’  $\kappa$  (which we used for inter-rater reliability in joins), and the metric to answer 2) is  $\tau$  (described in Section 5.3.3. The experiment uses both the *squares* and *animals* datasets.

We generated five queries that represent five sort tasks:

**Q1:** Sort squares by size

**Q2:** Sort animals by “Adult size”

**Q3:** Sort animals by “Dangerousness”

**Q4:** Sort animals by “How much this animal belongs on Saturn”

**Q5:** (Artificially) generate random **Compare** and **Rate** responses.

The instructions for Q3 and 4 were deliberately left open-ended to increase the ambiguity. Q4 was intended to be a nonsensical query that we hoped would generate random answers. As we describe below, the worker agreement for Q4’s *Compare* tasks was higher than Q5, which suggests that even for nonsensical questions, workers will apply and agree on some preconceived sort order.

For lack of objective measures, we use the **Compare** results as ground truth. The results of running **Compare** on queries 2, 3, and 4 are as follows:

**Size:** ant, bee, flower, grasshopper, parrot, rock, rat, octopus, skunk, tazmanian devil, turkey, eagle, lemur, hyena, dog, komodo dragon, baboon, wolf, panther, dolphin, elephant seal, moose, tiger, camel, great white shark, hippo, whale

**Dangerousness:** flower, ant, grasshopper, rock, bee, turkey, dolphin, parrot, baboon, rat, tazmanian devil, lemur, camel, octopus, dog, eagle, elephant seal, skunk, hippo, hyena, great white shark, moose, komodo dragon, wolf, tiger, whale, panther

**Belongs on Saturn**<sup>6</sup>: whale, octopus, dolphin, elephant seal, great white shark, bee, flower, grasshopper, hippo, dog, lempur, wolf, moose, camel, hyena, skunk, tazmanian devil, tiger, baboon, eagle, parrot, turkey, rat, panther, komodo dragon, ant, rock

---

<sup>5</sup>Fleiss’  $\kappa$  is defined as

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e}.$$

Strictly speaking, only  $\bar{P}$ , the average probability of agreement per category, measures agreement between workers. The term  $\bar{P}_e$  is a measure of the skew in categories to compensate for bias in the dataset (e.g., if there are far more small animals than big animals). Because of how we generated squares (i.e., squares with a smaller integer identifier were smaller) and internally represented pairs for comparison (i.e., (**smaller\_id**, **larger\_id**)), we generated a scenario where, if workers provided an absolutely correct response, they would achieve  $\bar{P}_e$  of 1. This is because all generated pairs have the correct category **less\_than**. (Note that these idiosyncracies did not affect workers’ user experiences because we randomized the visual order of images when displaying them to the user). In practice, a well-randomized comparison operation would see roughly half of pairs in the **less\_than** category and roughly the other half in the **greater\_than** category, resulting in a  $\bar{P}_e$  of .5. To get meaningful  $\kappa$  values for our synthetic data and have comparable  $\kappa$  values on the non-synthetic data, all  $\kappa$  values reported in our sorting experiments are simply  $\kappa = \bar{P}$ , the average worker agreement, unadjusted for bias in the categories.

<sup>6</sup>Note that while size and dangerousness have relatively stable orders, the Saturn list varies drastically as indicated by low  $\kappa$ . For example, in three runs of the query, **rock** appeared at the end, near the beginning, and in the middle of the list.

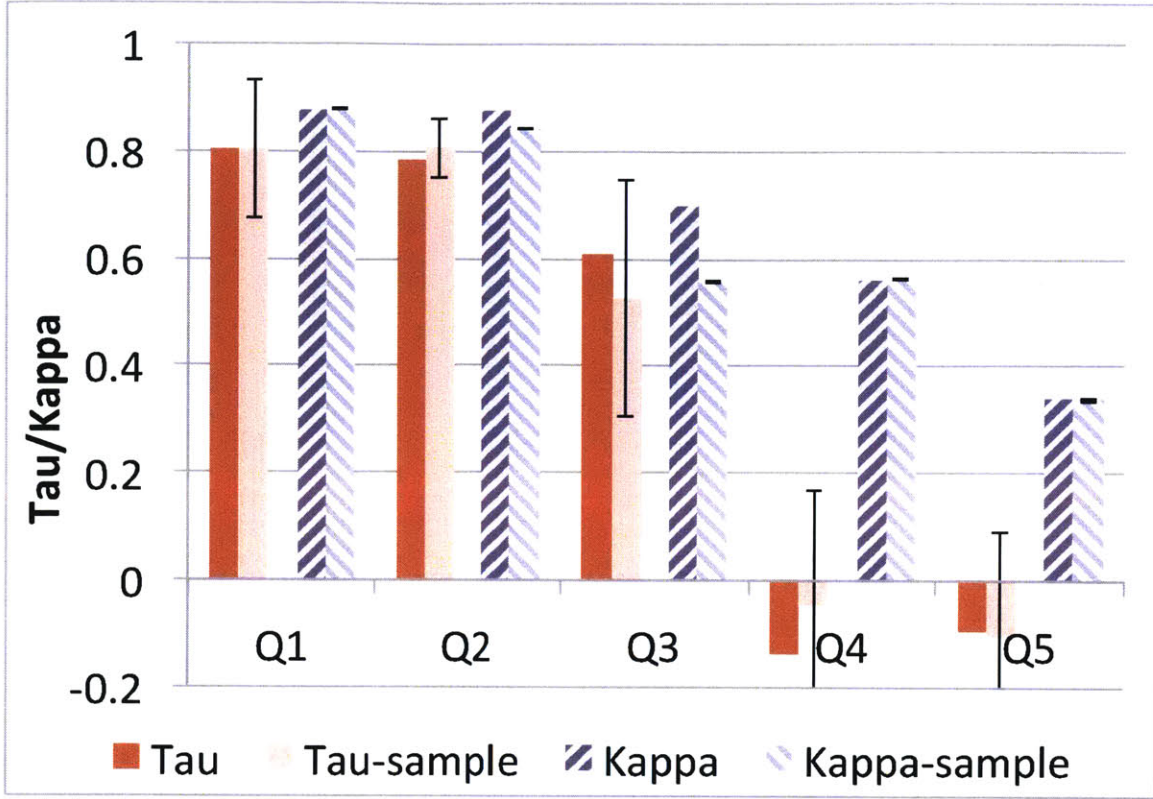


Figure 5-8:  $\tau$  and  $\kappa$  metrics on 5 queries.

Figure 5-8 show  $\tau$  and  $\kappa$  for each of the five queries. Here  $\kappa$  is computed on the comparisons produced by the **Compare** tasks. The figure also shows (in columns labeled **Kappa-sample** and **Tau-sample**) the effect of computing these metrics on a random sample of 10 of the squares/animals rather than the entire data set (the sample bars are from 50 different samples; error bars show the standard deviation of each metric on these 50 samples.)

The results show that the ambiguous queries have progressively less worker agreement ( $\kappa$ ) and progressively less agreement between comparing and rating ( $\tau$ ). While  $\kappa$  decreases between Q3 and Q4 (dangerousness and Saturn), it is not as low in Q4 as it is in Q5 (Saturn and random). While there is little agreement between workers on animals that belong on Saturn, their level of agreement is better than random. For example, **Komodo Dragon** was consistently rated as belonging to Saturn’s environment. The decrease in  $\kappa$  with increasing ambiguity suggests that  $\kappa$  is a useful signal in identifying unsortable datasets. This feature, combined with the fact we will soon show that we can accurately sample  $\kappa$  on a small subset of the data, suggests that a sampled  $\kappa$  is a good fail-fast metric for stopping queries that are buggy or intrinsically nonsensical.

$\tau$  is significantly lower for Q4 than for Q3, which suggests that ordering by rating does not work well for Q4, at least given the number of ratings we collected, and that we should probably use the **Compare** method for this workload rather than the **Rate**

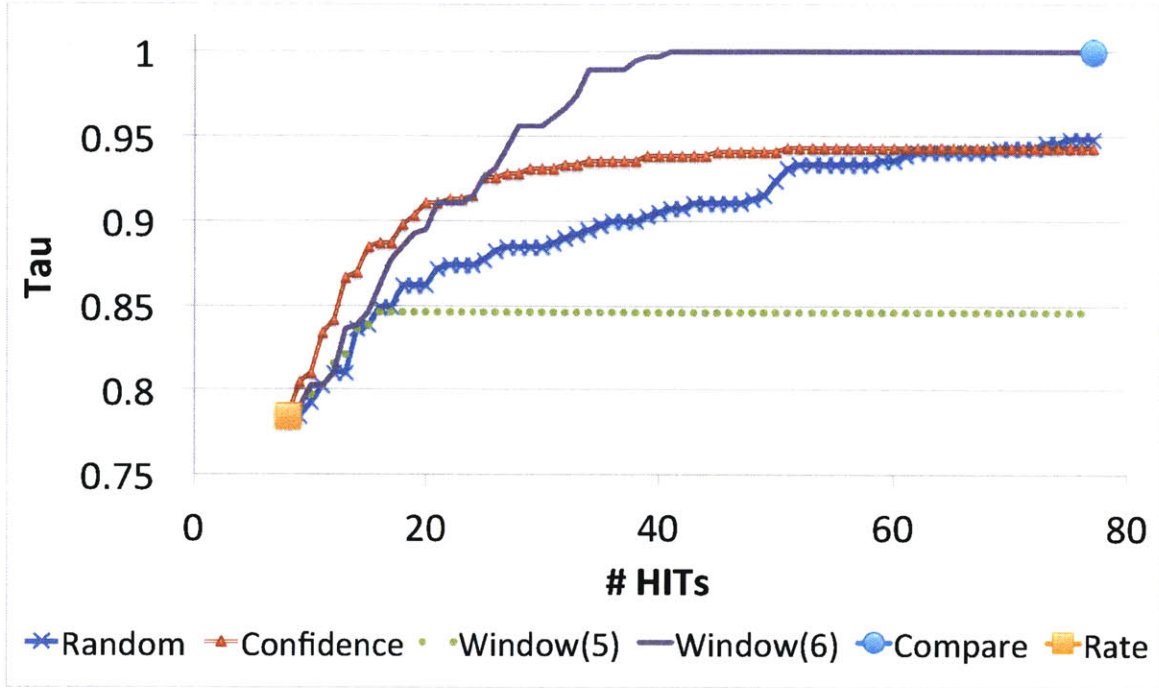


Figure 5-9: Hybrid sort algorithms on the 40-square dataset.

method. For Q1, Q2, and Q3, however, **Rate** agrees reasonably well with **Compare**, and because it is significantly cheaper, may be a better choice.

Finally, we note that sampling 10 elements is an effective way to estimate both of these metrics (in Figure 5-8, the bars for  $Tau/Tau-sample$  and  $Kappa/Kappa-sample$  are very close). This suggests a pattern for running sort routines. Any time we wish to sort, we can run both **Rate** and **Compare** on samples, and compute  $\tau$  and  $\kappa$ . We can then decide, for this particular sort query, whether to order the rest of the data set with **Rate** or **Compare** depending on  $\tau$ . We can also decide to stop ordering outright if the sample of  $\kappa$  is very low.

## Hybrid Approach

Our final set of experiments measure how the hybrid approaches perform in terms of accuracy with increasing number of HITs. We aim to understand how the sort order of hybrid changes between **Rank** quality and **Compare** quality with each additional HIT.

The first experiment uses the 40-square dataset. The comparison interface shows 5 items at a time. We set window size  $S = 5$  to be consistent with the number of items in a single comparison HIT. Figure 5-9 shows how  $\tau$  improves with each additional HIT. **Compare** (upper right circle) orders the list perfectly, but costs 78 HITs to complete. In contrast, **Rate** (lower left square) achieves  $\tau = 0.78$ , but only costs 8 HITs (batch size=5). In addition to these two extremes, we compared four schemes, based on those described in Section 5.3.1: random, confidence-based, windowing with



$t = 5$  (Window 5), and windowing with  $t = 6$  (Window 6).

Overall, Window 6 performs best, achieving  $\tau > .95$  in under 30 additional HITs, and converging to  $\tau = 1$  in half the HITs that **Compare** requires. Window 5 performs poorly because  $t$  is a multiple of the number of squares, so multiple passes over the data set (beyond the 8th HIT) do not improve the ordering. Because Window 6’s  $t$ -value (the amount we shift the window) of 6 is relatively prime with the window size of 5, Window 6 is able to move items that are globally out of their sort order, whereas Window 5 is unable to shift items beyond a local ordering.

As the list becomes more ordered, random is more likely to compare items that are correctly ordered, and thus wastes comparisons. Confidence does not perform as well as Window 6—prioritizing high-variance regions assists with fixing local sort order mistakes, but does not systematically move items that are far from their correct positions. In the sliding window scheme, after several passes through the dataset items that were far away from their correct position can migrate closer to the correct location.

Finally, we executed Q2 (animal size query) using the hybrid scheme and found similar results between the approaches. Ultimately, the window-based approach performed the best and improved  $\tau$  from .76 to .90 within 20 iterations.

### 5.3.4 Limitations

We ran most of our experiments on two datasets: 1) a synthetic squares dataset, and 2) a more realistic animal image dataset. We were able to identify comparison-based sorting as superior in quality to rating-based sort using only the synthetic dataset, so we can make no comments on how strong the improvement would be on less clearly sortable datasets. Still, the difference between the quality of comparison and rating (comparison did perfectly, and rating was around .2 lower on the  $\tau$ -b scale) suggests that we should expect a similar trend on other datasets.

Due to how expensive it is to run these experiments, it is also not possible to get enough data for statistical significance on experiments such as the one in Figure 5-8. Still, we are able to re-run the experiment and find the same trend remains: sort problems that are more ambiguous result in lower inter-rater agreement, and see less agreement between the comparison and rating tasks. Additionally, the sampled  $\kappa$  and  $\tau$  scores indicate that we can accurately sample these statistics to have them serve as early fail-fast mechanisms when running queries.

Finally, we can not make a claim as to what the best sorting algorithm is. In part, this depends on a user’s expectations for cost and quality. Additionally, our hybrid algorithm suggests that there is a good space between rating-based sorts’ reasonable accuracy and comparison-based sorts’ seemingly perfect accuracy. It remains an open question whether we can develop a lower-complexity comparison-based sort algorithm that achieves the same level of quality.

Operator	Optimization	# HITs
Join	Filter	43
Join	Filter + Simple	628
Join	Filter + Naive	160
Join	Filter + Smart 3x3	108
Join	Filter + Smart 5x5	66
Join	No Filter + Simple	1055
Join	No Filter + Naive	211
Join	No Filter + Smart 5x5	43
Order By	Compare	61
Order By	Rate	11
Total (unoptimized)		1055 + 61 = 1116
Total (optimized)		66 + 11 = 77

Table 5.5: Number of HITs for each operator optimization.

### 5.3.5 Summary

We presented two sort interfaces and algorithms based on ratings (linear complexity) and comparisons (quadratic complexity). We found that batching is an effective way to reduce the complexity of sort tasks in both interfaces. We found that while significantly cheaper, ratings achieve sort orders close to but not as good as comparisons. Using two metrics,  $\tau$  and a modified  $\kappa$ , we were able to determine when a sort was too ambiguous ( $\kappa$ ) and when rating performs commensurate with comparison ( $\tau$ ).

Using a hybrid window-based approach that started with ratings and refined with comparisons, we were able to get similar ( $\tau > .95$ ) accuracy to sorts at less than one-third the cost.

## 5.4 End to End Query

In the previous sections, we examined different operator optimizations in isolation. We now execute a complex query that utilizes joins and sorts, and show that our optimizations reduce the number of HITs by a factor of 14.5 $\times$  compared to a naive approach.

### 5.4.1 Experimental Setup

The query joins a table of movie frames and a table of actor photos, looking for frames containing the actor. For each actor, the query finds frames where the actor is the main focus of the frame and orders the frames by how flattering they are:

```
combined =
  JOIN actors, scenes
  ON inScene(actors.img, scenes.img)
```

```

AND OPTIONALLY numInScene(scenes.img) > 1;
ordered =
ORDER combined BY name, quality(scenes.img);

```

The query uses three crowd-based UDFs:

**numInScene**, a generative UDF that asks workers to select the number of people in the scene given the options (0, 1, 2, 3+, UNKNOWN). This UDF was designed to reduce the number of images input into the join operator.

**inScene**, an EquiJoin UDF that shows workers images of actors and scenes and asks the worker to identify pairs of images where the actor is the main focus of the scene.

**quality**, a Rank UDF that asks the worker to sort scene images by how flattering the scenes are. This task is highly subjective.

We tried several variants of each operator. For the **numInScene** filter we executed join feature filtering with batch size 4. We also tried disabling the operator and allowing all scenes to be input to the join operator. For the **inScene** join, we use Simple, Naive batch 5, and Smart batch 3×3 and 5×5. For the **quality** sort, we used **Compare** with group size 5, and **Rate** batch 5.

For the dataset, we extracted 211 stills one second apart from a three-minute movie. Actor profile photos came from the Web.

## 5.4.2 Results

The results are summarized in Table 5.5. The bottom two lines show that a simple approach based on a naive, unfiltered join plus comparisons requires 1116 hits, whereas applying our optimizations reduces this to 77 hits. We make a few observations:

**Smart Join:** Surprisingly, we found that workers were willing to complete a 5x5 SmartJoin, despite its relative complexity. This may suggest that SmartJoin is preferred to naive batching.

**Feature Extraction:** We found that the benefit of **numInScene** feature extraction was outweighed by its cost, as the selectivity of the predicate was only 55%, and the total number of HITS required to perform Smart Join with a 5x5 grid was relatively small. This illustrates the need for online selectivity estimation to determine when a crowd-based predicate will be useful.

**Query Accuracy:** The **numInScene** task was very accurate, resulting in no errors compared to a manually-evaluated filter. The **inScene** join did less well, as some actors look similar, and some scenes showed actors from the side; we had a small number of false positives, but these were consistent across join implementations. Finally, the scene **quality** operator had high variance and was quite subjective.

