Proceedings of the Fifth Conference on
Human Computation and Crowdsourcing
(HCOMP 2017)

Dynamic Filter: Adaptive Query Processing with the Crowd

Doren Lan, Katherine Reed, Austin Shin, Beth Trushkowsky
Department of Computer Science
Harvey Mudd College
Claremont, CA
dlan@hmc.edu, kireed@hmc.edu, ashin@hmc.edu, beth@cs.hmc.edu

Abstract

Hybrid human-machine query processing systems, such as crowd-powered database systems, aim to broaden the scope of questions users can ask about their data by incorporating human computation to support queries that may be subjective and/or require visual or semantic interpretation. A common type of query involves filtering data by several criteria, some of which need human computation to be evaluated. For example, filtering a set of hotels for those that both (1) have great views from the rooms, and (2) have a fitness center. Criteria can differ in the amount of human effort required to decide if data satisfy them, due to criterion's subjectivity and difficulty. There is potential to reduce crowdsourcing costs by ordering the evaluation of each of the criteria such that criteria needing more human computation are not processed for data that have not satisfied the less costly criteria. Unfortunately, for queries specified on-the-fly, the information about subjectivity and difficulty is unknown a priori.

To overcome this challenge, we present *Dynamic Filter*, an adaptive query processing algorithm that dynamically changes the order in which criteria are evaluated based on observations while the query is running. Using crowdsourced data from a popular crowdsourcing platform, we show that Dynamic Filter can effectively adapt the processing order and approach the performance of a "clairvoyant" algorithm.

Introduction

The availability of micro-task crowdsourcing platforms like Amazon's Mechanical Turk (AMT) that provide a programmatic interface to recruit workers for human computation tasks has led to the development of applications and systems that directly incorporate human knowledge, experience, and perception. Recent work investigates using this "crowdsourcing" within query processing systems such as a database management system (DBMS) to broaden the scope of questions users can ask about their data (Franklin et al. 2011; Marcus et al. 2011b; Parameswaran et al. 2012b). This work on hybrid human-machine query processors aims to enhance traditional machine algorithms with human computation to support queries that may be subjective and/or require visual or semantic interpretation. This paper focuses on an important operation of a crowd-powered query processor: filtering a set of data by several criteria, called *predicates*, some of which need human computation to be evaluated. Notably, different predicates can require different amounts of human effort, due to factors such as the subjectivity of the predicate and the difficulty in evaluating it.

To illustrate this, consider the following query that is well-suited for a crowd-powered DBMS. Given a list of hotels, return those hotels that satisfy all three of these constraints, or *predicates*: (1) has a gym that is open 24 hours a day, (2) has a nice view from the room, and (3) is located within a mile of a metro station with a direct line to downtown.

This simple query demonstrates several important features that can influence the most efficient way to process a query with the help of human computation. First, it may take a worker more time and effort to evaluate some predicates for a particular hotel. In this case, it may be easier to check a hotel's website for gym hours than it is to map out transit options. Second, predicates may have differing levels of subjectivity; determining if a hotel has the desired atmosphere may require aggregating the opinions of multiple workers, a process that further increases the amount of work to be done to evaluate that predicate. A key observation from this example is that a hotel that does not pass the gym requirement, which is cheap for a worker to evaluate, does not need to be checked for either of the expensive predicates, demonstrating the importance of the predicates' order of execution.

However, for queries specified on-the-fly, predicates' subjectivity and difficulty, as well as their likelihood of evaluating to true or false, i.e., their *selectivity*, are statistics that are unknown a priori. This property of the execution environment prevents us from constructing an optimal ordering of the predicates based on predicate rank (Hellerstein and Stonebraker 1993). An ordering which first processes predicates that are expensive and likely evaluate to true could result in wasted work and unnecessary cost to the user.

To evaluate crowd-powered queries with multiple predicates for which the aforementioned statistics are unknown at query time, we take an *adaptive query processing* approach (Deshpande, Ives, and Raman 2007): we dynamically change the order in which predicates are evaluated while the query is running, based on information gathered during runtime. To this end, we propose the Adaptive Crowd-Filtering problem that decides on-the-fly the order

of processing of predicates for each item in the data; the decision for the next item is based on observations about the selectivity and amount of work required for previously processed items. We then present Dynamic Filter, an adaptive algorithm that alters the predicate execution order without needing to explicitly measure statistics like selectivity and amount of work. Instead of direct measurements, the algorithm uses proxies for each statistic. A clogged "input queue" for a predicate indicates it requires a lot of work. Selective predicates are prioritized in a lottery scheduling scheme adapted from work on adaptive query processing (Avnur and Hellerstein 2000); a predicate is given a "ticket" when an item does not satisfy it, a reward for obviating the need for further processing on that item by other predicates.

Using crowdsourced data gathered from AMT, we show that Dynamic Filter can effectively adapt the predicate processing order and approach the performance of a "clairvoyant" algorithm that knows the optimal ordering a priori. In summary, the contributions of this paper are:

- A formal definition of the Adaptive Crowd-Filter problem, which is the challenge of filtering data based on multiple predicates without a priori knowledge of statistics like predicate selectivity or work required.
- The Dynamic Filter algorithm for reordering predicates that addresses the Adaptive Crowd-Filter problem without directly measuring runtime statistics.
- An evaluation of Dynamic Filter using crowdsourced data gathered from AMT, a popular human computation platform for micro-tasks.

The paper is organized as follows: first we provide background and elaborate on the properties of filtering data with the crowd that motivate our adaptive approach; we follow that with a discussion of how our work relates to existing work. We then provide a formal definition of the Adaptive Crowd-Filter problem and we describe the Dynamic Filter algorithm. We discuss experimental results using crowdsourced data before we conclude.