## 5.5 Takeaways and Discussion

**Reputation:** While not directly related to database implementation, it is important to remember that employer identity carries reputation on systems such as MTurk. Turkers keep track of good and bad requesters, and share this information on message



boards such as Turker Nation<sup>7</sup>. By quickly approving completed work and responding to Turker requests when they contact you with questions, you can generate a good working relationship with Turkers.

When we started as requesters, Turkers asked on Turker Nation if others knew whether we were trustworthy. A Turker responded:

[requester name] is okay .... I don't think you need to worry. He is great on communication, responds to messages and makes changes to the Hits as per our feedback.

Turker feedback is also a signal for price appropriateness. For example, if a requester overpays for work, Turkers will send messages asking for exclusive access to their tasks.

**Choosing Batch Size:** We showed that batching can dramatically reduce the cost of sorts and joins. In studying different batch sizes, we found batch sizes at which workers refused to perform tasks, leaving our assignments uncompleted for hours at a time. As future work, it would be interesting to compare adaptive algorithms for estimating the ideal batch size. Briefly, such an algorithm performs a binary search on the batch size, reducing the size when workers refuse to do work or accuracy drops, and increasing the size when no noticeable change to latency and accuracy is observed.

As a word of caution, the process of adaptively finding the appropriate batch sizes can lead to worker frustration. The same Turker that initially praised us in Section 5.5 became frustrated enough to list us on Turker Nation's "Hall of Shame:"

These are the "Compare celebrity pictures" Hits where you had to compare two pictures and say whether they were of the same person. The Hit paid a cent each. Now there are 5 pairs of pictures to be checked for the same pay. Another Requester reducing the pay drastically.

Hence, batching has to be applied carefully. Over time, ideal starting batch sizes can be learned for various media types, such as joins on images vs. joins on videos.

**Scaling Up Datasets:** In our sort and join experiments, we described techniques that provide order-of-magnitude cost reductions in executing joins and sorts. Still, scaling datasets by another order of magnitude or two would result in prohibitive costs due to the quadratic complexity of both join and sort tasks. Hence, one important area of future work is to integrate human computation and machine learning, training classifiers to perform some of the work, and leaving humans to perform the more difficult tasks. Another area to explore with larger datasets is the asymptotic trends in our findings: how do the various sampling- and hybrid-based approaches fare as the number of items increases?

## 5.6 Conclusion

We presented several approaches for executing joins and sorts in a declarative database where humans are employed to process records. For join comparisons, we developed

---

<sup>7</sup><http://turkers.proboards.com/>

three different UIs (simple, naive batching, and smart batching), and showed that the batching interfaces can reduce the total number of HITs to compute a join by an order of magnitude. We showed that feature filtering can pre-filter join results to avoid cross products, and that our system can automatically select the best features. We presented three approaches to sorting: comparison-based, rating-based, and a hybrid of the two. We showed that rating often does comparably to pairwise comparisons, using far fewer HITs, and presented metrics  $\kappa$  and  $\tau$  that can be used to determine if a dataset is sortable, and how well rating performs relative to comparison. We also present a hybrid scheme that uses a combination of rating and comparing to produce a more accurate result than rating while using fewer HITs than comparison. We showed on several real-world datasets that we can greatly reduce the total cost of queries without sacrificing accuracy – we reduced the cost of a join on a celebrity image data set from \$67 to about \$3, a sort by  $2\times$  the worst-case cost, and reduced the cost of an end-to-end example by a factor of 14.5.

# Chapter 6

## Counting with the Crowd

### 6.1 Introduction

We have just shown how to implement sort and join operators in crowd databases, and the costs of particular operator implementations. Given these costs, one fundamental problem that remains to be addressed to perform optimization in a crowd-powered database is that of *selectivity estimation*. In selectivity estimation, we are given an expression or predicate, and we must estimate the number of results that will satisfy the expression. Given cost estimates for individual operators, input cardinalities, and selectivity estimates, standard cost-based optimization techniques can be used to estimate overall query cost, allowing a query optimizer to reorder operators to reduce cost.

In this chapter, we study how to estimate the selectivity of a predicate with help from the crowd. Consider a crowdsourced workflow that filters a collection of photos of people to those of males with red hair. Crowd workers are shown pictures of people and provide either the gender or hair color of the person in the photo<sup>1</sup>. Suppose we could estimate that red hair is prevalent in only 2% of the photos, and that males constitute 50% of the photos. We could then order these predicates properly (asking about red hair first) and perform fewer HITs overall. Whereas traditional selectivity estimation saves database users time, properly identifying the filter operator in this query also saves users money by avoiding unnecessary HITs.

In addition to being used inside optimizers to estimate intermediate result size and cost, selectivity estimators can be used to estimate answers to COUNT, SUM, and AVERAGE aggregate queries with GROUP BY clauses. In a crowd context, such answers are fundamentally approximate, since humans may disagree on answers. For example, say we had a corpus of tweets and wanted to perform sentiment analysis on those tweets, identifying how many positive, negative, and neutral tweets were in our dataset. We could ask our database to provide us with a count of all tweets grouped by their crowd-identified sentiment.

---

<sup>1</sup>In the previous chapter, we saw that one can batch the same question about multiple records or batch multiple questions about the same record with similar cost effectiveness. As such, asking a worker to provide both the hair color and gender of the person in the photo would do little to reduce costs.

The simplest way to perform selectivity estimation and count-/sum-based aggregation would be to iterate over every item in the database and ask the crowd to determine if it satisfies a predicate for filters or which group it belongs to for aggregates. Unfortunately, this requires work proportional to the number of data items in the database, which could be prohibitive for large databases. A more efficient way to estimate selectivity (also used in traditional optimizers) is to sample a subset of the rows. This concept naturally translates to crowdsourcing: in our example, we can generate HITs on a subset of our photos, asking workers to label either the gender or hair color of the person in the photo. With enough samples, we could estimate the popularity of males or redheads in our dataset. This approach works, but does not take advantage of humans’ natural ability to process multiple elements—especially for heavily visual items like images—at the same time, as a batch [79]. Our hypothesis is that people can estimate the frequency of objects’ properties in a batch without having to explicitly label each item.

This observation leads to the first key contribution of this chapter: we employ an interface and estimation algorithm that takes advantage of humans’ batch processing capabilities. Instead of showing an image at a time, we can show 100 images (or 100 sentences) to a crowd worker, and ask them to tell us approximately how many people in the images are male or red-headed (or what fraction of sentences are positive / negative). By aggregating across several large batches and multiple crowd workers, we can converge on the true fraction of each property. This “wisdom of the crowds” effect, where an individual may not accurately estimate an aggregate quantity, but the average of a number of individual’s estimates do approximate the quantity, has been well documented [69]. We show that this approach allows us to estimate various aggregates across a number of image-based data sets using about an order of magnitude fewer HITs (and, correspondingly less money and latency for query results) than sampling-based approaches with comparable accuracy. We find that on textual datasets, the same effect does not apply—specifically, item-level labeling works better than asking people to estimate an aggregate property of a batch of short sentences (Tweets, in our experiment.)

The fast convergence of our count-based approach is not without challenges. Because workers are free to provide us with any estimate they wish, we must design algorithms to detect and filter out “bad” workers such as spammers who answer quickly in order to receive payment without doing work in earnest. The algorithm we designed filters out spammers in both the count- and label-based interfaces by removing workers whose answer distribution does not match other workers’. Unlike previous techniques that require redundant answers to each label from multiple workers, here we ask each worker to process different random subsets of a dataset. We also identify a solution to the problem of a coordinated attack by multiple workers, or sybil attacks [35] from a single worker with multiple identities.

In summary, in this chapter, our contributions are:

1. An interface and technique to estimate selectivity for predicates and GROUP BY expressions over categorical data. Specifically, for a dataset (e.g., a collection of images) the algorithm approximates the distribution of some property (e.g., gender) in the dataset. This has applications to query optimization in

selectivity estimation, and to SELECT-COUNT-GROUP BY queries. Our approach converges on a correct response with high confidence up to an order of magnitude faster than traditional sampling techniques.

2. A method for identifying low-quality or spam responses to our batched interface. Prior work estimates worker quality by asking multiple crowd workers to label the same data. We instead have workers provide us with non-overlapping estimates of the number of items with a property in a given batch. Workers whose answers are consistently near the global worker mean are judged to provide quality work, while workers who consistently stray from the mean are judged to be low-quality. This approach improves our accuracy on real estimation problems by up to two orders of magnitude.
3. A technique to identify and avoid coordinated attacks from multiple workers, or one worker with multiple identities. If multiple workers agree to provide the same estimate (e.g., “let’s report that any collection of 100 photos we see contains 80 males”), our spammer detection technique may be thrown off from the true value. To avoid this attack, we insert a random amount of verified gold standard data into each HIT, and show that this technique can weaken the strength of an attack in proportion to the amount of gold standard data used.

We show through experiments that our approaches generalize to multiple problem domains, including image estimation on human photos and text classification on tweets.

## 6.2 Motivating Examples

In this section, we provide three example use-cases that use crowd-based counting and selectivity estimation. While our examples utilize features of Qurk as a crowd-powered workflow system, these optimization opportunities are available in all such systems.

### 6.2.1 Filtering Photos

First, consider a table `photos(id, name, picture)` that contains photos of people, with references to their names and pictures. Suppose we want to filter this dataset to identify photos of redheaded males. In Qurk, we would write:

```
filtered_photos =  
  FILTER photos  
  BY gender(picture) = 'male'  
  AND hairColor(picture) = 'red'
```

In this case, `gender` and `hairColor` are crowd-powered user-defined functions (UDFs) that specify templates for HTML forms that crowd workers fill out. The `gender` UDF has the following definition in Qurk:

```

TASK gender(field) TYPE Generative:
  ItemPrompt: "<table><tr> \
               <td><img src='%s'> \
               <td>What is the gender of this person? \
             </table>", tuple[field]
  Response: Choice("Gender", ["male","female",UNKNOWN])
  BatchPrompt: "There are %d people below. \
               Please identify the gender \
               of each.", BATCHSIZE
  Combiner: MajorityVote

```

This UDF instantiates an `ItemPrompt` per tuple, with a radio button that allows the worker to select *male* or *female* for that tuple. It is common for workers to answer multiple such `ItemPrompts` in a single HIT, and in such situations, the generated HIT is preceded with a `BatchPrompt`, letting the worker know how many image labels are ahead of them. We discussed the effects and implications of batching in Chapter 5. Multiple worker responses may be required to prevent a single incorrect worker from providing a wrong answer, and the `MajorityVote` combiner takes the most popular worker response per tuple.

As we describe below, a good crowd-powered optimizer will identify `gender(picture) = 'male'` as filtering out less records than `hairColor(picture) = 'red'` and first filter all tuples based on hair color to reduce the total number of HITs.

## 6.2.2 Counting Image Properties

Our second example involves grouping and aggregating data. Imagine a table `shapes(id, picture)` that contains several pictures of shapes that vary in fill color and shape. If one wanted to generate an interface to navigate this collection of images, it would help to summarize all colors of images and their frequency in the dataset, as in the following query:

```

counted_shapes =
  FOREACH shapes
  GENERATE fillColor(picture), COUNT(id)
  GROUP BY fillColor(picture)

```

This query would also provide a histogram of all image colors in the `shapes` table. Here, `fillColor` is a crowd-based UDF that asks a worker to specify the color of a shape from a drop-down list of possible colors. These colors would be predefined, or a crowd-based technique for listing potential colors [72, 17] could be used.

## 6.2.3 Coding Tweet Text

Our final example shows how our approaches apply to datatypes beyond images. It is common in social science applications to “code” datasets of user data. For example, in

a study of Twitter usage habits by André et al. [15], the authors had crowd workers categorize tweets into categories such as “Question to Followers,” or “Information Sharing.” Given a table of tweet content such as `tweets(authorid, time, text)`, one might have crowd workers code those tweets in the following way:

```
counted_tweets =  
  FOREACH tweets  
  GENERATE category(text), COUNT(authorid)  
  GROUP BY category(text)
```

Here `category` is a crowd-based UDF that presents workers with a form asking them to code each tweet. Having a fast way to provide a histogram over such textual data would be valuable to social scientists and other data analysts.

## 6.3 Counting Approach

Our fundamental problem comes down to estimating the count of the number of items in a dataset that satisfy a predicate or belong to a group. These counts can be used to answer aggregate queries or estimate selectivities. We explore two methods for counting: a label-based approach and a count-based approach. The label-based approach is based on traditional sampling theory. We sample tuples and ask the crowd to label the category assigned to each tuple (e.g., whether a photo is of a male or a female) until we achieve a desired confidence interval around the frequency of each category. The count-based approach displays a collection of items to a worker and asks them to approximate how many of the items fall into a particular category (e.g., the number of images with males displayed).

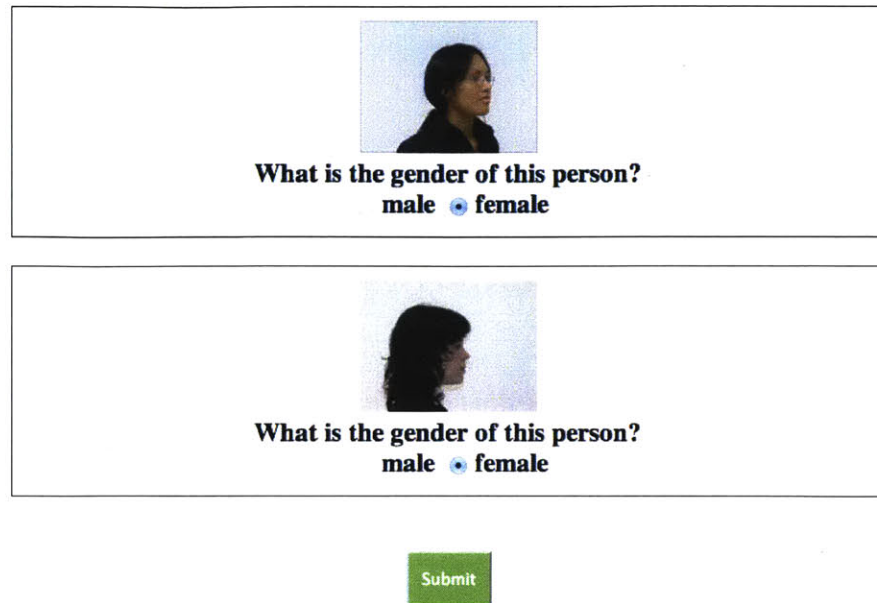
Both approaches result in estimates of the true frequency of each category that, in the absence of faulty worker responses, converge on the true frequency. Because in practice all crowdsourced worker output has some uncertainty, all crowd-powered techniques result in approximate answers. We study the convergence rate and accuracy of various approaches in Section 6.7.

For our purposes, we assume that categories are known ahead of time (e.g., we know the domain of the `GROUP BY` attributes). One interesting line of research lies in how we can determine all of the distinct categories covered by our estimation technique. Various projects explore how this can be done in a crowdsourced environment [17, 72].

### 6.3.1 User Interfaces for Count Estimation

Crowd worker user interface design, or the specification of tasks and user interface that workers see when they perform a HIT, is an important component in achieving good result quality. Asking questions in a way that does not bias workers to provide incorrect answers, and providing interfaces that make it as hard to provide an incorrect answer as it is to provide a correct one is crucial.

**There are 2 people below. Please identify the gender of each.**



The screenshot shows a web interface for a gender identification task. At the top, a prompt reads: "There are 2 people below. Please identify the gender of each." Below this, there are two separate boxes, each containing an image of a person and a question: "What is the gender of this person?". Under each question are two radio buttons labeled "male" and "female". In the first box, the "female" radio button is selected. In the second box, the "female" radio button is also selected. Below the two boxes is a green "Submit" button.

Figure 6-1: The label-based interface asks workers to label each item explicitly.

After some iterative design, we generated two interfaces that correspond to the two approaches above: a label-based interface that prompts workers to provide a label for each item displayed, and a count-based interface that shows workers a collection of items and asks them for an approximate count of items with a given property.

Label-based interfaces are common on MTurk, as image labeling is a popular use of the system. The label-based interface can be found in Figure 6-1. We see that several items, in this case images of people, can be displayed in a single HIT. A prompt at the top of the HIT asks workers to select the best label for each image. Following each image, we see two radio buttons that the worker toggles between to identify the gender of the person in the image. After going through several images (in this case, the batch size of images per HIT is 2), the worker clicks the Submit button. We error-check the submission to ensure all of the radio button pairs have a selection, and upon successful submission of the HIT, offer the worker another set of 2 images to label.

Count-based interfaces are less common on MTurk, so this interface required more design work. The interface we used for count-based experiments in this chapter can be seen in Figure 6-2. A prompt at the top of the HIT asks workers to identify how many images have a particular property. Below that general instruction, workers are prompted to enter the number of males and females in the collection of images below. To estimate the frequency of two categories such as *male* and *female*, we only need to ask about the number of items in one of those categories. We have included both questions in the screenshot to illustrate the appearance of a multi-class interface. Fixed-width images are displayed below the questions in tabular format. With more



There are 10 people below. Please provide rough estimates for how many of the people have various properties.

About how many of the 10 people are male? 4

About how many of the 10 people are female?

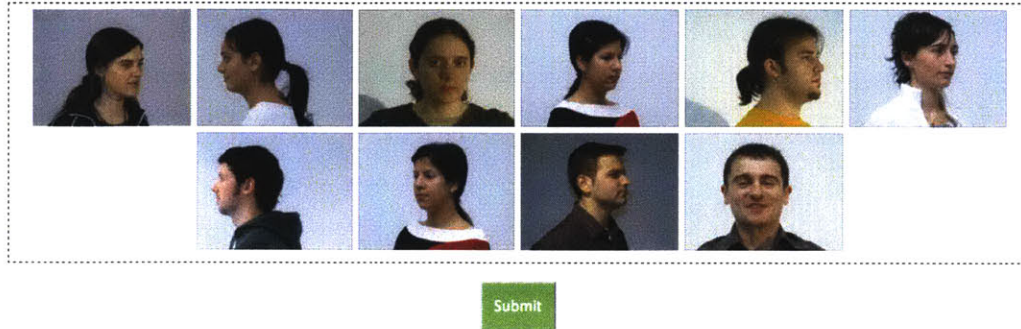


Figure 6-2: The count-based interface asks workers to estimate the number of items with a property.

images (e.g., 100) per page, the worker may have to scroll to reach the submit button. When a worker pushes submit, the interface alerts them if they have not filled in all of the requested counts, and performs bounds-checking on the inputs to ensure they are non-negative and add up to less than or equal to the total number of images displayed.