Background and Motivation

On human computation platforms like Amazon's Mechanical Turk (AMT), workers complete small units of work called *micro-tasks*, or *tasks*, posted by requesters in exchange for payment. It is common crowdsourcing practice to have multiple workers complete the same task and aggregate the responses for quality management purposes (Sheng, Provost, and Ipeirotis 2008; Ipeirotis, Provost, and Wang 2010). Tasks that are more subjective or ambiguous may require more tasks to reach consensus. Furthermore, some tasks may take longer due to difficulty or amount of worker effort needed; workers may also abandon difficult tasks.

For crowd-based filtering, tasks involve evaluating if a given data item satisfies a given predicate. Thus qualities of the predicates influence how long a query will take to process and how many tasks will be required: subjective predicates may require more tasks and difficult predicates will influence the time it will take to process. Number of tasks and processing time are aspects of the query's *cost*.

The hotel example described earlier illustrates that it is possible to avoid processing a costly predicate for a given item if that item has already been disqualified by another predicate. The likelihood that a predicate will return true or false is known as predicate *selectivity*. However, the cost information, as well as the selectivity, is unknown when a query begins execution. This is because the query is *ad hoc*, i.e., specified on the fly as the user decides what query they want to run. It is a goal of crowd-powered query processing systems to support these ad hoc queries. Work investigating optimization for crowd-powered DBMSs (Park and Widom 2013; Fan et al. 2015) assumes these statistics are known.

Another challenge is that the environment in which the crowd-powered filter query would be running may change over time. Workers on platforms like AMT decide when they want to complete tasks. Workers typically choose to complete different numbers of tasks as well; a few typically do a lot of tasks, while many choose to do only a few (Heer and Bostock 2010). This shifting of the crowd could cause the cost of predicates to change due to different worker approaches to tasks.

An adaptive query processing algorithm can tackle both (1) the lack of prior statistics that inform an optimal predicate ordering and (2) the changing of those statistics over time by periodically observing runtime performance and adjusting its strategy over time.

Related Work

The work in this paper is done in the context of crowd-powered database management systems (DBMSs), such as CrowdDb (Franklin et al. 2011), Deco (Parameswaran et al. 2012b), and Qurk (Marcus et al. 2011a). In a traditional DBMS, finding an efficient execution strategy for a query (a *query plan*) is the role of a cost-based query optimizer, which involves finding an order for operations on the data (Chaudhuri 1998). There has been recent work in query optimization for crowd-powered DBMSs. The CrowdOp system (Fan et al. 2015) devises a low-cost query plan that orders crowd-based operations such as combining data (JOIN), filling in missing data (FILL), and filtering data (SELECT). The authors investigate optimizing for monetary cost and latency given a crowdsourcing budget. Park and Widom (2013) look at estimating the size of intermediate results when executing a query, the cost of which is influenced by how much more crowdsourced data is required in addition to pre-existing data. The work done by Park and Widom (2013) and in CrowdOp (Fan et al. 2015) assumes selectivity information is known when constructing a query plan and does not adapt to a changing environment.

Some work in crowd-powered DBMSs has looked into filtering data with the crowd. In CrowdScreen (Parameswaran et al. 2012a), the authors look at incorporating a filter's selectivity as well as workers' false positive and false negative error rates to determine a strategy that minimizes the number of questions asked to the crowd while keeping the total error below a threshold. In follow-up work to CrowdScreen (Parameswaran et al. 2014), the authors incorporate individual worker abilities as well as per-item selectivity information into their approach. However, selectivity information is

assumed to be known. Work on CrowdFind (Das Sarma et al. 2014) investigates the cost versus latency tradeoff for filtering with the crowd to find k items that satisfy a predicate. The authors look at asking the crowd about items one-at-a-time and stopping after k are found, versus parallelizing by sending items to multiple workers at once. They do not consider the ordering of multiple predicates. Work on counting with the crowd (Marcus et al. 2012) investigates user interfaces for asking the crowd to estimate how many items in a set satisfy some constraint. In our work, we filter data on-the-fly without first asking the crowd to estimate selectivity.

Adjusting a query plan while the query is running is the goal of *adaptive query processing* (see Deshpande et al. 2007 for a survey). In this work we adapt techniques from the "Eddies" work (Avnur and Hellerstein 2000), an adaptive query processing approach that considers the order of operations for each item that is processed. Similar to Eddies, our approach does not directly measure selectivity or cost, as opposed to other adaptive query processing approaches which sample selectivity to re-adjust the query plan, such as the work by (Babu et al. 2004). We elaborate when we discuss our approach later on.

AskSheet (Quinn and Bederson 2014) is a spreadsheet application that can ask crowd workers to fill in cells. The application decides which cells to ask about first based on the estimated probability it will need that information to compute the user's final result; this saves overall crowdsourcing cost when some cells do not need to be crowdsourced. However, AskSheet does not consider if cells have different costs due to differing levels of ambiguity, nor how cost and selectivity could change over time, and makes some uniform distribution assumptions.

Recent work in micro-task assignment and scheduling involves choosing a worker for a task based on expertise, required skills, or history of performance in order to satisfy a quality objective, e.g., (Yuen, King, and Leung 2011; Ho and Vaughan 2012; Nunia et al. 2013; Rajan et al. 2013; Karger, Oh, and Shah 2014; Basu Roy et al. 2015), and scheduling tasks across work for multiple requesters (Difallah, Demartini, and Cudré-Mauroux 2016). Our work focuses on efficiency and does not assume workers have expertise; we also focus on efficiency for a single query.

Adaptive Crowd-Filter Problem

In this section we propose the Adaptive Crowd-Filter problem to filter data with multiple constraints called *predicates*. Properties of the predicates, namely their cost and selectivity, contribute to the efficiency of the filtering process. With these properties unknown at query time, the objective of the Adaptive Crowd-Filter problem is to adjust how the query is being processed to increase efficiency and thereby reduce overall query cost.

We first provide definitions and formalize the objective. We then discuss the role of predicate ordering in reducing query cost, and detail a general form for an algorithm addressing the Adaptive Crowd-Filter problem. Our particular approach follows in the next section.

Preliminary Definitions

A *query* is a set of predicates $Pred$ with which to filter a set of data items D. We assume the set of predicates is known and each is a question with a yes or no answer, e.g., from a declarative query specified in SQL; extracting predicates from natural language is an interesting area of future work.

Predicates are boolean functions that yield true or false for a given item, indicating whether an item *satisfies* the predicate. More formally, for a predicate p and a given item i to process, $p(i) \in \{1, 0\}$, where 1 and 0 represent true and false, respectively. Evaluating a query yields $D_{result} \subseteq D$, the set of items that satisfy all predicates in $Pred$:

$$D_{result} = \{i \in D | \forall_{p \in Pred}, p(i) = 1\}$$

The *selectivity* of a predicate p represents the likelihood that p will yield 1. A predicate with a low selectivity value would evaluate to false for many items; such a predicate is considered to be *selective*. The selectivity of p is estimated as:

$$selectivity_p = \frac{\sum_{i \in D} p(i)}{|D|} \quad (1)$$

In the crowd-filter context, a *task* involves a single worker evaluating a particular item i with a particular predicate p; we will call $\{i, p\}$ an *item-predicate pair*. Having each task be for one item-predicate pair instead of, say, evaluating all predicates for a given item allows the query processor to avoid evaluating a costly predicate for an item if that item has already failed a different predicate. As mentioned earlier, it is common practice to aggregate the responses from different workers for the same task to decide on the final outcome. Multiple approaches exist for aggregating responses; in our evaluation we use the approach from Sheng et al. (2008).

Let $Tasks_{i,p}$ be the set of tasks needed to reach consensus about whether item i satisfies predicate p. Each task $t \in Tasks_{i,p}$ has an associated cost $cost_t$, a value which can be a metric such as the time in seconds it took a worker to complete the task, the reward paid to the worker for the task, or simply 1 to signify that each task has equal weight. We can define the *cost* of evaluating an item-predicate pair as the sum of costs of the tasks to reach consensus:

$$cost(i, p) = \sum_{t \in Tasks_{i,p}} cost_t \quad (2)$$

And the overall query cost as the sum of costs for each item:

$$query\ cost = \sum_{i \in D} \sum_{p \in Pred} I_{i,p} * cost(i, p) \quad (3)$$

where $I_{i,p} \in \{0, 1\}$ is an indicator variable that signifies whether item i needed to be processed by predicate p. Note that this expression for query cost is an observed value, not an estimated or expected value.

Predicate Order to Minimize Query Cost

The objective of the Adaptive Crowd-Filter problem is to minimize the cost of the query from Equation 3, given that selectivity and cost information (Equations 1 and 2, respectively) is unknown at the time the query is executed. The key

to minimizing query cost is to schedule items to be evaluated by each predicate in an order that avoids processing costly predicates as much as possible.

Let $(p_1, p_2, \ldots, p_{|Pred|})$ be an ordering of the predicates in $Pred$ such that an item i is processed by p_{j+1} if and only if $p_j(i) = 1$. Intuitively, if each predicate had equal (known) cost, the optimal processing order would be in ascending order of predicate selectivity. To illustrate this idea, suppose two predicates p_1 and p_2 have selectivities 0.1 and 0.75, respectively. When ordered (p_1, p_2), p_1 will process $|D|$ items and p_2 will process $0.1|D|$ items. If ordered (p_2, p_1), p_2 processes $|D|$ items and p_1 does $0.75|D|$. The latter ordering results in more item-predicate pairs to be evaluated and thus higher query cost. A similar analysis could be done for ordering predicates with equal selectivity to yield an ordering that is in ascending order of predicate cost.

For predicates with varying cost and selectivity, an optimal predicate ordering is in ascending order of *predicate rank* (Hellerstein and Stonebraker 1993), where rank is:

$$predicate\ rank = \frac{selectivity - 1}{cost\ per\ item}$$

The notion of predicate rank illustrates the importance of both selectivity and cost in reasoning about ordering predicates. The work in this paper addresses a query processing environment in which both statistics are unknown at query time, a challenge for predicate ordering.

Task Assignment in Adaptive Crowd-Filter

This work targets a crowdsourcing environment like Amazon's Mechanical Turk (AMT) in which workers arrive and choose to complete tasks. Given this environment, the Adaptive Crowd-Filter problem can be thought of as iterative task assignment: for the next requested task for a filter query, which item-predicate pair should the task contain? Algorithm 1 shows the iterative process of choosing the next task. The first two lines of Algorithm 1 initialize the state by setting up all possible item-predicate pairs in the map `ipPairs`, which maps each predicate to the set of items it has left to process. The loop (lines 3-15) iteratively chooses the next task for a worker until there are no item-predicate pairs remaining in `ipPairs`. The choice of the next item-predicate pair is made on-demand when a worker requests a task (lines 4-5); workers should be prevented from evaluating the same item-predicate pair multiple times. We do not assume worker expertise. After the worker evaluates whether the selected item satisfies the selected predicate, his/her vote is aggregated with all other votes for that item-predicate pair to determine if the pair has enough votes to reach a consensus (lines 6-7). If consensus is reached, the item is either (1) removed from all predicates' set of items left to be processed, if the consensus was false, or (2) removed from just the selected predicate's set of items (lines 8-13). Otherwise the item-predicate pair needs to be processed by another worker.