Our count-based interface design has several alternatives. One interface would resize images so that, as more images are displayed on a page, the thumbnails shrink and no scrolling is required. We avoided this design so that we could study batch size (the number of images on a page) independent of task difficulty (identifying properties of images as they shrink). Another design choice involves where to place the prompts (i.e., the questions asking how many males are displayed). We found that placing prompts at the top of the page made it easier for first-time workers to learn the task without scrolling to the bottom of pages with large batch sizes. Scrolling to the top of the page to fill in an answer after counting items did not seem to disturb veteran workers. Finally, there are several options for the wording of the prompts. We can ask workers to approximate the number of males, or prompt them for a precise measurement, and study the effect on result accuracy. We prompted workers to tell us “About how many” items have a given property, but allow them to be as precise as they desire. We leave a study of the effect of wording on performance for future work.

As we will show in Section 6.7, the count-based interface allows a batch size of about an order of magnitude more images than the label-based interface. Given the lack of prior work on count-based interfaces, it is likely that further user studies and design iterations could lead to more effective count-based interfaces. It is less likely that label-based interfaces, which are so popular and well-researched on MTurk, will see much improvement.

### 6.3.2 Modifying UDFs to Support Counting

The label-based approach to estimating the fraction of items with a property simply requires sampling the tuples in a table and applying a labeling UDF. Because we can apply the UDF directly, the user interface for the label-based approach can be derived directly from the UDF definition used in Qurk with no additional information. In the case of the count-based approach, however, new prompts are necessary to give the worker instructions on how to answer each query.

As an example, we modify the `gender` UDF from Section 6.2.1 to include prompts for counting the males and females in the dataset. Below is the original `gender` definition with two additions in bold:

```
TASK gender(field) TYPE Generative:
  ItemPrompt: "<table><tr> \
               <td><img src='%s'> \
               <td>What is the gender of this person? \
               </table>", tuple[field]
  PropertyPrompt: "About how many of the %d \
               people are %s?", BATCHSIZE, PROPERTY
  Response: Choice("Gender", ["male", "female", UNKNOWN])
  BatchPrompt: "There are %d people below. \
               Please identify the gender \
               of each.", BATCHSIZE
  CountPrompt: "There are %d people below. \
               Please provide rough estimates for how \
               many of the people have various \
               properties.", BATCHSIZE
  Combiner: MajorityVote
```

The first addition is `PropertyPrompt`, that specifies how to ask about each individual property. We also add `CountPrompt`, that specifies the overall instructions for the entire HIT. With these modifications, the Qurk optimizer can now prompt workers with the count-based interface for selectivity estimation and `SELECT-COUNT-GROUP BY` queries.

### 6.3.3 Estimating Overall Counts From Worker Counts

The “wisdom of the crowds” [69] says one can get a reasonable estimate for the number of jelly beans in a jar by asking many people to estimate the number of beans in the jar and taking the average of their responses. Our problem is not quite this simple, because the population of items is large and the individual items are not compact enough to show every person all of the items at the same time. Instead, we show each worker a random sample of the items in our collection, and ask them how many items have some property. For example, we can take a sample of 100 images from a collection of several million, and ask a worker to approximate how many of the people in the images are male.

By the central limit theorem, the average of the fraction of items with a property in each sample approaches the fraction of items in the dataset with that property. There are three sources of error in this measurement:

1. Sampling error. In traditional sampling theory, we see errors in approximations derived from samples due to variations in the individual subsets that are selected.
2. Worker error. Workers provide incorrect responses due to human error, or because entering nonsensical or less-than-accurate results allows them to complete more tasks for money.
3. Dependent samples. Platforms such as MTurk allow workers to perform more than one of the available HITs, meaning that multiple responses can be received from each worker, and often spammers are the workers who complete the most tasks.

While averaging across all responses reduces the sampling error as the number of samples increases, the last two sources of error make taking an average inaccurate. Limiting each worker to a single response would reduce serious error due to spammers, but goes against the design of platforms such as MTurk, in which the majority of the work is often done by a small group of workers [40].

The majority of workers provide accurate results, and one should allow these workers the opportunity to do as much work as possible. Still, there is a danger in the spammer who sometimes provides the largest number of responses due to the speed at which they can generate inaccurate results. It is better to design algorithms that detect, discount, and block future results from these workers while letting other workers productively produce results. We explore a solution to this problem in Section 6.4.

### 6.3.4 Comparing the Approaches

The label- and count-based approaches are subject to different rates of worker error, as they provide different interfaces and different prompts to the worker in order to generate their estimates. Additionally, they incur different sampling errors because of the differing amount of items that workers are willing to process for a price in the different interfaces. We will leave worker error aside until the experiments in Section 6.7, and for now focus on how each approach results in different amounts of items sampled in a given number of HITs.

The approach to getting an item’s label (e.g., identifying a picture of a flower as being *blue*) through multiple crowd workers is well-studied [46, 49]. At a high level, we retrieve  $R$  redundant labels for each item, usually around 3–7. The redundancy allows us to run majority vote-like algorithms amongst the multiple worker responses to boost the likelihood of a correct item label. A label-based interface supports batching  $B_L$  item labels per HIT, usually in the range of 5–20. For example, to label 500 pictures as male or female, we may display 10 images per HIT, and ask 5 workers to provide labels for each image. This would require  $\frac{5 \cdot 500}{10} = 250$  HIT assignments to complete the labeling task. More generally, with a budget of  $H$  HITs, each HIT containing  $B_L$  items to be labeled, and  $R$  redundant labels per HIT, the number of items we can label ( $N_L$ ) is:

$$N_L = B_L \left\lfloor \frac{H}{R} \right\rfloor.$$

In the count-based approach, we are not labeling particular items precisely, and our spammer detection technique does not require redundant responses. We can batch  $B_C$  items per HIT, usually in the range of 50–150, since workers are performing less work per item labeled (they are counting or estimating images in chunks rather than explicitly working to label each image). In our image labeling example with 500 images of people and a batch rate of 75, we examine all images in 7 HITs, rather than the 250 it took to label images precisely. More generally, with a budget of  $H$  HITs and each HIT containing  $B_C$  items to be counted, the number of items we can count ( $N_C$ ) is:

$$N_C = B_C H.$$

Putting aside differences in worker error in each approach, we can compare how many items can be approximately counted or labeled in our approach given a budget of  $H$  HITs. Taking the ratio of items counted to items labeled and assuming  $H$  is a multiple of  $R$ , we get

$$\frac{N_C}{N_L} = \frac{B_C R}{B_L}.$$

In our running example ( $B_L = 10$ ,  $R = 5$ ,  $B_C = 75$ ), we end up with workers reviewing 37.5 times as many items counted as labeled for a given budget. As we will find experimentally, the label-based approach benefits from breadth rather than redundancy, and so in practice  $R = 1$ . Still, for certain classes of data (data with high “pop-out”),  $B_C \gg B_L$ , which allows the count-based approach to reach an answer more quickly than the label-based approach.

### 6.3.5 Estimating Confidence Intervals

In both the label- and count-based approach, a user wants to know that the estimate provided to them is within a certain margin of error from the true distribution of item properties. For example, in selectivity estimation, a user might be comfortable with a relatively wide margin of error between two filters, say 10%, as small errors will not harm query processing or cost too much. When performing an aggregate query, however, the user might prefer to attain high confidence in their results, demanding the confidence of the gender split in their dataset to be near 1%.

In the label-based approach, we can consider each item’s label a Bernoulli random variable (a categorical random variable in the case of more than two property values) with some probability of each property value occurring. For example, when estimating the fraction of male photos in a collection after labeling around 100, the gender displayed in any photo is modeled as a Bernoulli variable with probability of being male  $p_{male}$ . If we have labeled 40 photos as male, we can estimate  $p_{male}$  as  $\frac{40}{100} = 0.4$ . Using sampling theory, we can either utilize the normal approximation of the

confidence interval for binomial distributions or the more accurate Wilson score [77] to determine, with high probability, how close the actual answer is to the estimate.

The count-based approach works similarly. If we display  $B_C$  items (e.g., photos) per HIT and have workers provide us with a count (e.g., the number of males), we can model this count as a binomial random variable  $C$  that is distributed as  $\text{Binomial}(B_C, p_{\text{male}})$ . As we collect counts  $C_i$  from various HITs, the average of the fractions  $\frac{C_i}{B_C}$  will approach  $p_{\text{male}}$  and we can again employ the Wilson score or normal approximation of the confidence interval.

In practice, some of the responses will come from spammers, and assuming we can detect them using techniques in Section 6.4, we can calculate a confidence interval on the remaining values. Additionally, instead of calculating confidence intervals, one might employ resampling techniques such as the bootstrap [37] to estimate these values on smaller sample sizes.

## 6.4 Identifying Spammers

We now discuss how to identify spammers using count-based input from crowd workers. Spammer identification for label-based approaches is well-studied [46, 49], and so we focus on count-based spammer identification.

Unlike the label-based approach, the count-based one is not built around redundant responses (each worker sees a random sample of the items in our dataset), and the responses are continuous/ordinal (workers provide us with sample property estimates which map to fractions in the range  $[0, 1]$ ). While the spam detection techniques we discuss below apply to our estimation problem, it is more generally applicable for any worker response for data which is sampled from a continuous distributions concentrated around some mean (e.g., counts, movie ratings, best-frame selection [20]).

A useful side-effect of our algorithm working on non-redundant worker output is that, while we designed it for count-based inputs, it also works for label-based inputs where no redundant labels are collected. If a worker labels five sampled images as male and 20 as female, we can feed these numbers into the spammer detection algorithm below to determine how likely that worker is to be a spammer. We will show in Section 6.7 that collecting redundant labels requires excessive amount of worker resources for the label-based approach, and so the applicability our spammer detection technique to non-redundant label-based data is also a benefit of the approach.

We will first define some variables and terms used in our algorithmic descriptions. With these definitions, and some insights we got while engineering an effective algorithm, we will conclude with the algorithm that we will show works well in Section 6.7.

### 6.4.1 Definitions

Say we have  $N$  workers  $T_1, \dots, T_N$ . Each worker  $T_i$  completes  $H_i$  HITs, and for each HIT provides us with counts  $C_{i1}, \dots, C_{iH_i}$  of the number of items in that HIT with a particular property. If there are  $B_C$  randomly sampled items in each HIT, we can convert  $C_{ij}$  (count from HIT  $j$  reported by worker  $i$ ) into a fraction

$$F_{ij} = \frac{C_{ij}}{B_C}.$$

We wish to calculate  $\hat{F}$ , an approximation of  $F$ , the overall fraction of items with a given property in our population. If we know there are no spammers, we could simply average the worker responses to get  $\hat{F} = \frac{\sum_{i,j} F_{ij}}{\sum_{i,j} 1}$ . However, given the presence of spammers, we would like to discount workers with bad results. We will accomplish this by identifying workers as potential spammers and assigning each worker a weight  $\theta_i \in [0, 1]$ , with 0 representing a spammer and 1 representing a worker with high quality.

### 6.4.2 A Spammer Model

We consider the case where each worker is a spammer or a good worker ( $\theta_i \in \{0, 1\}$ ). Say there is some underlying fraction  $\delta$  of workers that are good. Then we can model each  $\theta_i$  as a random bernoulli variable with probability  $\delta \in [0, 1]$ . In this case,  $\theta_i$  is 1 (a good worker) with probability  $\delta$ , and 0 (a spammer) with probability  $1 - \delta$ .

We can define a probability distribution on  $\delta$ . A reasonable distribution is one with decreasing probability of high  $\delta$ , since we usually see a small number of spammers. A negative exponential distribution accomplishes this goal:

$$\Pr[\delta = x] = \alpha(\lambda)e^{-\lambda x}$$

Where  $\alpha(\lambda) = \frac{\lambda}{1-e^{-\lambda}}$  is a normalization term to make the probability distribution function of  $\delta$  sum to 1 in the range  $\delta \in [0, 1]$ . We use a regularization term  $\lambda$  to control how biased the distribution is toward a large or small amount of spammers.

We can now characterize the worker responses  $C_{ij}$  depending on whether the worker is a spammer or not. If the worker is not a spammer, they are shown a sample of size  $B_C$  with each displayed item having probability  $F$  of having some property. Their response  $C_{ij}$  (the number of items with some property) is distributed as a binomial random variable on  $B_C$  items with probability  $F$ . If they are a spammer, their response is a random number with no relationship to  $F$ . Thus, the fractions  $F_{ij}$  reported by each worker are defined as:

$$F_{ij} = \begin{cases} \frac{\text{Binomial}(B_C, F)}{B_C}, & \text{if } T_i \text{ is a good worker} \\ \text{independent of } F, & \text{if } T_i \text{ is a spammer} \end{cases}$$

Under this probabilistic model, the likelihood is given by

$$\Pr[\delta, \{F_{ij}\}, \{\theta_i\}] = \alpha(\lambda)e^{-\lambda\delta} \prod_i \left\{ \delta^{\theta_i} (1-\delta)^{(1-\theta_i)} \prod_j \left( \binom{B_C}{\lfloor B_C F_{ij} \rfloor} F^{B_C F_{ij}} (1-F)^{B_C(1-F_{ij})} \right)^{\theta_i} \right\}$$

### 6.4.3 An Ineffective Algorithm

To maximize the likelihood function provided in the previous section, we use a version of alternating minimization methods popular in machine learning and computer vision applications.

1. Initialize  $\theta_i$  to 0 or 1 randomly (with probability .5).
2. Given  $\hat{\theta}_i$ 's calculate  $\hat{F}$ :

$$\hat{F} = \frac{\sum_{i,j} \hat{\theta}_i F_{ij}}{\sum_{i,j} \hat{\theta}_i}$$

3. Given  $\hat{\theta}_i$ 's calculate  $\hat{\delta}$ :

$$\hat{\delta} = \frac{\lambda + N - \sqrt{(\lambda + m)^2 - 4\lambda \sum_i (1 - \hat{\theta}_i)}}{2\lambda}$$

4. Given  $\hat{F}$ ,  $\hat{\delta}$  compute  $\hat{\theta}_i$ 's:

$$\hat{\theta}_i = \begin{cases} 0, & \text{if } \log(1 - \hat{\delta}) - \log(\hat{\delta}) - \sum_j L(\hat{F}; F_{ij}) < 0 \\ 1, & \text{otherwise} \end{cases}$$

Where

$$L(\hat{F}; F_{ij}) = F_{ij} \log(F_{ij}) + (1 - F_{ij}) \log(1 - F_{ij}) - F_{ij} \log(\hat{F}) - (1 - F_{ij}) \log(1 - \hat{F}).$$

5. Iterate back to step 2 until convergence of  $\hat{F}$  (usually within 20 iterations).

Briefly, this algorithm iterates between approximating the fraction of workers that are spammers and identifying which of the workers are spammers. With a notion of the workers that are spammers, it updates  $\hat{F}$ , the estimate of the fraction of items with a given property.

In practice, this algorithm does not provide good results. There are two key issues with the algorithm as derived:

1. The algorithm estimates  $\hat{F}$  is based on the average of all responses, weighted to discount spammers' contributions. Our calculation of  $\hat{F}$  is theoretically optimal in that it is a minimum-variance unbiased estimator of  $F$ . It fails in that it allows workers with more responses to bias results in their favor. When spammers attack, they tend to answer more HITs than good workers because they can spend less work on each response. Increasing the number of HITs they complete allows spammers to pull  $\hat{F}$  toward their responses, and reduces our ability to discern spammers from good workers in step 4.

2. The algorithm is sensitive to  $\lambda$  and, by proxy,  $\delta$ . Changing  $\lambda$  changes the fraction of spammers step 3 calculates, and identifying a stable  $\lambda$  value that works across experiments is difficult, because the fraction of spammers varies by experiment. Even if we discard  $\lambda$  and set  $\delta$  manually in step 3, we see erratic changes in  $\theta_i$  values as we change  $\delta$ .

#### 6.4.4 A Spammer Detection Algorithm That Works

Given our experience with a principled model and the algorithm we derived from it, we engineered a more effective algorithm. The algorithm limit workers' contributions to  $\hat{F}$  by averaging each workers' responses. It also does away with approximating the fraction of good workers  $\delta$  by calculating a worker's weight  $\theta_i$  based on their distance from the estimate  $\hat{F}$  regardless of the other workers' responses. Our approach relies on the assumption that the majority of workers are not spammers, and that most workers will be able to reasonably estimate the correct answer to HITs. This means a spammer is a worker whose answer deviates substantially from the other workers' distribution of answers, and in particular from the mean of all of their responses. We propose an iterative algorithm that estimates the mean, uses that estimated mean to identify spammers, and then repeats, re-estimating the mean and finding spammers until a fixed point is reached.

Because spammers often answer many questions (often they are trying to maximize profit by doing many HITs cheaply), we limit the contribution of an individual worker to the overall estimate by first averaging that worker's estimate across all of the HITs they have submitted:

$$F_i = \frac{\sum_j F_{ij}}{H_i}$$

We might be able to use other summaries of the workers' input rather than an average of their responses. For example, we could use a worker's first response or pick a random response, but in practice we have found these approaches to be more fickle when a worker makes a single mistake that is not representative of their overall contribution.

We can now compute our first estimate of  $\hat{F}$ :

$$\hat{F}_{\text{initial}} = \frac{\sum_i F_i}{N}.$$

Here  $\hat{F}$  is our current estimate, where each workers' contributions are limited by averaging over all of their answers.