Algorithm 1: Iterative task assignment

Input: Set of items D and set of predicates $Pred$

```
1  Map ipPairs ; /* map pred. → items    */
2  foreach predicate p ∈ Pred do ipPairs(p) ← D;
3  repeat
4      w ← workerArrival();
5      item,pred ← chooseNextTask(w,ipPairs);
6      w.doTask(item,pred);
7      ipPairStatus ← aggregateVotes(item,pred);
8      if ipPairStatus is a consensus for false then
9          foreach predicate p ∈ Pred do
10             ipPairs(p) ← ipPairs(p) - item
11         end
12     else if ipPairStatus is a consensus for true then
13         ipPairs(pred) ← ipPairs(pred) - item;
14     end
15 until no item-predicate pairs remain in ipPairs;
```

The former case regarding consensus is the scenario in which query cost savings are realized: when an item does not satisfy one of the predicates, it no longer needs to be processed by any remaining predicates. The challenge for implementing `chooseNextTask()` is to route items first towards less costly, more selective predicates. With cost and selectivity statistics unknown a priori, this effort includes learning these statistics and adapting the routing choices as the query is running.

Dynamic Filter Algorithm

We now present *Dynamic Filter*, an adaptive algorithm for crowd-based filtering with multiple predicates. As described in the previous section, ordering of predicates based on selectivity and cost may decrease the number of required tasks and thus the algorithm should route items first to predicates that are selective and have lower cost. To this end, Dynamic Filter learns and adapts to knowledge about predicates' cost and selectivity. In choosing the next item-predicate pair in `chooseNextTask()`, Dynamic Filter explores predicates' behavior to refine its model of costs and selectivities while exploiting its current notion of the best predicate ordering based on that model; this exploration and exploitation process is similar to a multi-armed bandit problem (Berry and Fristedt 1985).

We begin with an overview of Dynamic Filter's workflow and then describe in more detail the aspects of the algorithm for choosing the next item-predicate pair for a worker to evaluate.

Workflow Overview

Rather than explicitly updating estimates of cost and selectivity for each predicate, computing predicate rank, and then deciding how to balance exploration versus exploitation, we adapt techniques from work on Eddies in adaptive query processing (Avnur and Hellerstein 2000) to use other state as proxies for these statistics. Dynamic Filter uses these proxies to choose the next item-predicate pair, incorporating exploration while moving the algorithm towards a low-cost and selective predicate ordering; this process is done without needing to maintain cost or selectivity information.

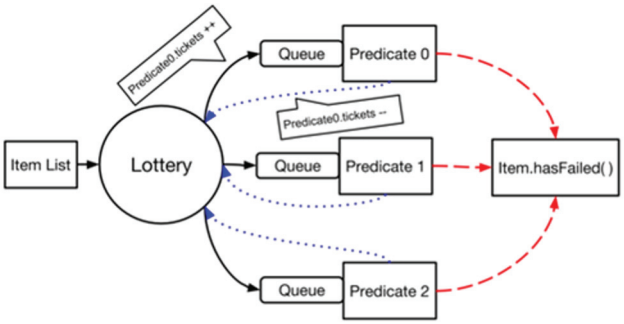

Figure 1: Routing items. Dotted arrows represent when the item satisfies the predicate, and dashed lines indicate where the item is routed when it fails a predicate.

The diagram in Figure 1 depicts how Dynamic Filter decides which predicate an item should be sent to for processing. We first describe the flow before elaborating on the role of the lottery and queue below. To determine which predicate a given item should be sent to, a "lottery drawing" is held; this is a lottery scheduling approach (Waldspurger and Weihl 1994). Initially each predicate has a single lottery ticket and thus is equally likely to win. The item is placed on the queue for the winning predicate and that predicate is given another ticket. Once the item has enough votes to reach consensus about whether it satisfies the predicate, the item is routed as follows: if the item fails the predicate, it needs no further processing; otherwise, the predicate must return its ticket and the item returns to the lottery to be sent to another predicate.

Lottery scheduling: accounting for selectivity The lottery system serves as a proxy for selectivity (Avnur and Hellerstein 2000). A predicate is given a ticket every time an item is sent to it by the lottery. Once the item is evaluated in the predicate, the item is either true or false for that predicate. If the item returns true, the number of tickets for that predicate is decremented and the item is sent back into the lottery. However, if the item returns false, it no longer needs evaluation by any remaining predicates. To reward the predicate for removing an item from further processing, the predicate *retains* the ticket it received for the item. As a result, the more a predicate returns false, i.e., the more selective it is, the more tickets it receives over time and consequently, more items are routed to this predicate.

Predicate queues: accounting for cost While a predicate may be selective, it may also be expensive. Predicates with high cost are best avoided because the number of tasks required to complete each item sent to this predicate may also be high. To avoid constantly routing items to a high cost predicate, each predicate contains a queue that holds items that are waiting to be evaluated by the predicate. While the queue is full, additional items are no longer routed to that predicate. The more expensive a predicate is, the more tasks it takes to evaluate the item passed to it. Thus its queue will remain full longer, reducing the number of items that end up being routed to that predicate and thereby increasing the number of items sent to less expensive predicates instead.

With its queuing and lottery systems, Dynamic Filter will begin to favor cheaper and more selective predicates over time: cheap predicates will receive more items into their queues, which means they have a higher chance of acquiring more tickets, and more selective predicates will have a higher chance of retaining tickets.