We then compute  $\theta_i$ , the weight of each worker, by computing the bias of their average response  $F_i$  relative to the current global estimate  $\hat{F}$ . We also threshold, disallowing workers who are too far from the mean estimate. More precisely, in each iteration:



$$\theta_i = \begin{cases} 1 - |F_i - \hat{F}|, & \text{if } |F_i - \hat{F}| < \lambda \\ 0, & \text{otherwise} \end{cases}$$

For  $\theta_i$  values to be accurate, we must calculate the global mean estimate  $\hat{F}$  iteratively after every recalculation of  $\theta$  values, as above, weighting by the values in  $\theta$ .

$$\hat{F} = \frac{\sum_i \theta_i \hat{F}_i}{\sum_i \theta_i}.$$

In our experiments in Section 6.7, we show that setting  $\lambda = 0.14$  (approximately two standard deviations from the mean) tends to remove outlier workers while still including workers with moderate bias. This technique can be thought of as an iterative outlier detection method: workers whose performance is outside of a 95% confidence interval (two standard deviations) are removed.

The algorithm iterates between estimating  $\hat{F}$  and recalculating  $\theta_i$ 's until convergence. Once we have identified spammers, we no longer need to protect against a disproportionate number of responses from a single spammer, so we calculate our final  $\hat{F}$  based on individual worker responses:

$$\hat{F}_{\text{final}} = \frac{\sum_{i,j} \theta_i F_{ij}}{\sum_{i,j} \theta_i}.$$

With this working algorithm in mind, we can now explain how each of the two weaknesses in the last algorithm are addressed. First, we average the averages of worker responses while we learn who the spammers are, reducing spammer bias. Once our iteration converges, we calculate the final  $\hat{F}$  value based on a weighted average of all worker responses, benefitting from the use of a theoretically better estimator. Second, this algorithm does not suffer from sensitivity to a global estimate of the number of spammers ( $1 - \delta$  in the previous algorithm), because each worker is considered without regard to the others in deciding if they are a spammer or not. The threshold of  $\lambda$  is the same for each worker, and is robust to small changes in  $\lambda$ .

## 6.5 Protecting Against Coordinated Attacks

In the previous section, we presented an algorithm that estimates the fraction of items with a given property ( $F$ ) by preventing any one worker from skewing the estimate  $\hat{F}$  toward their own by quickly responding to many HITs. This is because we calculate  $\hat{F}$  as the average of each worker's average response, rather than the average of all worker responses. Each worker only contributes a single value toward the average regardless of how many HITs they performed.

The algorithm described in Section 6.4 is still vulnerable to a coordinated attack from multiple workers, however. While we have not seen a coordinated attack in our experiments, there are reports of sophisticated Turkers who deploy multiple accounts

to subvert worker quality detection algorithms [48]. If several workers, or a single worker with multiple automated accounts, all submit estimates  $F_{ij}$  such that several values of  $F_i$  are similar to each other but far from the true mean, they will be able to skew their worker quality estimates in their own favor so that they are not blacklisted, or receive bonuses or additional pay for doing good work.

Susceptibility to coordinated attacks is not limited to our algorithm. Any crowd-powered approach that rewards workers for being in the same range as other workers is susceptible. For example, a recent approach by Bernstein et al. [20] to identify the best frame in a video by getting crowd workers to agree on the best scenes in the video would also be vulnerable.

A simple way to avoid this is to collect “gold standard” data (items with known labels), on several items  $\{G_1, \dots, G_M\}$ . With gold standard data in hand, one can then generate a HIT over that data for workers to complete to ensure that they accurately count data for which we know the distribution before we give them additional HITs over unknown data. We will see in Section 6.7 that in a few HITs we can sample several thousand items, and so collecting a few tens or hundreds of gold standard items is not too resource-intensive.

The simple gold standard approach has several issues. First, sophisticated workers could learn to identify the page with the gold standard data and the correct answer to the estimates for that page. Second, because the distribution of the number of tasks completed by each worker is zipfian [40], many workers will only complete one or two tasks. This limitation means that the gold standard task would constitute the bulk of the contribution of many workers, which is not ideal.

We propose a different use of the gold standard data. Rather than generate an entire gold standard task, we randomly distribute the gold standard items throughout each task. To do this, we first vary the number of gold standard items with a given label in each task a worker completes. For example, if a worker is estimating the number of men and women in a task with 100 images, we will randomly place between 1 and 20 gold standard images with known labels into the 100. When a worker tells us the number of items with a property in the task they just completed (say, 25 males in 100 images), we subtract the number of gold standard images with that property (e.g., 13 gold standard male images) from their count when calculating the fraction  $F_{ij}$ <sup>2</sup>.

Using this approach, there is no one repeating gold standard HIT, as some gold standard data is mixed into every HIT. As a result, attackers can not identify and correctly solve the gold standard HIT. Additionally, we can continue to measure worker quality throughout the worker’s contributions.

### 6.5.1 Correcting Worker Responses

Formally, we show a worker  $R$  items at a time, where  $G_{ij}$  gold standard items are introduced into the  $j$ th HIT shown to worker  $i$ . When a worker tells us there are  $C_{ij}$

---

<sup>2</sup>Note that we cannot just check how the worker did on the gold standard items, because we are not asking the worker to label individual items, but to estimate the overall frequency of some property on a number of items.

items with a given property, we subtract the  $G_{ij}$  known items when calculating the fraction:

$$F_{ij} = \frac{C_{ij} - G_{ij}}{R - G_{ij}}.$$

In this approach, since each worker sees a different fraction of gold standard data, two or more workers who are colluding to skew  $F$  to some value should no longer agree once the gold standard data is removed. If these attackers coordinate in repeating the same value  $C_{ij}$  for each task, the estimate  $F_{ij}$  will still vary per task due to the randomness in  $G_{ij}$ . The larger the variance of  $G_{ij}$  is per hit, the larger the variance will be in the corrected  $F_{ij}$ . This variance will reduce the power of disruption of a coordinated attack, and increase the probability that attackers will be removed as outliers because their answer varies from the overall mean.

In contrast, non-colluding workers who estimate the true value of  $F$  should still agree once the gold standard data is removed.

A second benefit of our approach comes in cases where the counts of items with a given property that a worker reports are smaller than the number of gold standard items with that property in a HIT ( $C_{ij} < G_{ij}$ ). In these cases, a worker's  $F_{ij}$  estimate will be negative, which is not possible. Aside from boundary situations where some worker error results in negative  $F_{ij}$  estimates that are close to 0, we can discard such workers as spammers.

This benefit is value-dependent, since attackers who submit lower count values are more likely to be identified by the approach than attackers who submit high counts. However, there is symmetry to the value dependence: assuming we can determine early on which property the attackers will say is least frequent, we can increase our likelihood of identifying attackers by providing a higher proportion of gold standard examples of that property. This benefit works against individual attackers and coordinated attacks by outright removing some workers with negative fraction estimates.

Our approach is not without drawbacks. The most direct is that, while utilizing more gold standard on average can increase the number of identified spammers and spread out the coordinated responses of attackers, it also decreases the amount of new information we learn with each HIT. If on average we use  $G$  gold standard items per HIT, and have  $R$  items per hit total, then to estimate selectivity using a sample of  $S$  items total, we require

$$\left\lceil \frac{S}{R - G} \right\rceil$$

HITs to process this data. As  $G$  grows, the number of required HITs to have workers provide information on  $S$  items grows. Additionally, collecting gold standard data takes time (and possibly money), which in practice limits the maximum value of  $G$ .

We explore the costs and benefits of this approach in Section 6.7.3.

### 6.5.2 Random Gold Standard Selection

For our gold standard technique to be effective, we need to introduce a different fraction of gold standard data into each HIT, while keeping the average amount of gold standard data used per HIT at some level. This allows users to control how much of their budget is spent on gold standard data.

Given a user-specified average number of gold standard items  $\mu$ , we intersperse  $G_{ij} = \mu + \mathcal{G}$  gold standard items into each task, where  $\mathcal{G}$  is a random integer chosen uniformly from the range  $[-g \dots g]$ . We require that the user pick  $\mu > g$  and  $R - \mu > g$  to prevent  $G_{ij}$  from falling below 0 or exceeding  $R$ . In our experiments we use  $g = .2R$ .

## 6.6 Further Exploring the Design Space

In our experiments, we will explore the various interfaces and algorithms we have outlined. In order to fully understand where in the design space our experiments lie, it helps to identify other representative solutions that we have not explored but could make for interesting directions for future work.

**Interface details.** The label-based interface is relatively common in crowdsourced workflows, as it is already used for filtering and labeling tasks. The count-based interface, however, is novel to our application, and there are several variations on it that we can explore.

Our current interface design asks workers to tell us “About how many of the [N] [item type] displayed [have some property]?” One could explore various wordings of this prompt, ranging from less precise questions to ones that suggest an exact answer is required. In our initial design iterations, we did not find a notable effect from the wording of this prompt.

The mode by which workers report a count, be it as an exact count, a rough count, or a percentage, could also affect accuracy and worker mental load. In our initial brainstorming and prototypes, we were told by test subjects that estimating the percentage of a dataset that has some property would be difficult, as it is difficult for them to think in percentages. We never explored this input mechanism or others on systems like MTurk, however, and it would be interesting to study these approaches.

**A count-label hybrid.** Whereas the label-based interface requires workers to click on a radio button to label each item they look at, the count-based interface allows workers to scan a densely packed collection of items in a grid. For features that readily pop out at the worker, the count-based interface requires less input, and can thus be completed faster. We could generate a label-based version of this interface that allows a dense presentation of this information while still getting per-item feedback. One version of this design would lay items out on a grid, exactly as they are displayed in the count-based interface. Users would be instructed to click on all items with a given property, allowing them to scan the items and only provide input on items with that property. It would be interesting to study the accuracy and speed with which workers can provide labels for items using this interface.

**Exploring the factors behind interface efficacy.** We have designed experiments

to identify when the count-based interface allows us to quickly estimate the fraction of items with a property in a dataset. We do not, however, experiment with the ways in which workers use the items presented in the interface to generate their responses. Exploring this path might allow us to design better interfaces and interactions.

One experiment would help determine if, when presented with hundreds of items at a time, workers bias their attention to only the items at the top, bottom, or other visual grouping of items. We could determine if worker attention by location varies by biasing the location in which we place various items and seeing how this affects accuracy.

Another bias in worker responses might be toward round numbers. Are workers more likely to respond to our count requests in multiples of 5 or 10? We could measure this by varying the actual count in any task and seeing how worker response histograms shift with different counts.

## 6.7 Experiments

We now quantify and compare the various methods presented in this chapter across two image datasets and a text dataset. The goal of these experiments is to understand how the convergence rates and accuracy of count-based and label-based counting/selectivity estimation compare on these different datasets. We show that the count-based approach has similar accuracy and much faster convergence than the label-based approaches on imagery, but does far worse on textual data (tweets). Finally, we simulate the effect of sybil attacks, and show scenarios in which our randomized gold standard approach averts the attacks.

### 6.7.1 Datasets

We utilize three datasets in our experiments, each of which is aligned with one of the three motivating examples in Section 6.2.

**Face Gender.** The dataset we use for most of our performance evaluation is the GTAV Face Database [71]. This dataset contains 1767 photos of 44 people. For each person, there are between 34 and 65 images of that person in different poses. We manually labeled each person in the dataset as male or female, so that we could use our estimation techniques to determine the number of images in the dataset of a given gender. Some sample images from this dataset can be seen in Figures 6-1 and 6-2.

**Shapes and Colors.** To test the difficulty of perceptual tasks, we generated an image dataset of shapes. Our generated dataset consists of triangles, circles, squares, and diamonds. The fill color of each shape is either yellow, orange, red, green, blue, or pink. Additionally, each shape has an outline that is one of these colors. For variety, shapes are sized to fit various proportions inside a 100x100 pixel box. An example can be seen in Figure 6-3. We generated 1680 such sample shapes.

**Tweet Categorization.** To test our approaches on non-image data, we also ran our approximation techniques on tweet text. Our tweets come from an initial collection of approximately 4200 from André et al., who studied the value of tweet content to

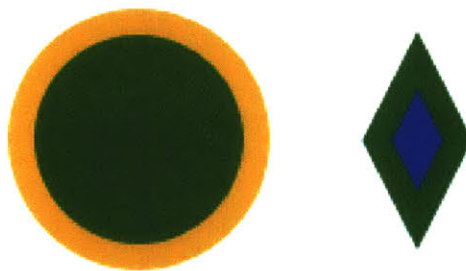


Figure 6-3: Example shapes in our Shapes and Colors dataset.

Twitter users’ followers [15]. Tweets in this study were categorized using Crowd-Flower into eight categories. The authors of that paper found several tweets to which it was hard to assign a single label, so we selected a subset of 2530 tweets in the three most stable categories: Information Sharing, where a user links to a piece of information (e.g., *Awesome article explaining the credit crisis: <http://...>*), Me Now, where a user says what they are doing now, (e.g., *Just finished cleaning the apartment!*), and Question to Followers, where a user poses a question (e.g., *What is your favorite headphone brand?*). We use our estimation techniques to determine the fraction of tweets in these three categories. “Coding” tasks, where a crowd is asked to categorize various items, are common in the social sciences, and we could imagine our techniques being useful in these situations.

For all datasets, since we have ground truth data on their labels, we can generate artificial datasets by sampling at different frequencies of labels. For example, we can generate a version of the face dataset with 10% males or 50% males by sampling more heavily from a subset of the data. For the faces and shapes dataset, all samples are of size 1000, meaning that regardless of the property distribution, we always select 1000 images on which to estimate properties. On the tweet dataset, our samples are of size 2500 tweets.

### 6.7.2 Estimating Counts

To better quantify the error of our different algorithms, we tested several estimation scenarios on MTurk, varying different parameters. For each parameter setting, we generated 1000 HITs. We tried two variations of the worker interface: label-based and count-based, as shown in Figures 6-1 and 6-2. For the count-based interface we used per-HIT batch sizes of: 5, 10, 25, 50, 75, 100, 125, and 150. For label-based we used batch sizes of: 5, 10, 15, and 20. In label-based scenarios, we collected both redundant labels (generating 200 HITs, asking 5 different workers to complete each HIT) and no-redundancy labels (generating 1000 HITs, asking 1 worker to complete each HIT)<sup>3</sup>. Redundant labels are reconciled (i.e., the crowd’s answer is determined) using Ipeirotis et al.’s approach [46] (the same technique as **QualityAdjust** in earlier

<sup>3</sup>Some item labelings are redundant across HITs in the no-redundancy case, but we do not take advantage of this to improve our item labels. We simply treat each HIT as providing another count toward our estimate.



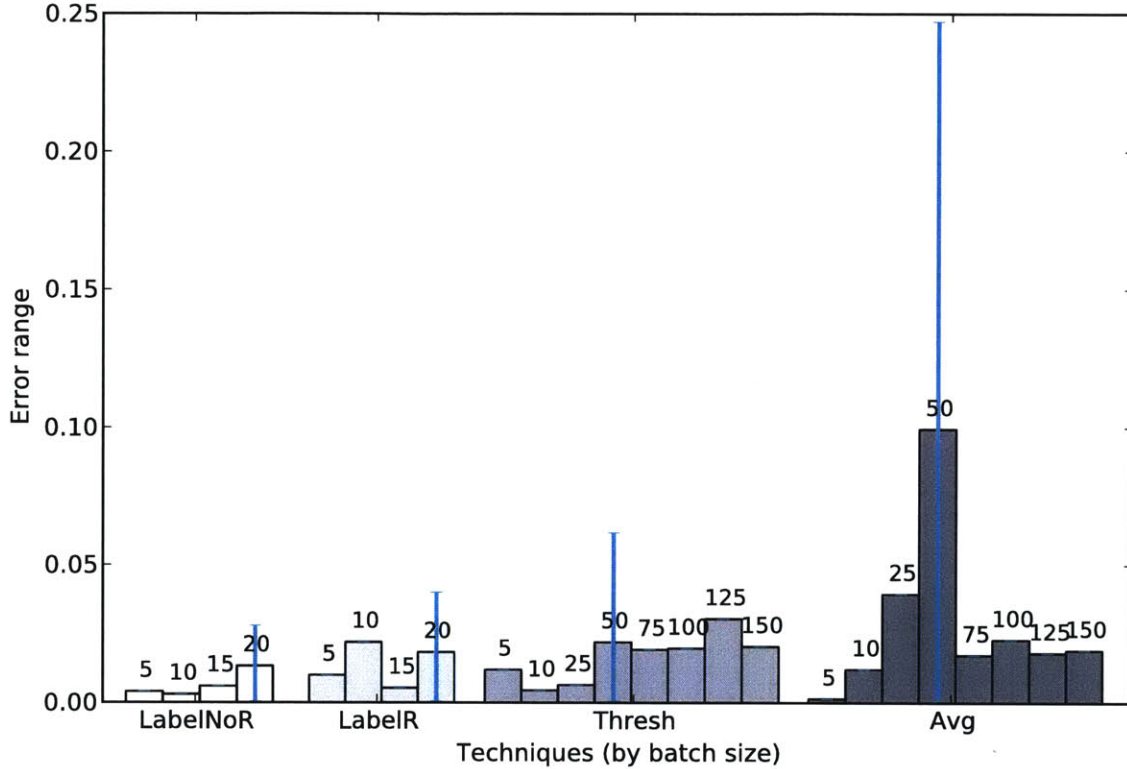


Figure 6-4: Bar chart of average error by approach, with error bars for minimum and maximum values. Numbers above each bar represent batch size. *LabelNoR* means labeling with no redundancy. *LabelR* means labeling with 5-worker redundancy. *Thresh* means count-based worker input with spammer detection. *Avg* means count-based worker input with no spammer detection. *Thresh* and *Avg* have the same inputs.

chapters). We ran tests using the Faces, Shapes, and Tweet datasets. Finally, to understand the effect of selectivity on performance, we sampled the Faces dataset such that it had 1%, 10%, 25%, 50%, 75%, 90%, and 99% male faces. To ensure repeatability, we ran each experiment at least twice during business hours in the Eastern US time zone. For all experiments, we paid workers \$0.01 per HIT without trying higher pay increments because batching more than one item per HIT was possible even at this lowest pay increment.

**Overall Error.** We start by studying the errors of various approaches on the Faces dataset. Figure 6-4 shows the error rates of the label- and count-based approaches. For the label-based approach, we report the errors with redundant labels (*LabelR*) and without (*LabelNoR*). For the count-based approach, we feed worker values into the thresholding-based spammer-detection algorithm described in Section 6.4 (*Thresh*)

and into a simple average<sup>4</sup> of all worker-reported counts (*Avg*). The bar charts displayed in this figure represent the average error across the experiments we ran, with the error bars representing the minimum and maximum error values. Error is calculated as the absolute difference between the estimated fraction and the ground-truth result of each experiment using all 1000 HITs in each experiment. We break the results down by the batch size of the various approaches.

We see that when using labeling, getting multiple labels per item does not appear to reduce the overall error. That is, we do not have evidence of heavy spamming of these interfaces, and we do have evidence that across several experiments, non-redundant labels have commensurate error rates to redundant labels. For the count-based approaches, we see that taking a simple average across all worker-provided counts generates error rates up to 25%. Feeding that same count data into the spam-detecting thresholded approach generates error rates on par with the label-based approaches. In terms of error rates on 1000 HITs, it is possible to achieve similar results using label- and count-based approaches, although the count-based approach requires spam detection. The same count-based spam detection approach could be used on the non-redundant label-based approach should spammers strike.

**Interface Latency.** While the largest batch size we tried in the count-based interface was 150, we limited label-based interfaces to batching at most 20 items per HIT. Anecdotal, even at a batch size of 20, we would sometimes run into situations where workers would take a long time to pick up our tasks because they were bordering on undesirable. More importantly, as evidenced by Figure 6-5, workers took longer to label 20 images than they did to report counts on 150. In this figure, we show different experiments along the X-axis. Along the Y-axis, we display a box-and-whiskers plot of the number of seconds to complete each HIT. For each experiment, the edges of the box represent the 25th (bottom) and 75th (top) percentile, while the red line inside each box represents the median seconds to complete a HIT. The whiskers, or ends of each dotted line, represent the minimum and maximum seconds to complete a HIT in that experiment. Maximums are often not shown, because of pathological cases where workers would pick up a HIT but not submit it until a maximum of 5 minutes was up.

The different experiments are labeled on the X-axis. Experiments labeled C150, ..., C5 are count-based with batch size 150, ..., 5. Experiments labeled L20, ..., L5 are label-based with batch size 20, ..., 5. In this section, we focus only on experiments for Face 0.1 (Faces with males representing 10% of the dataset); the other experiments are described below.

First, we see that the count-based batch 150 approach takes workers less time, in general, than the label-based batch 20 approach. Decreasing batch size generally reduces the time it takes workers to complete each task. For any equivalent batch size, the count-based approach takes less time than the label-based one. The fact that we paid \$0.01 per HIT regardless of batch size suggests MTurk is not a market with fine-granularity pricing for tasks. Because label-based batches larger than 20

---

<sup>4</sup>We tried the median and the average of the middle 5% and 10% of answers, and found their results to be just as vulnerable to spammers as the average.



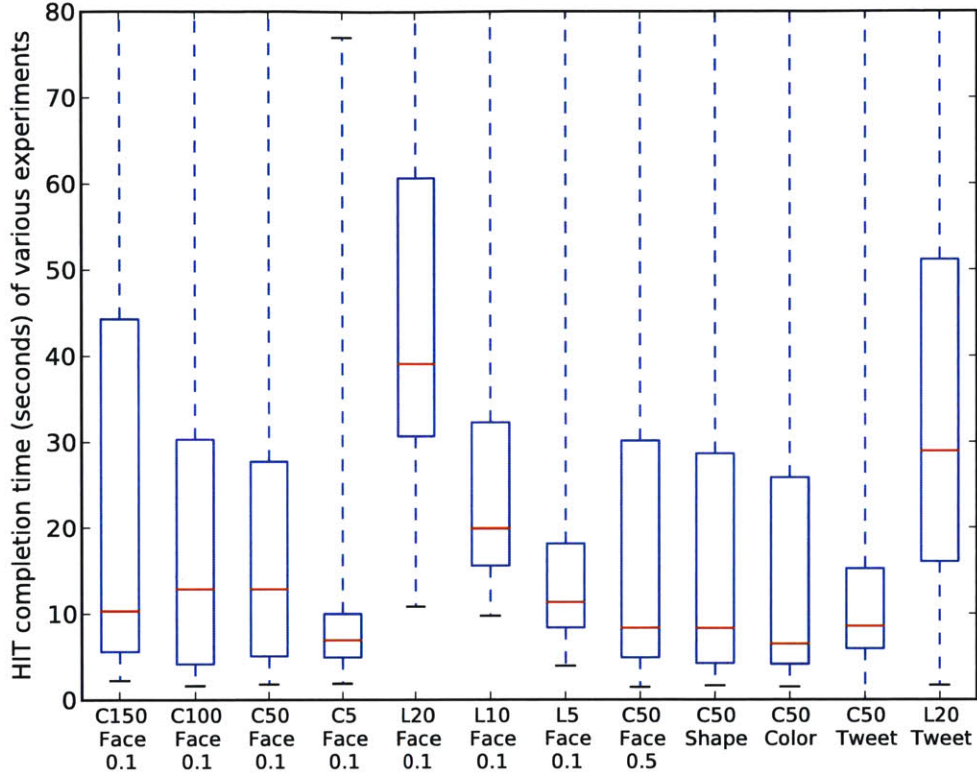


Figure 6-5: Box-and-whisker plots of the number of seconds workers took to complete different tasks. The top of each box represents the 75th percentile completion time and the bottom represents the 25th percentile. Red lines indicate medians. Whiskers (error bars) represent minimum (bottom) and maximum (top) values.

were not guaranteed to be picked up by workers in reasonable time, and because workers were spending longer on these tasks than on batch 150 count-based tasks, we decided to limit our batching at these values. The fact that counting is so much faster than labeling for images suggests that people are quickly scanning and estimating the quantities the HITs ask about, rather than manually iterating through each item and explicitly counting frequencies. This scanning behavior is the desired effect of our count-based interface design.

Another factor to consider is the end-to-end query time, or how long it takes to complete these HITs from start to finish. In our experiments, completing 1000 HITs in the count-based interface with batch size 150 takes about as long as 1000 HITs in the label-based interface with batch size 20 (30-40 minutes). For smaller batch sizes, 1000 HITs of counting 25 takes about as long as labeling 5 (15 minutes). In our experience, these end-to-end times go down as an employer gets a reputation for reliably paying for work and for providing many similar work units.

**Spam Detection Example.** In Figure 6-6, we show an example where our spam

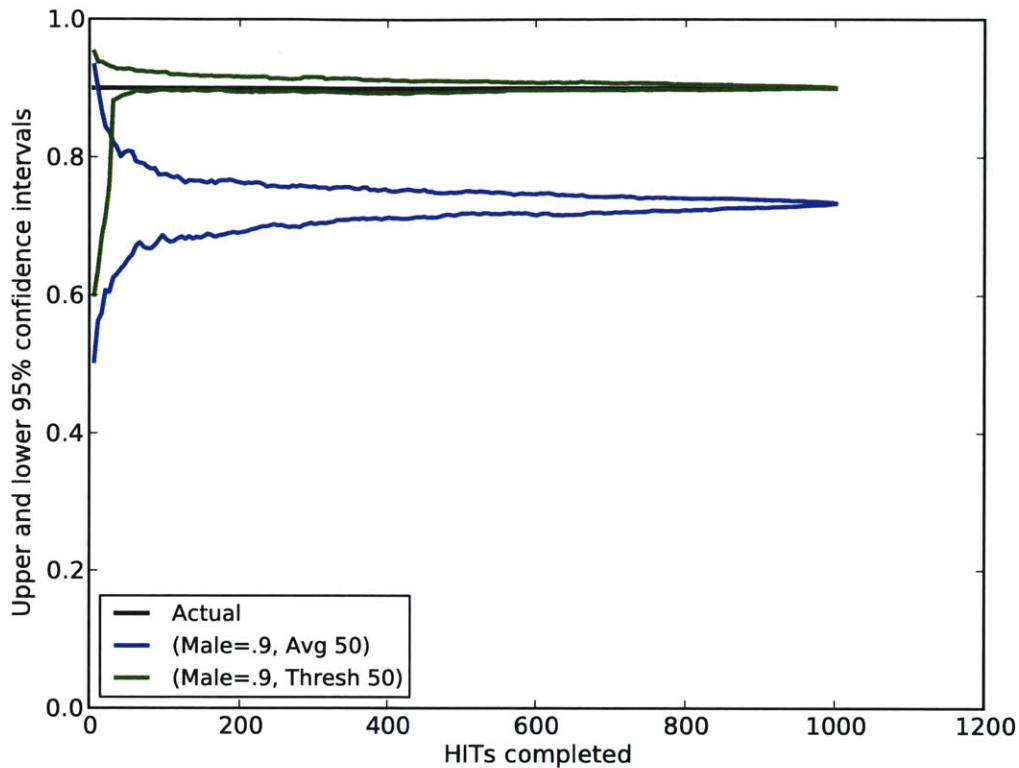


Figure 6-6: As we vary the number of HITs sampled from 1000 (X-axis), we measure the empirical 95% confidence intervals (Y-axis) of two algorithms on samples of that size. Given the same input data with a lot of spammer-provided values, the spammer-detection algorithm (*Thresh*) arrives at the correct estimate while a simple average of inputs (*Avg*) does not.

detection algorithm reduces error. Here, the two top workers finished 349 and 203 HITs of the 1000 HITs with error rates of about 46% and 26% respectively. In this figure we see that even after 1000 HITs, simply averaging across worker responses, more than half of which are spam, results in an inaccurate response. Applying our threshold-based spam detection technique allows us to arrive at the actual fraction of items with high accuracy. This happens because the algorithm is able to identify both of the high-error, high-output workers, and multiplies their contribution to the overall average by zero.

The chart also shows the convergence rate of the two approaches. Along the X-axis, we vary the number of HITs we sample from the 1000. For each X-axis value, we take 100 subsamples of that size from the 1000 HITs, and run *Avg* and *Thresh* on each of those subsamples. We plot the value of the 2nd smallest and 3rd largest estimates based on these subsamples, providing a bootstrapped (sampled) measurement of the 95% confidence interval of the approach at each HIT size. Note that the gap between

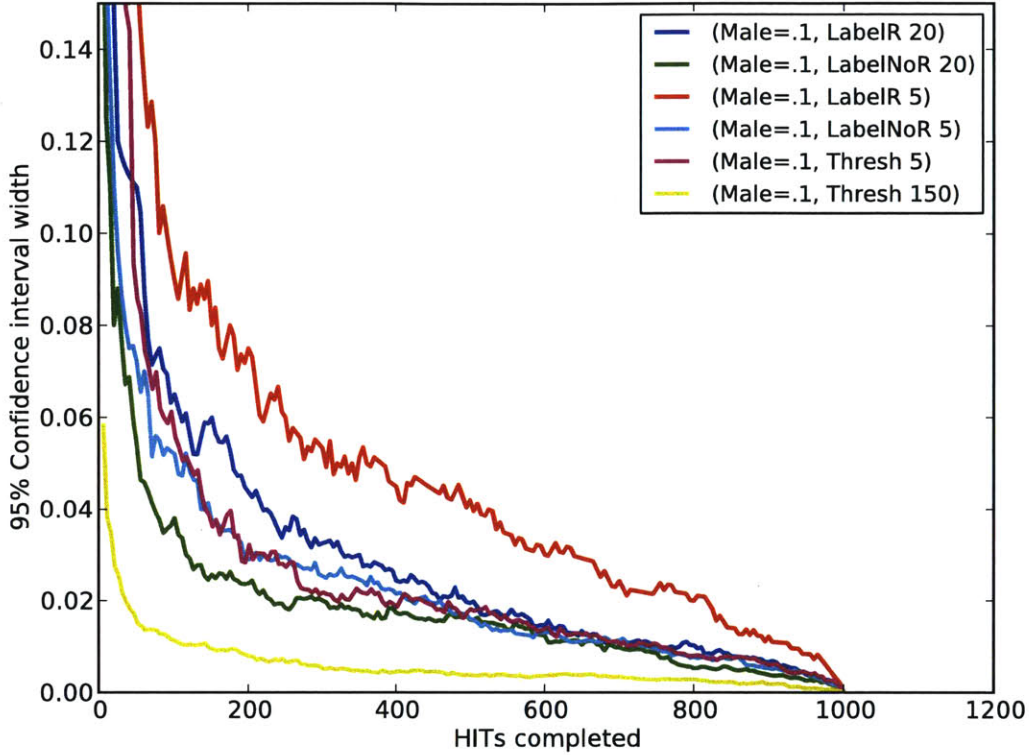


Figure 6-7: As we vary approaches and batch sizes, what 95% confidence interval width (Y-axis) do we achieve in a given number of HITs (X-axis)?

the lower and upper lines is smaller in the case of our threshold-based approach. From this, we can conclude that in addition to getting an accurate estimate with a large amount of HITs, we also converge on the correct value with a small number of them.

By using *Thresh*, the error in this example is reduced from 16.64% to 0.06% (a factor of about 265). The largest absolute improvement of the spam detection algorithm in our dataset brings an error from 26.74% to 1.77% (a 24.97% improvement).

**Convergence Rates.** As evidenced by the lack of a pattern in error rates amongst batch sizes for each technique in Figure 6-4, we did not find that batch size had a significant impact on the accuracy of our estimates at the limit (after 1000 HITs). As long as workers were willing to take on a count-based task, the error across batch sizes did not vary. When error rates did vary drastically, those errors are attributable to spammers, which are detected by algorithms such as *Thresh*.

Since the label- and (spammer-eliminated) count-based approaches achieve similar error rates at the limit, what separates them is how quickly they converge on the true value. We explore the convergence rate of various approaches in Figure 6-7. In this chart, we measure the rate of convergence of various approaches when the fraction of males is 0.1. As we vary the sample size of the 1000 total HITs along the X-axis, we



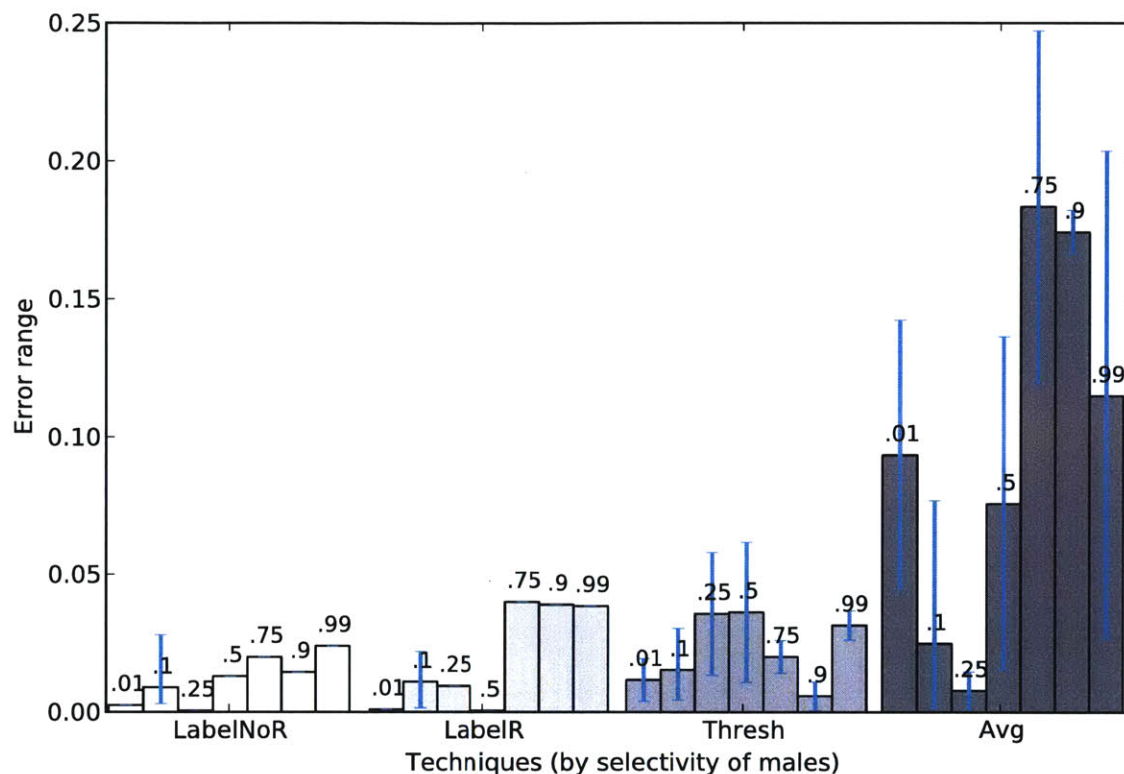


Figure 6-8: Bar chart of average error per approach, broken down by selectivity of males in the dataset. See Figure 6-4 for a description of the graph elements.

measure the 95% confidence interval width (e.g., the gap between the top and bottom lines in Figure 6-6), using the same subsampling method as the previous experiment.

We see that the fastest convergence rate is achieved with a count-based interface after spammer elimination with a batch size of 150 (the yellow line closest to the bottom left). The next fastest convergence rate is achieved by the label-based approach with no redundancy and a batch size of 20, followed by the batch size 5 count-based and non-redundant label-based approaches. The two approaches with the slowest rates of convergence are the batch size 20 and batch size 5 redundant label-based approaches. Essentially, because they provide no additional protection against spammers, the 5 redundant HITs for each label derive value from only 1 of every 5 HITs. Most importantly, we see that to be within 0.05 of the correct answer, the batch 150 count-based approach requires about 5 HITs, whereas the next fastest approach takes about 50. Similarly, we reach a confidence interval of .01 in less than 100 HITs with count-based batches of size 150, whereas the next fastest approach takes almost 600 HITs.