Choosing the Next Task

Algorithm 2 shows how Dynamic Filter implements the procedure `chooseNextTask()` from Algorithm 1. To construct a task for a worker, first a predicate is chosen based on a lottery drawing (line 2). An item is then chosen from that predicate's queue (line 6), which is replenished if the queue has availability (lines 3-5).

Algorithm 2: Choosing next task in Dynamic Filter

Input : A worker w for whom to create a task,
ipPairs as in Algorithm 1,
Map predQueue of pred → items queued
Output: An item and predicate for w

```
1 procedure chooseNextTask
2     pred ← runLottery();
3     if predQueue(pred) has a slot available then
4         add to it an item from ipPairs(pred);
5     end
6     item ← an item from predQueue(pred);
7     return item, pred;
8 end
```

Note that the lottery drawing occurs per task. Thus, unlike in the Eddies work (Avnur and Hellerstein 2000), the lottery scheduling is actually acting as a scheduler for the predicates: predicates are allocated tasks proportional to their relative ticket count. With workers completing tasks in parallel, more worker effort is sent to predicates with more tickets.

Sliding Window: Adapting to Fluctuations

When dynamically filtering items, the algorithm should be able to adapt to fluctuations in selectivity or cost over time. If the algorithm only used the ticketing and queue systems discussed so far, a specific predicate may become heavily weighted over time due to possessing many more tickets. As a result, the algorithm will have difficulty adapting quickly to changes in the predicates' relative selectivity or cost. To add more adaptivity, we adopt the technique from Eddies (Avnur and Hellerstein 2000) and give each ticket a *lifetime*. Every time a predicate is chosen by the lottery, that predicate's tickets become "older". If a ticket reaches its lifetime threshold, it is removed from its predicate. This ensures that predicates must continually remain selective and complete items sooner than the lifetime of their tickets in order to maintain their number of tickets. In essence, this new ticketing system acts as a sliding window that focuses on the most

recent behavior of the predicates and ignores how well each predicate behaved in the far past.

Experimental Evaluation

In this section we demonstrate the efficacy of Dynamic Filter in comparison to several baseline algorithms. We use simulation experiments using crowdsourced data gathered from Amazon's Mechanical Turk (AMT), a popular crowdsourcing platform.

Experiment Setup

We collected two batches of crowdsourced data to use in simulations of Dynamic Filter and comparison algorithms. One batch was five questions (predicates) about ninety hotels in a city in Missouri, and the other was a batch with ten questions about twenty restaurants in a city in California. The set of hotels and restaurants were chosen by the authors based on knowledge of the cities; the authors chose questions they predicted would show variety in selectivity and subjectivity. Each task, which paid $\$0.10$, presented a worker with one question and one item, and asked the worker to respond yes or no. The average task completion time was 1.5 minutes for the hotels and 54 seconds for the restaurants. Each item-predicate pair was evaluated by twenty-one workers. We used majority vote from the twenty-one responses as ground truth for hotels; the authors provided ground truth for restaurants.

The Dynamic Filter simulation implements Algorithms 1 and 2 and uses the crowdsourced data described above. After selecting an item and predicate for the next task, the simulation samples without replacement from the set of worker responses for that item-predicate pair; this sampling implements `doTask(item,predicate)` in the pseudocode. When aggregating worker responses (votes), we use the technique from Sheng et al. (2008) that estimates an uncertainty value signifying the likelihood that the majority vote from a set of $\{$yes, no$\}$ votes is the true value. In our experiments, we require at least five votes for an item-predicate pair before aggregating; twenty-one is the maximum possible. We used an uncertainty threshold of 0.2. This means that the level of uncertainty must be below 0.2 in order for the votes to be considered a consensus. We use a queue size of 1 to easily keep track of which items are in the queues. Simulation results are aggregated from $50-200$ runs of each algorithm. We first present experiments without windowing enabled, and contrast with windowing afterwards.

Experiments and Results

For the experiments we detail next, we first use different combinations of two predicates to demonstrate a range of how Dynamic Filter responds to different predicate selectivities and costs. We chose the particular two-predicate combinations to specifically illustrate cases when the selectivities are similar but the costs are different versus when the costs are the same but the selectivities differ. We calculated observed selectivity using the majority vote and observed cost as the average number of votes to reach consensus for each predicate. The experiments using two predicates also allow us to show the comparison against all possible static predicate orderings on one graph. Towards the end of this section we report on experiments with more than two predicates to demonstrate generality of our approach.

We implemented a few baseline algorithms against which to compare. The first is an algorithm that chooses a predicate uniformly at random when an item should be routed to a predicate. We chose to implement this algorithm to observe the benefit of Dynamic Filter's approach to choosing a predicate versus just randomly picking a predicate. In addition, because each experiment has two predicates, we also implemented the two possible *static ordered* filtering algorithms: one that always chooses to send items to the "optimal" predicate first, and one that sends items to the "worst" predicate first. We determine the optimal and worst orders using the observed selectivity and average cost, as described above.

In these experiments, we focused specifically on comparing our algorithm to the optimal algorithm and the random algorithm. The metric we will observe is the number of tasks to finish processing the query.

Varying selectivity Table 1 shows the first configuration: predicates with the same cost but very different selectivities. This configuration shows how the ticketing system affects the number of tasks used by Dynamic Filter. Both predicates had low cost to ensure that selectivity would be the main contributor to variation in the number of tasks.

Figure 2 shows the distribution of the number of tasks after running 50 simulations for the Dynamic Filter algorithm and the other baseline algorithms. We can see that Dynamic Filter outperforms the worst-case static algorithm and

Predicate	Selectivity	Cost
Does this hotel have a gym?	84%	5.0 votes
Does this hotel cost under $80 a night?	12%	5.0 votes

Table 1: Varied selectivity, 5.5% of items satisfy both.