**Varying Frequency of Males.** In Figure 6-8, we show the error rates of the different approaches as selectivity changes. While there is no clear trend for the label-based

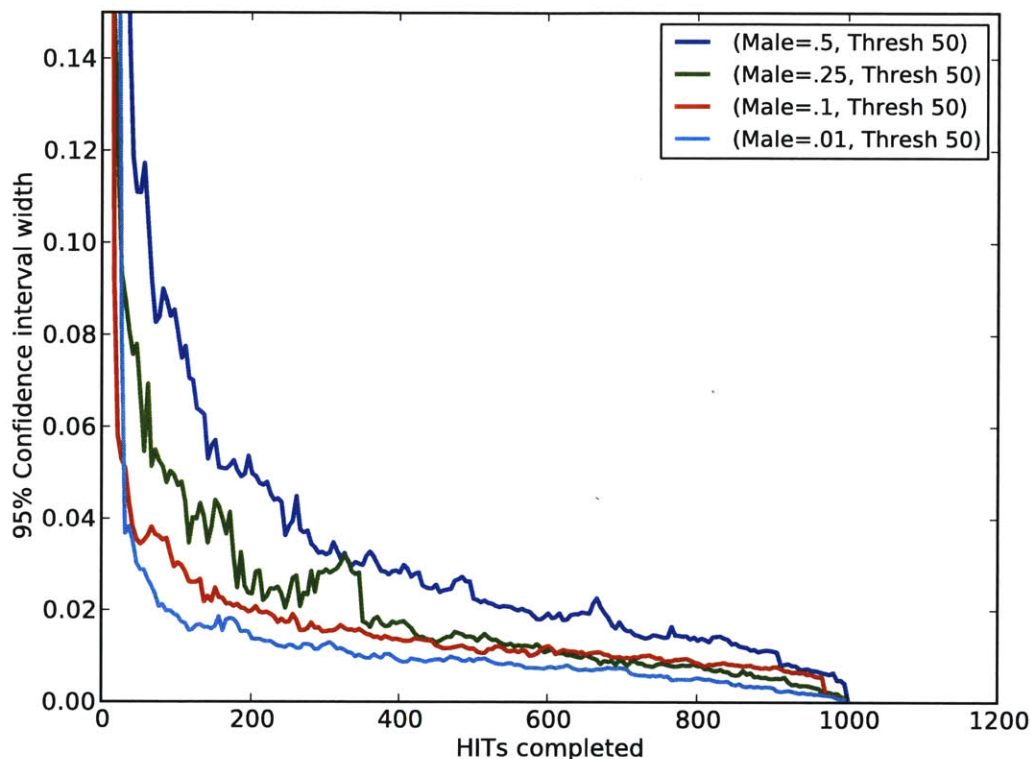


Figure 6-9: As we vary the selectivity of males, what 95% confidence interval width (Y-axis) do we achieve in a given number of HITs (X-axis)?

interfaces, the spam-detected count-based technique (*Thresh*) sees a slight trend toward higher errors as selectivities approach 0.5. We explore further by studying the convergence rate and interface latency as selectivity changes.

In Figure 6-9, we show the convergence rate of the 95% confidence interval when we vary the frequency of males in the Face data set. The results are symmetric for selectivity .75, .9, and .99, and we omit them to make the graph clearer. We see slower convergence of the confidence interval as the selectivity approaches 0.5. This suggests that there are less errors in estimating the fraction of items with very frequent or very infrequent occurrences. As the number of males approaches 50% of the data, workers have to perform more work to spot subtle differences in distribution. To understand this intuitively, imagine having to find a photo of a male amongst a sea of female photos (a relatively easy task), versus having to find all of the male photos in a sea of 50% male/50% female photos (a task that is difficult to perform with 100% accuracy). If the distribution is instead lopsided toward males or females, a quick glance can identify the outliers (less frequent) items in the dataset.

A second observation backs this hypothesis. In Figure 6-5, we see that the time workers spend in the count batch 50-based interface for male selectivities of 0.1 and

0.5 are about equal. If workers spend equivalent amounts of time on tasks that are easier to spot-check (e.g., identify 5 males of 50 pictures) than more nuanced tasks (e.g., are there 20 or 25 males out of 50?), the higher-selectivity tasks will see less accurate results.

A natural question arises: if the confidence interval becomes wider at higher frequencies, is there still a benefit to the count-based approach over the label-based one? The result is a rough draw at the highest level of batching for both techniques.

**Generalizing Our Results.** To see how well the results on the Face dataset generalize, we ran similar ones on the Shape and Tweet datasets. We were able to achieve similar conclusions on the Shape dataset, with very similar error and convergence rates, but our findings on the Tweet dataset revealed important differences.

On the Shape dataset, we tried two variations. Using a batch size of 50 for counting and 20 for labeling, we had workers approximate the border color and shape of two sampled distributions. For shape frequency estimation, we generated a dataset with 10% triangles, 30% circles, 30% squares, and 30% diamonds. For shape outline estimation, we generated a dataset with shape outline colors of 10% yellow, 30% orange, 30% red, and 30% green. These tasks are of different difficulty: given that the primary color of each shape is its fill color, identifying outline color is harder than determining its shape [79]. Still, we found similar result and accuracy trends to the ones on the Face dataset. Additionally, as evidenced in Figure 6-5 (see C50 Shape and C50 Color), workers spent about as much time on these tasks as they did on batch 50 counts of faces at different selectivities. This further supports our hypothesis that across selectivities and visual task difficulties, workers tend to spend a constant amount of time per task, varying only with batch size. Since all tasks were priced at \$0.01 per HIT, we can not determine if workers would have worked for a longer time in exchange for more money in these scenarios.

Results from the Tweet dataset were more surprising. Workers were asked to label and count tweets into one of the three categories described in Section 6.7.1. This task is harder than other estimation tasks described so far, since each tweet has to be read and analyzed, without any visual “pop-out” [79] effects.

Even though the task was harder, in Figure 6-5, we see that workers spent less time on the Tweet counting (*C50 Tweet*) and labeling (*L20 Tweet*) tasks than they did on the image tasks with equivalent batch sizes. In particular, workers spent significantly less time on the count-based tasks than the equivalently sized count-based image counting tasks.

The error rates on the tweet datasets were also interesting. The label-based approach saw larger error rates than label-based image counting techniques, but still provided usable outcomes. The count-based approaches, on the other hand, saw error rates of up to 50%, signaling that workers were not accurately providing count estimates on tweets in our interface. These two error rate results are consistent with our timing results: as workers spent less time on a more difficult task, their error rates made the count-based interface less dependable.

Given our ability to reproduce the low error rates and fast convergence rates of the count-based interface on another image dataset, and the poor performance of

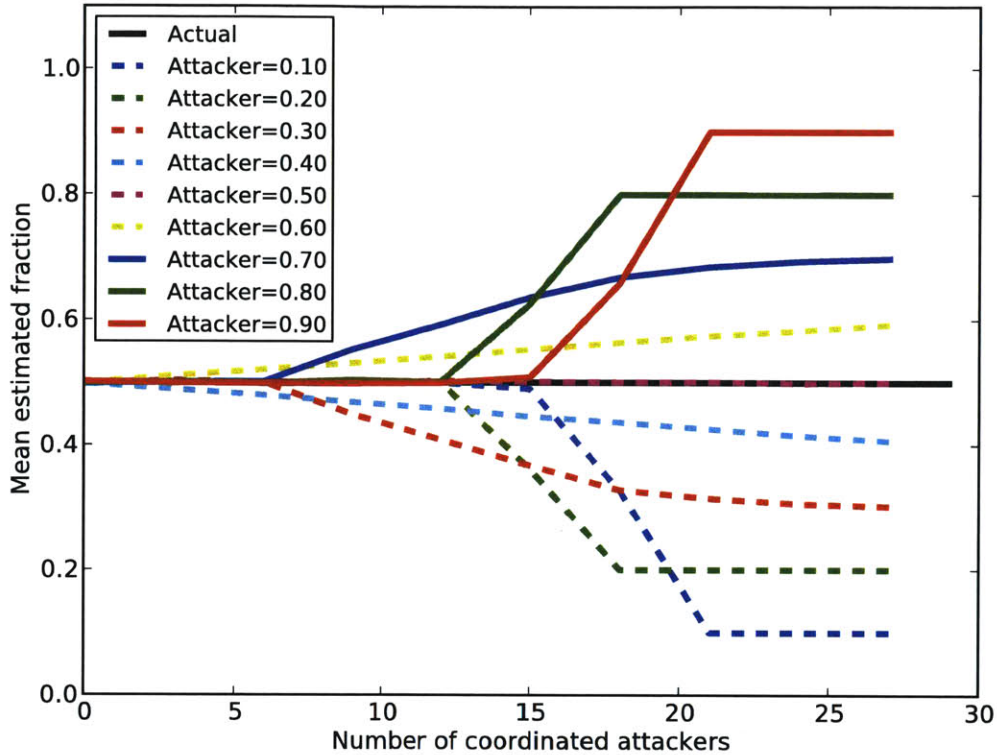


Figure 6-10: As we increase the number of coordinated attackers (X-axis), the spam detection algorithm eventually reports the attackers' values (Y-axis).

the count-based approach on the tweet dataset, we offer a hypothesis and guidance for future usage. Text classification, which has no skimmable affordances, can not accurately be counted using the count-based interface. The label-based interface, which calls for a classification of each individual tweet or body of text, is able to draw a worker's attention to each item and achieve better accuracy.

### 6.7.3 Sybil Attacks

Coordinated attacks by a single worker with multiple identities (so-called "sybil attacks") or multiple workers attacking in concert make our count-based approach less able to catch spammers, since such coordinated attacks can skew our estimate of the true mean. In Section 6.5 we described a technique for placing random amounts of gold standard data into each HIT to filter out such attackers, and we now quantify the benefit of this approach.

Sybil attacks are not yet common on platforms like MTurk, but there are reports of sophisticated Turkers who deploy multiple accounts to subvert worker quality detection algorithms [48]. Because (to the best of our knowledge) we did not experience



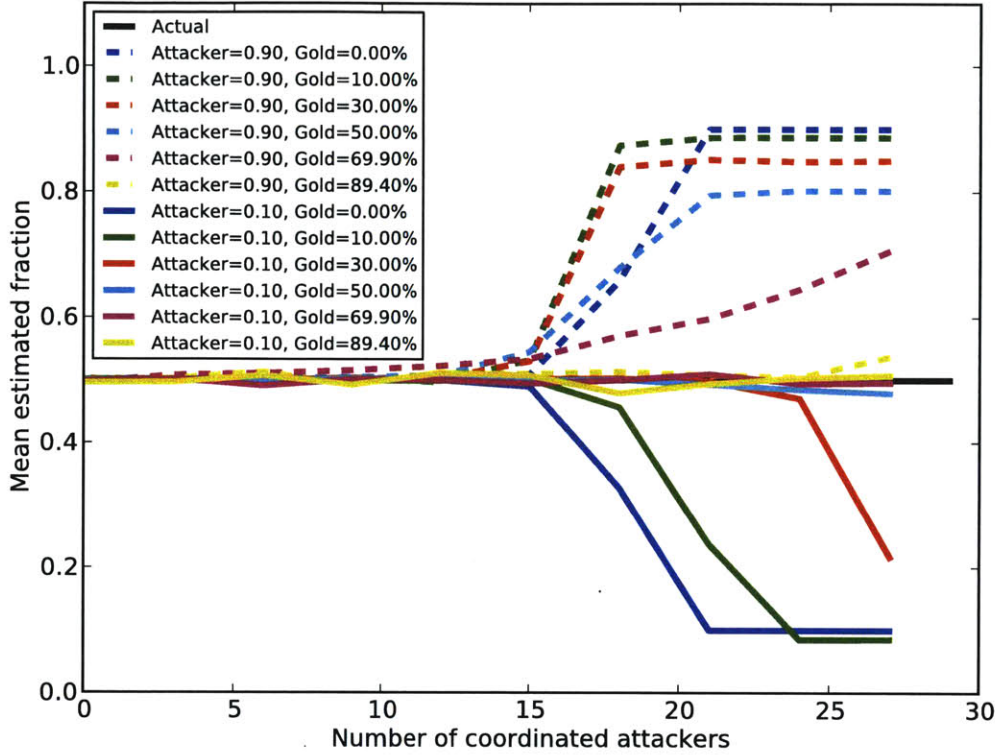


Figure 6-11: How do various amounts of gold standard data reduce the effectiveness of a coordinated attack?

any sybil attacks in our actual experiments on MTurk, we ran a series of simulations. We simulate a crowd of 30 workers estimating the fraction of items with a given property (actual fraction= 0.5). Figure 6-10 shows the effect of this coordinated attack as we increase the number of attackers on the X-axis. The attackers do not count the items shown to them, instead agreeing to report the same fraction. The fraction they coordinate on (from 0.1 to 0.9) is shown on different lines. We use the same spam-detection technique employed in the previous experiments and described in Section 6.4, without our gold-standard-based provisions to detect sybil attacks.

We see that as the number of coordinated attackers increases, the results become less and less accurate, eventually converging on the fraction that the attackers coordinate on. Larger attack fractions are farther from the overall mean when there are still workers accurately performing their task, and so our spammer elimination technique filters them out for longer. With a large enough fraction of the attackers, however, the algorithm is always subverted.

In Figure 6-11, we see what happens when we add random amounts of gold standard data to each HIT, depicted in two scenarios. The dotted lines represent coordinated attackers providing response fractions equal to 0.9 pitted against HITs with



various amounts of gold standard data ranging from 0% to 90% of each HIT on average. The solid lines represent coordinated attacks from attackers who provide a false value of 0.1, again pitted against HITs with varying average gold standard amounts.

We see that when attackers provide an incorrect value of 0.1 (solid lines), adding between 10% and 30% gold standard data nearly eliminates their effect. This shows the first benefit of gold standard data: outright identifying spammers. If attackers say that 10% of a dataset contains males, for example, when we have inserted on average 20% images of males to each HIT, the adjusted fraction for each attacker will be negative, and we can discard those workers' responses. Note that these results follow a similar pattern for different values of the true fraction, with the patterns in Figure 6-11 shifted up or down with higher or lower true fractions.

When attackers instead provide an incorrect value of 0.9 (dotted lines), it takes more gold standard data to neutralize their attack. This is because the benefit of outright spammer elimination is not available: inserting on average 20% males into tasks for which attackers claim to see 90% males will not result in a negative fraction after correction. Still, the other benefit of the gold standard approach kicks in with enough gold standard data (around 40%-70%): attackers who agreed on a single attack value are spread out by the random amounts of gold standard data on each HIT, reducing the coordination in their attack and making them easier to spot with spam elimination techniques.

As we discuss in 6.5, we could eliminate attackers colluding on a high value by estimating the inverse class or providing gold standard data of both types of classes. For example, if workers approximate 90% males, then they are claiming 10% females. Introducing a small amount of gold standard females would allow us to spot the spammers.

## 6.8 Takeaways and Discussion

Our results show that for estimation problems on datasets that exhibit visual “pop-out” effects such as images, count-based estimation is as accurate as label-based estimation, but arrives at an accurate result up to an order of magnitude faster. For datasets that require crowd workers to examine each item in detail, such as text classification, the label-based approach provides better accuracy. In our experience, the label-based approach works better with no redundant labels, instead using each additional HIT to sample more of the dataset and increasing the amount of information we learn.

Human vision and psychology research has seen many studies of people's performance at visual search tasks. A related finding from Wolfe et al. suggests that for rare (low-selectivity) items, error rates will increase [78]. Our findings suggest a different pattern, though the scenario is also different. In situations such as security scans or radiology, rare detection events (e.g., bombs or tumors) require a report, while frequent events are to go unreported. In our tasks, both rare and frequent events must be acted on by reporting a count or a label. Thus, more frequent items require counting more, resulting in higher fatigue and mistakes than low-frequency items.

We thus found higher error rates for frequent/high-selectivity items than rare items that pose less of a threat of being miscounted. Item frequency is not the only factor in human perception. The “pop-out” effect [79] also matters. Due to the effect, the relative prevalence of colors, shapes, location, and orientation of items affect reaction time and accuracy of finding a particular item. an interesting direction for future work would be to design two tasks that are only different because one presents a data item as text and the other presents the data item as an image. These two tasks would help us measure the precise effects of pop-out in our counting scenarios.

Our spam detection algorithm successfully and significantly improved count-based estimation accuracy, in some cases by more than two orders of magnitude, over techniques such as taking the average or median worker result. While we did not experience label-based spammers, if the label-based approach is susceptible to spammers in the future, we can apply our count-based spammer elimination technique to the counts of non-redundant labels, thus eliminating spammers without requiring redundant labels.

In simulations, we showed how effective sybil attackers can be at subverting the spammer elimination technique. We also showed that applying random amounts of gold standard data to every HIT can eliminate the effect of these attackers. Finally, we discussed that it is important to use gold standard data from the *least frequent* class in our dataset, so that spammers attacking an infrequent property will be identified with less overall use of gold standard data.

To put our findings into context, we can get to within 5% of the fraction of items in a dataset using about 10 HITs. Labeling datasets that require 1000 HITs takes on the order of an hour at higher batch sizes. It would take less than five minutes to process 10 HITs on a system like MTurk. Thus, in addition to saving more than an order of magnitude of work in picking the correct operator to process first, we can also make the operator selection decision quickly, saving query processing time and money.

One area for future development is in scaling up count- and label-based approaches to higher-cardinality groupings. Both interfaces are overwhelming to workers when more than five categories are available, and we are interested in interfaces that avoid this.

In this work, we study how to estimate counts for two data types: images and text. As we discover in our experiments, different interfaces are more effective for estimating properties of different data types. One extension of our work would be in broadening the space to other types, such as audio or video. Audio and video are two data types that humans can not batch process easily, and would likely require a different estimation interface. Imagine sampling a clip from each audio or video whose property one wants to estimate. One could then, for example, stream 10 clips at a time to each worker, experimenting with a serialized or parallelized presentation of the clips, and ask workers how many clips have a given property. Such an interface would reduce the overhead associated with switching between videos, but lose accuracy by only displaying clips.