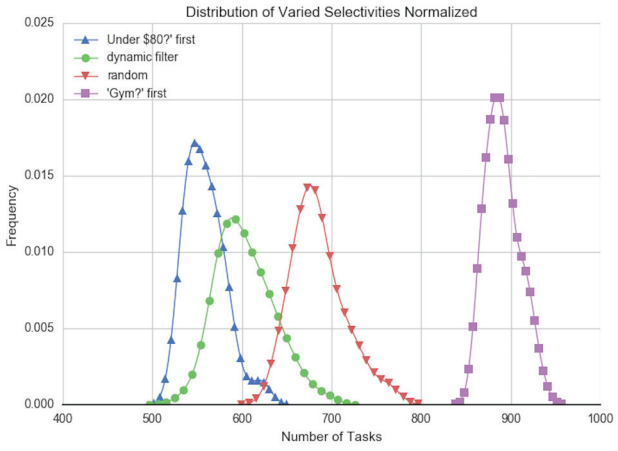

Figure 2: Distributions of number of tasks in the varied selectivity configuration. Distributions shown as continuous density plots; area under curves normalized to sum to 1.

the random algorithm, and remains close to the optimal in which the more selective predicate is always chosen first. When comparing Dynamic Filter to the random algorithm, we found that the mean number of tasks to process the query were 603.84 and 685.52 respectively; the difference between the two algorithms is statistically significant (the standard deviations of Dynamic Filter and random were 31.37 and 29.06, respectively; the t-value was 13.50 and the p-value was < 0.0001). As shown by the simulation runs, choosing the "Gym?" question first more often results in fewer overall tasks assigned. Dynamic Filter averaged around 55 more tasks than the optimal. This difference represents how many tasks it took to learn which predicate was better to first send items. Dynamic Filter and random had accuracies of 0.986 and 0.987, a difference that was not significant. Accuracy is calculated as $(TP + TN)/(TP + TN + FP + FN)$. The precision and recall were also not significantly different.

Varying cost We also investigated the performance of Dynamic Filter with predicates with different costs that are selective. This configuration (shown in Table 2) shows how the queue system of Dynamic Filter, in combination with the ticketing system, responds to the different cost between predicates and if it is able to effectively decrease the number of tasks required. We chose more selective predicates for this combination to show the benefit of avoiding costly predicates, as not satisfying a cheaper predicate obviates the need to be processed by the costly predicate. If both predicates were not selective, many items would be processed by both predicates, regardless of which was chosen first.

Predicate	Selectivity	Cost
Does this hotel cost under $80 a night?	12%	5.0
Are there great views from the rooms of this hotel?	38%	9.7

Table 2: Varied cost, no items satisfy both.

Figure 3 shows the distribution of required tasks for this combination. We can see again that Dynamic Filter takes fewer tasks on average to filter than the random and worst case algorithms. In comparison with optimal, the algorithm only takes 12% more tasks. The ticketing and queue systems worked in tandem to prevent the more expensive predicate from processing many items. As the "Great Views?" question required more votes on average to reach consensus on an item, the queue prevents other items from entering this predicate's queue until the predicate was finished. The decrease in items routed to the expensive predicate led to the other predicate receiving more new items, which incremented its amount of tickets. Thus, the ticketing and queue systems worked together to sense which predicate had greater cost.

The average number of tasks was 626.90 for Dynamic Filter and 674.08 for the random algorithm; the difference was significant (the standard deviations of Dynamic Filter and random were 31.67 and 26.87, respectively; the t-value was 8.03 and the p-value was < 0.0001). However, the difference between these two algorithms is less in this configuration than in the previous configuration because our algorithm is more aggressive when it comes to differentiating between selectivity. In addition, the difference in the costs between the two predicates in this configuration does not range as greatly as the two predicates' selectivity in the previous configuration. Also, while these two predicates lean towards more selective, their selectivities are not the same, which may also affect the differences between Dynamic Filter and the random algorithm. All these factors taken into consideration, Dynamic Filter still significantly outperforms randomly choosing either predicate. Accuracies were 0.997 and 0.995 for Dynamic Filter and random and not significantly different.

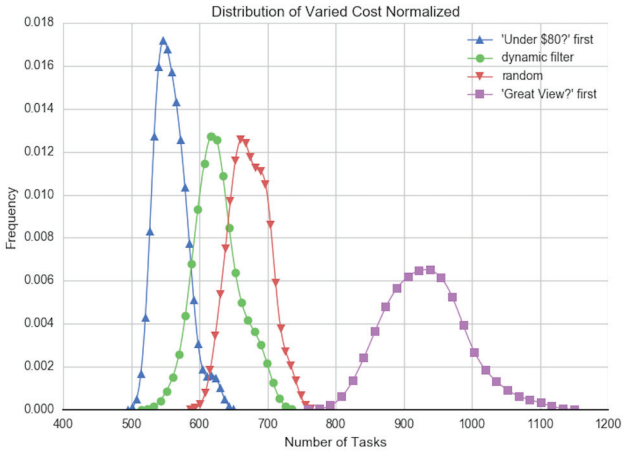

Figure 3: Distributions of the number of tasks for the various algorithms for varied costs.

Table 3 summarizes the comparison between the algorithms for the first two configurations (the configurations from Tables 1 and 2). The table shows how many more tasks than optimal were required for each algorithm, calculated as the average number of tasks for each algorithm divided by the average number of tasks for the optimal algorithm.