Finally, given that our spammer detection technique does not require redundant labels, it would be interesting to see how we can improve result quality in domains

other than selectivity estimation. Using our technique, one could imagine an image labeling task that has high coverage but does not require redundantly labeling each item. Instead, once a worker is determined to be a spammer because their answer distribution differs from the crowd's, we could ask another trusted worker for an assessment.

## 6.9 Conclusion

In this chapter, we have shown that, for images, a count-based approach with a large batch size can achieve commensurate accuracy to a label-based approach using an order of magnitude less HITs. For text-based counts, we found that the label-based approach achieves better accuracy. We present a spammer detection algorithm that improves accuracy by up to two orders of magnitude without requiring redundant worker responses. In simulations, we show how a randomized gold standard data approach can reduce the effects of a coordinated attack by multiple workers or one worker with multiple identities. This work pushes crowdsourced computation into database optimizers, saving crowd-powered database users significant time and money.



# Chapter 7

## Discussion and Future Work

So far, we have shown how crowd-powered workflows can benefit from database techniques such as declarative query languages and optimization. We have also seen that database functionality can be enhanced by increasing the functionality of commonly used operations such as sorts and joins. In this chapter, we explore further research directions for systems such as Qurk, and discuss research that is motivated by looking at user needs in the broader crowd labor market.

### 7.1 How Crowdsourced Labor is Different

As the crowdsourced labor market grows, we see questions about the legal status of crowd work, and calls for more protection of both employers and workers. Zittrain explores several extreme real and hypothetical situations that help us define the axes on which both acceptable and unacceptable crowdsourcing practices lie [80, 81]. In one example, Zittrain describes a face-matching technology similar to the join interface described in Chapter 5. The technology behind the join logic can serve as a powerful and innocuous tool in clustering a personal photo collection. At the other end of the spectrum, however, the technology could be converted into a nefarious memory game found on a children’s educational game website, with a photoset provided by a tyrannical government that wishes to match photos of dissidents from a protest with a government ID database.

While most uses of crowdsourcing are not as extreme, they do fall along a spectrum of ethics and laws that require us to think harder about the rights of workers and employers. Quinn and Bederson outline some of the challenges faced by both parties and describe some considerations that system and platform designers should take into consideration [19]. We continue this discussion by identifying how crowdsourcing ecosystems are different from traditional employment markets through the lens of computer systems.

**The Blessing and Curse of Abstraction.** Abstraction is one of the tenets of software design, and data independence is one of the tenets of database system design. When we combine the two, it becomes easy to solve multiple problems with a single solution, but difficult to give workers and employers the agency to decide with whom

and on what to work.

The Internet and Web standards on which crowdsourcing platforms are built are internationally understood. This means that crowdsourced labor can span companies, social norms, and national boundaries. A crowd-powered image-matching user interface can be used by a photography studio or an autocratic regime without any functional effect on the crowd worker's progress or perception. Two particularly salient features of a datasource that would help workers decide whether to help process it are the originator (e.g., the Government of Iran), and the stated purpose of the task (e.g., to track dissidents). Getting reliable answers to both of these questions would be difficult in the most important situations, but it suggests interesting research in database provenance.

Abstraction is also a thorny issue for crowd employers. Crowd workflow designers think of workers in terms of price, speed, work quality, work history, and expertise. Limiting worker properties to those that affect the workflow's output is useful in designing effective systems, but also allows us to avoid answering harder questions. Qurk makes no distinction between a 27-year-old man in the Phillipines and a 12-year-old girl in Pakistan. With demographic information, employers could select workers that meet various moral and regulatory guidelines. This information is also a double-edged sword: if an optimizer discovers that males younger than 40 are best at performing a task, what restrictions on gender and age discrimination should apply? Traditional employment mechanisms are regulated in most countries to force employers to consider such worker properties or avoid them to prevent discrimination, but we have yet to tackle such issues in the crowdsourced world.

Zittrain suggests that at a minimum, we should allow crowd workers the moral agency necessary to decide which tasks to complete, and allow them to participate in collective bargaining to effect change when necessary. This requires that platforms and intermediaries collect information such as the source of a dataset and the purpose of a task. To allow employers the same moral agency in hiring, we propose adding demographic information to a worker's profile, with the understanding that we need to explore ways for that information to not be used for discrimination.

**Reduced Worker Interaction.** Crowd labor is most tangibly different from other forms of labor in that it removes a need for worker-worker and worker-employer interaction. Crowd workers interact with systems and interfaces rather than employers, which changes their ability to act and react according to traditional social norms. Workers also see less interaction with one-another: gone is the watercooler, the unscripted and informal assistance from a coworker, and the ability to discuss and propose improvements to tasks and processes with peers and superiors. Crowdsourcing-based startups are implementing virtual versions of some of these interactions, but it is important to quantify which interactions matter and how to re-implement them for the virtual world.

To the extent that this reduced interaction negatively impacts crowd work and crowd workers, it opens up avenues for research in human-computer interaction and computer supported collaborative work. How can we provide a virtual watercooler for the crowd? What systems can we build to allow workers to coach one-another? How do we identify bottom-up mechanisms for improving the work environment and

process?

**Who Regulates Crowd Work?** Many of the difficult questions in crowd work are made more difficult by the lack of clear regulatory entity for crowd labor. If an employer is in Monrovia, a crowd labor platform is in Istanbul, and a crowd worker is in New York, which countries regulate the interaction? Even if all three parties reside in America, how does the Fair Labor Standards Act apply? Which portions of the legislation is the crowd employer to uphold, which portions is the crowd platform responsible for, and which do both have to follow? Felstiner addresses many of these questions from the perspective of the American legal system [38], but more work remains in answering the difficult international issues that arise.

One set of rules that is more difficult to apply is child labor laws. Does a child in a developing nation with computer literacy, access to a computer, and enough language skills to utilize MTurk or oDesk benefit more from participating in crowd work or a potentially corrupt education system? What if the work they perform has training modules that are at least equivalent to what the child would receive in an apprenticeship? Does the level of danger the child is in differ from what they would be exposed to if working in a sweatshop? It is likely that knowledge work-based crowd labor raises yet-unanswered questions in employment and child labor regulations.

**Reduced Friction, Increased Mobility.** One argument against regulating crowd work as much as we regulate other forms of employment is that there is less overhead in establishing an employer-worker relationship in crowdsourced environments. Employers do not have to advertise work opportunities as extensively as they have to for traditional employment relationships. A worker does not have to travel for the work. Workers and employers do not have to read, understand, fill out, and sign complex agreements before each new job.

The low overhead to start work means that workers and employers have less friction in deciding to end one relationship and start another. Assuming a minimum amount of agency on both sides to get past the previously discussed abstractions, an unsavory worker or employer should be discovered early on in a relationship that could be ended quickly. Given this freedom to roam, it is possible that the extremes raised in this discussion are self-regulated without much external intervention.

## 7.2 Breaking out of Microtasks

It is not difficult to draw analogies between some kinds of crowdsourced work and the industrial production line. Many repetitive tasks on systems like MTurk require little training overhead to get started, have a large supply of workers suited to perform a task, utilize queueing theory metrics to identify performance, call for repeatable processes, and gravitate toward predictable error rates. As one follows these analogies to their natural conclusion, the draw of designing workflows through the use of microtasks increases. The smaller the task any one worker performs, the easier it is to verify that workers' quality, and the easier it is to use basic methods such as latency and throughput to quantify productivity.

For tasks such as image labeling or filtering, the microtask model is reasonable.

While crowdsourced markets are a great source of labor for these microtasks, it is also possible to do more with the crowd, ranging from building a picture book [3] on MTurk to hiring a development team on oDesk. As we move toward performing more creative tasks on crowdsourced platforms, however, we will not be able to rely on microtasks. While microtasks make it easy to measure and predict metrics, they make it difficult for workers to have enough context to do high-quality work on more creative tasks.

As an example, consider Soylent’s Shortn. The Find-Fix-Verify pattern helps keep workers on task while aggregating their work into a higher-level creative task such as paper editing. In some respects this pattern shines: a fresh pair of eyes can look at every paragraph and identify hard-to-find mistakes. In other respects, arguably those for which we needed the higher-order thinking of an editor in the first place, the system fails. Since Find-Fix-Verify is paragraph-based, it is possible for a worker to remove a sentence from a paragraph that seems unimportant, but cause a problem in a later paragraph that references the idea raised in the removed sentence. While providing workers with more context and giving them more freedom to see how their contributions affect the bigger picture may avoid such mistakes, it directly conflicts with the design requirements of microtasks.

We need a workflow that allows us to build a crowd-powered Shortn without removing as much context from tasks and still maintaining result quality. In speaking with two startups about their crowd-powered workflow design, a common design pattern emerges: *Work-Review-Spotcheck*. In the context of Shortn, the workflow would be implemented as follows. Entry-level workers are assigned a paragraph each, and told to shorten the paragraph. Their work is reviewed by a trusted party, initially bootstrapped by the person requesting the workflow, but eventually by other crowd workers who have proven themselves. The reviewer corrects mistakes and sends suggestions to the entry-level worker, who iterates on their work. To ensure that reviewers are still providing high-quality feedback and catching errors, we can periodically have reviewers spotcheck one-another, providing oversight amongst reviewers.

The Work-Review-Spotcheck model has several benefits that future research can quantify. Entry level workers provide useful work while learning. Workers are incentivized to do better through three mechanisms: bonuses for good work, promotion to reviewer status, or promotion to more complex tasks such as paper sections instead of paragraphs. Employers receive work that is vetted by high-quality workers.

There are also some issues with the Work-Review-Spotcheck model. The first involves how we bootstrap the reviewers. For example, it might be frustrating for a user who wants an automated version of Shortn to be asked to review the crowd’s work until a stable base of reviewers is identified. Bootstrapping is less of an issue for organizations that run such workflows consistently, since at steady state, there will be a pool of reliable workers. A second issue involves workflow latency. The asynchronous process of giving feedback and improving on previous work with the feedback might take longer than Find-Fix-Verify, but we might be able to design a version of Work-Review-Spotcheck in which inexperienced and experienced workers collaborate side-by-side.

Designing workflows that provide more context to workers allows for worker growth



and improved work quality with added context. To avoid bootstrapping and latency concerns, we can look to integrate these workflows into a longer-term employment model, where employers interact with workers over longer time horizons. We discuss this employment model next.

### 7.3 Moving Toward an Employment Model

As researchers, our typical crowdsourced usage patterns are sporadic, one-off processing tasks. During the course of a project, we run several experiments on systems like MTurk, analyze the results, and move on to other topics or workflows. This usage pattern is not necessarily atypical: it can also be found in an organization that temporarily needs help processing a dataset (e.g., a journalist that needs to label several thousands of images she just received from a contact). Still, this usage pattern causes us to think about workers in an adversarial way: we only see a few task submissions from each worker, we have to build redundancy into our workflows to ensure we do not trust any one worker’s output, and we often do not have the time to put workers through training modules. Because it is easier to apply canned worker quality metrics than it is to design task-specific training modules, the adversarial relationship also extends to larger crowd employers.

Several trends will move us from an ad-hoc ephemeral worker scenario to a longer employment-style relationship with workers. First, companies such as MobileWorks, and even more general-purpose platform providers like MTurk and CrowdFlower now provide specific APIs for popular tasks such as image labeling or text classification. These providers have an incentive to think about their relationship with workers who do such tasks as a long-term one, as this can improve their result quality. Ipeirotis and Horton argue that providing such task-specific APIs with a sliding price scale depending on the desired result quality will improve the market for the most popular crowdwork [45]. Second, as crowdsourced workflows are popularized, some organizations will build crowdsourced components into their everyday workflows, putting crowdsourced labor pools in the common path of these organizations rather than the one-off limited-sized task that they need completed.

In a world with longer-term relationships between employers and crowdworkers, several research opportunities develop. The guiding light of this research involves augmenting the adversarial “what workers should I filter due to bad results?” regime to one that includes the question “how can I improve the output of my workers?” Rather than redundantly asking small, verifiable questions as we do with Find-Fix-Verify in order to weed out lazy workers, we can move toward Work-Review-Spotcheck workflows in which entry-level workers are guided by more experienced workers to improve their skills.

The concept of training-based workflows was touched on by Dow et al. as they studied how to “Shepherd” crowd workers by providing them with feedback on their work [36]. Worker feedback is key in a training-based workflow. Another research opportunity is to classify the feedback a worker has received across several tasks, transparently identifying how that worker has performed along various axes (e.g., “Practice

your capitalization, and keep up the good work on avoiding sentence fragments!”). Given a long-term relationship with a worker, employers can design training modules that both onboard new workers and assign workers to training modules depending on their identified weaknesses.

Research supporting an employment model of crowd work celebrates a notion of upward mobility that we point to in effective workplaces. Providing workers visibility into the promotion process required in workflow patterns that rely on more experienced crowd workers to play the role of reviewers and trainers provides an incentive to improve oneself. While the research community has spent significant effort exploring notions of fault tolerance and redundancy in crowdsourced workflows, there remain several opportunities in studying long-term training-based workflows.

## 7.4 Alternative Optimization Objectives

In this dissertation, we explored various algorithms and optimizations that trade off the cost, time, and quality of crowd-provided responses. Beyond research prototypes, an organization has to consider other tradeoffs, such as the longevity, happiness, and alertness of their workers, and is often compelled by regulation to make other guarantees such as a minimum wage or maximum work week. One of the benefits of utilizing a system to manage interactions with crowd workers is that, assuming a policy is encodable (e.g., workers should only work 40 hours per week), the system can more accurately enforce the policy than a human would. We now explore a few high-level objectives that could be encoded into a system like Qurk.

**Minimum Wage Requirements.** Traditional workplace regulations might require employers to provide a minimum hourly wage for workers. Beyond regulation, a minimum hourly payout establishes an expectation at the outset of an employer-crowd worker relationship that defines the minimum payout a worker can expect. A pay-per-task model with a minimum wage requirement leaves in place an incentive mechanism for workers to complete more work so that they can be paid above the minimum while contributing to a relationship of trust.

Qurk could enforce such a payment mechanism through employer-paid bonuses. For each hour logged by a worker, the system could provide a bonus to any workers whose task-based pay does not meet the minimum. This mechanism would likely necessitate better metrics and employer reporting interfaces to ensure that workers whose cost-normalized quality is too low are trained to improve performance.

**Context.** As we discuss in Section 7.2, microtasks make it difficult for crowd workers to have enough context about the larger task they are completing to perform optimally. Still, microtasks make it easier to verify crowd work, and there are cases where we would like to decompose large tasks.

With some understanding of the problem being deconstructed into microtasks, Qurk could optimize for context. Consider executing Soylent’s Shortn using a Find-Fix-Verify workflow at the paragraph level. If inter-paragraph context is a concern, Qurk could, instead of showing a crowd worker a random paragraph as they complete several tasks, show them a paragraph from the same section as previous paragraphs

they have completed. The benefits of microtasks are still readily available, but the crowd worker now has more visibility into how their contributions affect the higher-level process. To generalize this solution, tasks could be defined with a desired order and affinity to other tasks. When a worker seeks another task, the system could, instead of showing a random task, bias task selection toward subsequent tasks with high affinity.

**Training.** In Section 7.3, we discuss the idea of a training-based employment model for crowd work. In such a model, we do not simply reject bad work and workers, but identify their mistakes, provide them with feedback, and assign them to go through specialized training modules.

If Qurk were mistake- and training-aware, it could optimize task assignment based on this information. For example, Qurk could classify a worker’s mistakes as being indicative of being bad at punctuation. It could then assign reviewers to provide the worker with punctuation-oriented feedback, and require the worker to complete a training module on punctuation.

**Predictable Talent Pool.** In addition to low latency and cost Qurk can optimize for predictable throughput and quality. As we move beyond crowd-powered workflows that are built for ad-hoc purposes on finite datasets, the long-term predictability of the system becomes more important. An organization might have a sense of the rate of incoming tasks that require human attention, and wants to predictably be able to handle the load by the end of a business day, for example.

For Qurk, such throughput predictions go hand-in-hand with quality metrics. Over time, Qurk can learn the aggregate quality of its longest-participating workers, and take into account the training and ramp-up time required for new workers to adjust to the tasks they are asked to perform. As a user requests throughput levels that go beyond what the high-quality crowd can provide, the system should be able to bring on and train new workers in advance of the throughput bump so that they are trained and able to produce high-quality work by the time of the expected throughput increase.

## 7.5 Designing with and for the Crowd

There are largely three parties involved in crowdsourced labor: crowd platform providers, crowd employers, and crowd workers. As facilitators, platform providers have an incentive to research and design for themselves. A significant research and design effort has gone into supporting crowd employers, as easy, effective, and understandable task generation processes are a necessary component of scaling up crowd work. The last group, the crowd laborers, has seen some attention from the economics and social sciences fields, but has received almost no attention from systems builders.

Within the crowd worker community, several efforts to improve crowd workers’ lives suggest that the current tools at workers’ disposal are not sufficient. The existence of worker-hosted message boards such as *Turker Nation* show that crowd workers are willing to share information with one-another, and that platform-based communication mechanisms are insufficient. Irani and Silberman have taken an ac-

tive position in building tools to facilitate information-sharing between workers. They built TurkOpticon [12], a browser plugin that allows Turkers to see the ratings other workers have given to potential employers. In order to better understand workers' needs, they also asked Turkers to write up a bill of rights [68]. Such efforts can give us better insight into how we can build tools to benefit crowd workers.

Building systems for and alongside crowd workers is important for several reasons. First, it allows us to address their concerns directly, allowing us to tackle ethical and economic issues faced by crowd workers that it might take regulators years to enforce. Second, optimizing with only task generation in mind can lead to locally optimal but globally sub-optimal solutions in crowdsourced markets. For example, if workers can only sort available tasks by quantity and price, a worker who wishes to perform tasks that will improve their language skills will have a difficult time finding an appropriate task. Finally, designing alongside crowd workers will give us more insight into the process and preferences of the crowd, which might result in better crowd-powered workflow design.

Given the benefits of designing with and for the crowd, we now explore some design directions from the crowd's perspective.

**Task-specific Backgrounding.** What happens in a worker's mind at the critical moment when they decide to take on a task? Horton and Chilton touch on this question from the economic perspective, identifying the distribution of worker reservation wages, or hourly wage below which that workers are not willing to take on work [43]. Embedding with and interviewing workers in order to observe the other factors that go into task selection may inform the design of worker-oriented tools.

**Reporting.** Workers in the process of finding tasks to perform on platforms such as MTurk are aided by a listing of available HIT groups that is sortable by how old a task is, how much money is paid per HIT, and how many HITs are available in a particular HIT group (e.g., "1043 remaining image labeling tasks for John Doe at \$0.01 per image"). Chilton et al. explored how workers use the available task search mechanisms in MTurk [28]. More broadly, how does a task discovery dashboard designed alongside crowd workers look?

One could add more information to the dashboard, such as data about employers with whom the worker has had a good relationship in the past, employers that have been rated highly by other workers, or the effective hourly wage of tasks from each employer. This information would provide valuable information and incentivize employers and workers to be respectful of one-another. Identifying task types that a worker has a particular skill at, preference for, or desire to become better at, would also be helpful to workers.

**Collaboration.** Given the level of information-sharing on message boards such as Turker Nation, it would be interesting to see just how much crowd workers are willing to work together. Can information-sharing in real-time help workers identify unsavory or particularly rewarding employers? Can we help workers organize into self-training, self-evaluating groups that vouch for the quality of work that a certified group member can provide? Is "pair crowd work" where two workers are able to contribute to a task side-by-side, perhaps with different levels of experience or expertise, desirable to workers and can it improve result quality for employers?

**Optimizers for Workers.** The database community has decades of experience building cost-based optimizers, and several efforts to apply these to managing crowd-powered workflows are starting to bear fruit. What if we built optimizers for crowd workers? Instead of searching for tasks to perform in a sorted list, what if crowd workers identified their preferences for task type, wage, employers, and training paths, and an optimizer automatically presented them with tasks that, as a whole, optimize for their desires? The challenges in building such a crowd-empowering optimizer are twofold:

1. Encoding workers’ desires in a method that is both intuitive to workers and meaningful for the optimizer, and
2. Explaining to workers how a selection of several tasks optimized for their preferences.

**Goal-setting and Upward Mobility.** Two related practices that traditional workplaces espouse are setting longer-term goals (e.g., “I want to learn to program in Scala”), and Upward Mobility (e.g., “My ultimate goal is to become a Creative Director one day”). An important question is whether crowd workers want to view their relationship with crowd labor as one that enables upward mobility, and if they want crowd work to help them grow professionally. If the workers want their work to facilitate this sort of growth, then how can a crowd-empowering optimizer take the learning and professional goals of crowd workers into account as we select tasks for them?

## 7.6 More System Support

As Qurk is described in this dissertation, it is an end-to-end implementation of a crowd-powered workflow engine with a query language, data model, basic crowd-powered operator set, and some crowd-aware optimizations included. There are several areas of research in expanding the system.

**Whole Plan Budget Allocation:** Qurk can determine and optimize the costs of individual query operators. We can also perform cross-operator optimizations by combining or reordering operators through the selectivity estimation engine in the Qurk optimizer. In practice, crowd-powered workflows include several operators, and a user will want to assign a single payout amount for the entire workflow.

The single budget-multiple operator problem is an interesting and only lightly explored one, especially in the face of unpredictable worker performance and response latency constraints. In the Deco system [65], Parameswaran et al. explore the cost and latency of various query plans, which is a good first step toward budget allocation.

An additional consideration arises when there is too much data to process with a given budget. We would like Qurk to be able to decide which data items to process in more detail. This optimization would require modelling how the user values different data items and optimizing against this cost function.

**Iterative Debugging:** In implementing queries in Qurk, we found that workflows would fail because of poor worker interface design or the wording of a question. Crowd-powered workflow engines could benefit from tools for iterative debugging. One of the design goals of TurKit [55] is to facilitate iterative development, where previous worker responses are cached and reused as programmers tweak crowd-powered workflows.

A fruitful direction for future work in this area is in the design of a SQL EXPLAIN-like interface that annotates operators with signals such as worker agreement, latency and throughput, cost per result, and other indicators of where a query has gone astray. Additionally, it is important to generate representative datasets for trial runs on new workflows before expending the entire budget on a buggy user interface element. This work would expand on prior research by Olston et al. in sample tuple generation for Pig workflows [61], taking into account complexities such as worker disagreement and result uncertainty.

**Runtime Optimization:** The techniques we have explored in this paper have reflected a mostly static view of per-dataset crowd-powered workflow optimizations. In practice, timing and pricing dynamics, as well as the types of workers available on a crowd platform at a given time of day, will vary.

An interesting direction for future research is in runtime pricing and optimization. If it appears that one type of task takes longer to complete than others, Qurk can offer more money for it in order to get a faster result, or have workers perform more of the task type that is bottlenecking a workflow to improve throughput. An alternative design that is useful in cases where response latency is not critical would be to schedule tasks so that they run during times when more workers are available online to potentially reduce costs.

**Task Result Cache:** Once a HIT has run, its results might be relevant in the future. For example, if the `products` table has already been ranked in one query, and another query wishes to rank all red `products`, the result of the old HITs can be used for this. Additionally, as explored in TurKit, queries might crash midway, or might be iteratively developed. In such scenarios, a cache of completed HITs can improve response time and decrease costs.

While the cache can make it easier to iteratively improve workflows and reduce unnecessary repetitive costs, cache invalidation is an art. Cache invalidation with crowd-provided answers is a semantic challenge. For example, survey results for presidential candidates might not be valid the day after a scandal is revealed, whereas product reviews are likely stable until a new product model is released. To provide users with the ability to cache results that are semantically meaningful, we can modify the Qurk query language to support cache expiration policies at both the UDF and query level.

**Model Training:** In Chapter 4, we described how machine learning models may be able to substitute for worker responses after a training phase bootstrapped from worker results. For example, a sentiment analysis algorithm can reach the precision of human raters with enough training data, but is essentially free to utilize relative to crowd workers.

From an architectural perspective, Qurk supports learning models, and can trade

off crowd responses with model responses depending on result precision and recall. A good amount of work remains in exploring how far we can take general-purpose learning models in place of human responses, how to utilize active learning to reduce the training phase, and what forms of result quality to use in deciding when to move from human- to machine-generated results.

## **7.7 Limitations of Experimental Results**

In order to conduct repeatable experiments that were fair to all study conditions, we made the decision not to modify experimental conditions based on observations made in previous experiments. Most importantly, while we had access to worker quality metrics after every experiment, we did not ban or filter out workers with poor performance in subsequent experiments. In practice, biasing toward workers who have provided accurate results in the past would improve work quality over time, much in the same way that building a stable working relationship with employees does in traditional firms. Were we to implement basic worker selection metrics, we suspect the cost-normalized result quality would increase. Because utilizing worker quality information should improve result quality, we believe that the results presented in this dissertation would be of a similar quality if repeated, and could increase in quality as worker quality metrics are applied to worker selection on larger datasets.





# Chapter 8

## Conclusion

Crowdsourcing has seen high uptake in recent years, and shows great promise in its integration into data processing systems. Adding a human in the loop of a previously automated or previously impossible task is challenging, both in terms of encoding the logic for dealing with human input and in terms of designing and optimizing human-powered tasks.

In this dissertation, we explored how declarative workflow management systems can make the crowdsourcing process easier to use and more efficient to execute. We identified meaningful properties of such systems by exploring the data and query models of Qurk alongside its architecture. Using Qurk, we were able to explore the implementation of interfaces and algorithms that optimize human-powered operators such as sorts and joins. Finally, we saw that human-powered workflows can be optimized by adapting traditional database optimization techniques such as selectivity estimation, and in the process learned about humans' ability to count using different user interfaces.

In the end, we find a symbiosis: crowds make novel forms of data processing possible, and declarative workflow management systems can optimize the process of working with crowds.



# Bibliography

- [1] Help find jim gray with web 2.0, February 2007. <http://techcrunch.com/2007/02/03/help-find-jim-gray-with-web-20/>.
- [2] How crowdsourcing helped haiti's relief efforts, March 2010. <http://radar.oreilly.com/2010/03/how-crowdsourcing-helped-haiti.html>.
- [3] Amazing but true cat stories, July 2012. <http://bjoern.org/projects/catbook/>.
- [4] Amazon's mechanical turk website, April 2012. <http://mturk.com/>.
- [5] The Apache Pig project, April 2012. <http://pig.apache.org/>.
- [6] Captricity website, April 2012. <http://captricity.com/>.
- [7] CardMunch website, April 2012. <http://www.cardmunch.com/>.
- [8] CastingWords website, April 2012. <http://castingwords.com/>.
- [9] Crowdfunder website, April 2012. <http://crowdfunder.com/>.
- [10] Mobileworks website, April 2012. <http://www.mobileworks.com/>.
- [11] Smartsheet website, April 2012. <http://www.smartsheet.com/>.
- [12] Turkopticon, May 2012. <http://turkopticon.differenceengines.com/>.
- [13] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: a system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154. VLDB Endowment, 2006.
- [14] Salman Ahmad et al. The jabberwocky programming environment for structured social computing. In *UIST*, 2011.
- [15] Paul André, Michael S. Bernstein, and Kurt Luther. Who gives a tweet?: evaluating microblog content value. In *CSCW*, pages 471–474, 2012.
- [16] Shay Assaf and Eli Upfal. Fault tolerant sorting network. In *FOCS*, 1990.
- [17] Josh Attenberg, Panagiotis G. Ipeirotis, and Foster J. Provost. Beat the machine: Challenging workers to find the unknown unknowns. In *HCOMP*, 2011.

- [18] N Barrier, Tom Leighton, Yuan Ma, and C. Greg Plaxton. Breaking the  $\theta(n \log^2 n)$  barrier for sorting with faults, 1997.
- [19] Benjamin B. Bederson and Alexander J. Quinn. Web workers unite! addressing challenges of online laborers. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 97–106, New York, NY, USA, 2011. ACM.
- [20] Michael S. Bernstein, Joel Brandt, Robert C. Miller, and David R. Karger. Crowds in two seconds: enabling realtime crowd-powered interfaces. In *UIST*, pages 33–42, 2011.
- [21] Michael S. Bernstein et al. Soylent: a word processor with a crowd inside. In *UIST*, pages 313–322, 2010.
- [22] Jeffrey P. Bigham et al. Vizwiz: nearly real-time answers to visual questions. In *UIST*, pages 333–342, 2010.
- [23] Steve Branson, Catherine Wah, Boris Babenko, Florian Schroff, Peter Welinder, Pietro Perona, and Serge Belongie. Visual recognition with humans in the loop. In *European Conference on Computer Vision (ECCV)*, Heraklion, Crete, Sept. 2010.
- [24] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.
- [25] Dana Chandler and Adam Kapelner. Breaking monotony with meaning: Motivation in crowdsourcing markets. Technical report, May 2010.
- [26] Kuang Chen, Harr Chen, Neil Conway, Joseph M. Hellerstein, and Tapan S. Parikh. Usher: Improving data quality with dynamic forms. *IEEE Trans. Knowl. Data Eng.*, 23(8), 2011.
- [27] Kuang Chen, Joseph M. Hellerstein, and Tapan S. Parikh. Data in the first mile. In *CIDR*, pages 203–206, 2011.
- [28] Lydia B. Chilton, John J. Horton, Robert C. Miller, and Shiri Azenkot. Task search in a human computation market. In *HCOMP*, 2010.
- [29] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [30] OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee. Web services business process execution language version 2.0. In *OASIS Standard*, 2007.
- [31] Peng Dai, Mausam, and Daniel S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI*, 2010.

- [32] Nilesch Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDBJ*, 16(4):523–544, 2007.
- [33] A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Journal of the Royal Statistical Society.*, 28(1):pages 20–28, 1979.
- [34] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 73–84, New York, NY, USA, 2006. ACM.
- [35] John R. Douceur. The sybil attack. In *IPTPS*, pages 251–260, 2002.
- [36] Steven Dow, Anand Kulkarni, Scott Klemmer, and Björn Hartmann. Shepherd-ing the crowd yields better work. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, CSCW '12*, pages 1013–1022, New York, NY, USA, 2012. ACM.
- [37] Bradley Efron and R.J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1993.
- [38] Alek Felstiner. Working the crowd: Employment and labor law in the crowd-sourcing industry. *Berkeley Journal of Employment and Labor Law*, 2011.
- [39] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):pages 378–382, 1971.
- [40] Michael Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.
- [41] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, 1994.
- [42] Stephen Guo, Aditya Parameswaran, and Hector Garcia-Molina. So who won? dynamic max discovery with the crowd. In *SIGMOD*, 2012.
- [43] John Horton and Lydia Chilton. The labor economics of paid crowdsourcing. *Proceedings of the 11th ACM conference on Electronic commerce EC 10*, (1):209, 2010.
- [44] Panagiotis G. Ipeirotis. Analyzing the amazon mechanical turk marketplace. *XRDS*, 17:16–21, December 2010.
- [45] Panagiotis G. Ipeirotis and John J. Horton. The need for standardization in crowdsourcing. In *Proceedings of the Workshop on Crowdsourcing and Human Computation at CHI*, 2011.

- [46] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *HCOMP*, 2010.
- [47] Panos Ipeirotis. Demographics of mechanical turk. Technical report, March 2010.
- [48] Panos Ipeirotis. Identify Verification (and how to bypass it), March 2012. <http://www.behind-the-enemy-lines.com/2012/01/identify-verification-and-how-to-bypass.html>.
- [49] David R. Karger, Sewoong Oh, and Devavrat Shah. Iterative learning for reliable crowdsourcing systems. In *NIPS*, pages 1953–1961, 2011.
- [50] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2):pages 81–93, 1938.
- [51] Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E. Kraut. Crowdforge: crowdsourcing complex work. In *UIST*, pages 43–52, 2011.
- [52] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. In *IEEE Transactions on Computers*, 1991.
- [53] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):pages 1–55, 1932.
- [54] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Exploring iterative and parallel human computation processes. In *HCOMP*, 2010.
- [55] Greg Little et al. TurkIt: human computation algorithms on mechanical turk. In *UIST*, pages 57–66, 2010.
- [56] Adam Marcus, Eugene Wu, et al. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
- [57] Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, and Robert C. Miller. Demonstration of qurk: a query processor for humanoperators. In *SIGMOD Conference*, pages 1315–1318, 2011.
- [58] Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, and Robert C. Miller. Human-powered sorts and joins. *PVLDB*, 2011.
- [59] Winter Mason and Duncan J. Watts. Financial incentives and the “performance of crowds”. In *HCOMP*, pages 77–85, 2009.
- [60] Jon Noronha, Eric Hysen, Haoqi Zhang, and Krzysztof Z. Gajos. Platemate: crowdsourcing nutritional analysis from food photographs. In *UIST*, pages 1–12, 2011.
- [61] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD ’09, pages 245–256, New York, NY, USA, 2009. ACM.

- [62] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, New York, NY, USA, 2008.
- [63] Aditya Parameswaran et al. Crowdscreen: Algorithms for filtering data with humans. 2011.
- [64] Aditya Parameswaran and Neoklis Polyzotis. Answering queries using databases, humans and algorithms. In *CIDR*, 2011.
- [65] Hyunjung Park, Richard Pang, Aditya Parameswaran, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Deco: A system for declarative crowdsourcing. Technical report, Stanford University, March 2012.
- [66] Joel Ross, Lilly Irani, M. Six Silberman, Andrew Zaldivar, and Bill Tomlinson. Who are the crowdworkers?: shifting demographics in mechanical turk. In *CHI Extended Abstracts*, New York, NY, USA, 2010. ACM.
- [67] Larry Rudolph. A robust sorting network. In *IEEE Transactions on Computers*, 1985.
- [68] M. Six Silberman, Joel Ross, Lilly Irani, and Bill Tomlinson. Sellers' problems in human computation markets. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 18–21, New York, NY, USA, 2010. ACM.
- [69] James Surowiecki. *The Wisdom of Crowds*. Anchor Books, 2005.
- [70] Omer Tamuz, Ce Liu, Serge Belongie, Ohad Shamir, and Adam Kalai. Adaptively learning the crowd kernel. In *ICML*, pages 673–680, 2011.
- [71] F. Tarrés and A. Rama. GTAV Face Database, March 2012. <http://gps-tsc.upc.es/GTAV/ResearchAreas/UPCFaceDatabase/GTAVFaceDatabase.htm>.
- [72] Beth Trushkowsky, Tim Kraska, Michael J. Franklin, and Purnamrita Sarkar. Getting it all from the crowd. *CoRR*, abs/1202.2335, 2012.
- [73] Luis von Ahn. Games with a purpose. *IEEE Computer*, 39(6):92–94, 2006.
- [74] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: Using hard ai problems for security. In *EUROCRYPT*, pages 294–311, 2003.
- [75] Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M. Hellerstein. BayesStore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.
- [76] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. CrowdER: Crowdsourcing Entity Resolution. *PVLDB*, 2012.

- [77] Edwin Bidwell Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.
- [78] Jeremy M. Wolfe, Todd S. Horowitz, Michael J. Van Wert, and Naomi M. Kenner. Low target prevalence is a stubborn source of errors in visual search tasks. In *Journal of Experimental Psychology*, 2007.
- [79] Jeremy M. Wolfe and H. Pashler (Editor). *Attention*, chapter Visual Search, pages 13–73. University College London Press, 1998.
- [80] Jonathan Zittrain. Ubiquitous Human Computing. *SSRN eLibrary*, 2008.
- [81] Jonathan Zittrain. Human Computing’s Oppenheimer Question, May 2012. <https://www.youtube.com/watch?v=dqnQTEhPOPc>.