Hotel Predicates: Varying Selectivity, Low Cost (Gym?, Under $80?)	
Dynamic Filter	x1.10
Random	x1.25
Worst Case	x1.63

Hotel Predicates: More Selective, Varying Cost (Under $80?, Great views?)	
Dynamic Filter	x1.12
Random	x1.21
Worst Case	x1.67

Table 3: A table of the multiplier of each algorithm in comparison to the optimal case.

Low selectivity We also investigated situations where the order of the predicates has minimal effect on the number of tasks per simulation. This would occur if items typically satisfy both predicates, and thus the potential for savings is reduced. An example of this is the following configuration; the two predicates (shown in Table 4) have varying costs and low selectivity (i.e., many items satisfy the predicates).

Predicate	Selectivity	Cost
Does this restaurant serve drinks for those under 21?	100%	8.6 votes
Does this restaurant have more than 20 menu items?	80%	5.5 votes

Table 4: Low selectivity configuration.

As expected, when running the simulation, the number of tasks for each algorithm are close; the mean number of tasks for all four algorithms were not statistically significant (the t-values for Dynamic Filter vs. the optimal case, random algorithm, and worst case were: 0.8504, 1.5140, and 1.6902; p-values were 0.3956, 0.1308, and 0.0918). The multiplier differences from the optimal case between the three algorithms were both small and within 0.01 of each other: $x1.01$, $x1.02$, and $x1.03$ for Dynamic Filter, random, and worst case, respectively. Accuracies were 0.852 and 0.862 for Dynamic Filter and random and not significantly different.

While the multiplier differences still follow the same trend of Dynamic Filter outperforming both the random and worst case algorithms, the difference between these algorithms is not enough to make a significant claim. Because both predicates have low selectivities, they often return true for items passed to them by the filter. Thus, no matter which predicate the item is routed to first, the item will often need to be routed to the other predicate as well. As a result, the number of tasks that can be avoided through the ordering of the predicates is minimal.

Results using three predicates Queries with more than two predicates further illustrate the observations made previously. Table 5 shows the average number of tasks required for two hotel queries involving three predicates, labeled Q1 and Q2. Q1 uses the three distinct predicates from Tables 1 and 2. Q2 has one different predicate asking about safety; the safety predicate has low selectivity and medium cost. The advantage of Dynamic Filter increases with more predicates as a low-cost predicate ordering will avoid even more work than an algorithm that does not prioritize cheap and selective predicates. For both queries the differences are significant (Q1's t-value was 19.9 and p-value was < 0.0004; Q2's t-value was 13.7 and p-value was < 0.0002).

Query with 3 Predicates	Dynamic Filter	Random
(Q1) Gym?; Under $80?; Views?	680.36	841.44
(Q2) Gym?; Safe?; Views?	1290	1479.2

Table 5: Average number of tasks for two queries. No items satisfy all for Q1; 34.4% items satisfy all for Q2.

Evaluating the sliding window We now discuss how the "sliding window" adapts to changes in predicate properties.

Setup To test Dynamic Filter with windowing enabled, we generated synthetic data based on the crowdsourced data. Each vote was cast as a random sample based on the selectivity of the predicate, with noise added to influence cost. This noise factor was simply the probability of choosing the value that the predicate leans towards. For example, a noise level of 0.5 would yield a very expensive predicate while a predicate of 1.0 would be the cheapest possible predicate.

In the synthetic data, there are two predicates and 100 items. Predicates P0 and P1 both have selectivity of 0.1 and a noise level of 0.9 and 0.6, respectively. The predicates switch noise level when the simulation has completed 200 tasks. Ideally, an adaptive algorithm would first favor P0 first, and adapt to choose P1 first after the switch. We tested five algorithms: Dynamic Filter with sliding window, Dynamic Filter without the windowing, random, and a best and worst case scenario. In this configuration, we chose to make the switch based on cost, not selectivity, to simulate cost changes in an environment where workers come and go. The ticket lifetime was ten tasks.

Results In addition to Dynamic Filter with and without windowing, a few baseline algorithms were implemented. The "optimal switch" algorithm chooses P0 first and switches to P1 first after 200 tasks. This way, the optimal always chooses the cheaper predicate to first send items. The "worst switch" algorithm is the opposite: it always chooses the more expensive predicate first. The random algorithm was also implemented to test if our algorithms were better than simply randomly choosing either predicate first.

Table 6 shows the multiplier of each algorithm's average number of tasks to complete the query in comparison to the optimal switch algorithm. We can see that Dynamic Filter with sliding window outperforms Dynamic Filter without, and is close in performance to the optimal switch algorithm.

Synthetic Cost Switch at 200 tasks	
Dynamic Filter w/ Window	x1.02
Dynamic Filter w/o Window	x1.11
Random	x1.16
Worst Case	x1.69

Table 6: Algorithm multipliers compared to optimal.

We also evaluate how Dynamic Filter without the sliding window reacts to the fluctuation to ensure that Dynamic Filter alone still results in fewer tasks than randomly choosing either predicate. The average number of tasks for Dynamic Filter and the random algorithms were 846.84 and 883.46 respectively. We found that the mean number of tasks were significantly different (the standard deviations of Dynamic Filter without windowing and the random algorithm were 54.55 and 45.97 respectively; the t-value was 3.63 and the p-value was < 0.0005). Thus, Dynamic Filter without windowing can still recover from the cost switch at 200 tasks to outperform the random algorithm, but not as efficiently as Dynamic Filter with windowing enabled.

Conclusion and Future Work

In this paper we propose an adaptive approach to processing a crowd-based filter query that involves multiple predicates for which predicate selectivity and cost information is unknown at query time. By choosing an order of evaluating predicates that delays expensive predicates, we can yield substantial savings in the total query cost. These savings are highlighted when comparing the results for Dynamic Filter versus the worst case static ordering; without knowledge of an efficient ordering, a chosen fixed ordering can be quite expensive. The savings can be even more dramatic as the number of predicates increase. Using an adaptive approach in Dynamic Filter, we are able to learn a low-cost ordering for evaluating the predicates, even as the runtime environment is changing.

Future work includes dynamically ordering other query operations in addition to filters, such as the JOIN operation that combines items from multiple sources. We also want to investigate user interfaces for specifying crowd-based queries and extracting predicates queries expressed in natural language.

By requiring less a priori information, we believe adaptive query processing, and specifically the Dynamic Filter algorithm, will increase the utility of crowd-powered query processing systems for ad hoc queries.

Acknowledgments

We thank the reviewers and our colleagues at Harvey Mudd College for their helpful feedback. We also thank the crowd workers for their participation in the project. This work was supported in part by the National Science Foundation under Grant No. 1359170 and Grant No. 1657259, as well as a gift from Citadel LLC.

References

Avnur, R., and Hellerstein, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 261–272.

Babu, S.; Motwani, R.; Munagala, K.; Nishizawa, I.; and Widom, J. 2004. Adaptive ordering of pipelined stream filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 407–418.

Basu Roy, S.; Lykourentzou, I.; Thirumuruganathan, S.; Amer-Yahia, S.; and Das, G. 2015. Task assignment optimization in knowledge-intensive crowdsourcing. *The VLDB Journal* 24(4):467–491.

Berry, D. A., and Fristedt, B. 1985. *Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability)*. Springer.

Chaudhuri, S. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*.

Das Sarma, A.; Parameswaran, A.; Garcia-Molina, H.; and Halevey, A. 2014. Crowd-powered find algorithms. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)*.

Deshpande, A.; Ives, Z.; and Raman, V. 2007. Adaptive query processing. *Foundations and Trends in Databases* 1(1):1–140.

Difallah, D. E.; Demartini, G.; and Cudré-Mauroux, P. 2016. Scheduling human intelligence tasks in multi-tenant crowd-powered systems. In *Proceedings of the International Conference on World Wide Web (WWW)*, 855–865.

Fan, J.; Zhang, M.; Kok, S.; Lu, M.; and Ooi, B. C. 2015. Crowdop: Query optimization for declarative crowdsourcing systems. *IEEE Transactions on Knowledge and Data Engineering* 27(8):2078–2092.

Franklin, M. J.; Kossmann, D.; Kraska, T.; Ramesh, S.; and Xin, R. 2011. CrowdDB: Answering Queries with Crowdsourcing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Heer, J., and Bostock, M. 2010. Crowdsourcing graphical perception: using mechanical turk to assess visualization design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

Hellerstein, J. M., and Stonebraker, M. 1993. Predicate migration: optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Ho, C.-J., and Vaughan, J. W. 2012. Online task assignment in crowdsourcing markets. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 45–51.

Ipeirotis, P. G.; Provost, F.; and Wang, J. 2010. Quality management on Amazon Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*.

Karger, D. R.; Oh, S.; and Shah, D. 2014. Budget-optimal task allocation for reliable crowdsourcing systems. *Operations Research* 62(1):1–24.

Marcus, A.; Wu, E.; Karger, D.; Madden, S.; and Miller, R. 2011a. Human-powered sorts and joins. *Proceedings of the VLDB Endowment* 5(1):13–24.

Marcus, A.; Wu, E.; Madden, S.; and Miller, R. 2011b. Crowdsourced Databases: Query Processing with People. In *Proceedings of the Conference on Innovative Data Systems Research*.

Marcus, A.; Karger, D.; Madden, S.; Miller, R.; and Oh, S. 2012. Counting with the crowd. *Proceedings of the VLDB Endowment* 6(2):109–120.

Nunia, V.; Kakadiya, B.; Hota, C.; and Rajarajan, M. 2013. Adaptive task scheduling in service oriented crowd using slurm. In *International Conference on Distributed Computing and Internet Technology*, 373–385. Springer.

Parameswaran, A. G.; Garcia-Molina, H.; Park, H.; Polyzotis, N.; Ramesh, A.; and Widom, J. 2012a. Crowdscreen: Algorithms for filtering data with humans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 361–372.

Parameswaran, A. G.; Park, H.; Garcia-Molina, H.; Polyzotis, N.; and Widom, J. 2012b. Deco: declarative crowdsourc-

ing. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*.

Parameswaran, A.; Boyd, S.; Garcia-Molina, H.; Gupta, A.; Polyzotis, N.; and Widom, J. 2014. Optimal crowd-powered rating and filtering algorithms. *Proceedings of the VLDB Endowment* 7(9):685–696.

Park, H., and Widom, J. 2013. Query optimization over crowdsourced data. *Proceedings of the VLDB Endowment* 6(10):781–792.

Quinn, A. J., and Bederson, B. B. 2014. Asksheet: Efficient human computation for decision making with spreadsheets. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, 1456–1466.

Rajan, V.; Bhattacharya, S.; Celis, L. E.; Chander, D.; Dasgupta, K.; and Karanam, S. 2013. Crowdcontrol: An online learning approach for optimal task scheduling in a dynamic crowd platform. In *Proceedings of ICML Workshop: Machine Learning Meets Crowdsourcing*.

Sheng, V. S.; Provost, F.; and Ipeirotis, P. G. 2008. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 614–622.

Waldspurger, C. A., and Weihl, W. E. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*.

Yuen, M.-C.; King, I.; and Leung, K.-S. 2011. Task matching in crowdsourcing. In *Internet of Things, 4th International Conference on Cyber, Physical and Social Computing*, 409–412. IEEE.