Concurrency and Joins in Dynamic Filter

Beth Trushkowsky
Harvey Mudd College
P.O. Box 1212
Claremont, California 43017-6221
beth@cs.hmc.edu

Cienn Givens
Harvey Mudd College
340 E. Foothill Blvd
Claremont, CA 91711
cgivens@g.hmc.edu

Israel Jones
Harvey Mudd College
340 E. Foothill Blvd
Claremont, CA 91711
ijones@g.hmc.edu

Jasmine Seo
Harvey Mudd College
340 E. Foothill Blvd
Claremont, CA 91711
jseo@g.hmc.edu

Anna Serbent
Harvey Mudd College
340 E. Foothill Blvd
Claremont, CA 91711
aserbent@g.hmc.edu

ABSTRACT

Crowdsourcing is a powerful tool for solving some problems, especially those that are subjective. Therefore, using the crowd to help process database queries can significantly expand the type of questions users can ask. The Dynamic Filter algorithm, defined and discussed in previous work, is an adaptive query processor made to leverage the crowd to evaluate filters for arbitrary queries. We expand upon Dynamic Filter to better address the necessity of concurrent workers and to implement crowd-powered joins. By exploring multi-armed bandit problems with delay, implementing batch assignment, and altering the existing ticketing system, we have been able to significantly improve the performance of the algorithm in terms of worker cost and time in a concurrent setting. Simultaneously, we have provided a framework for the implementation of joins in Dynamic Filter, including our expected approaches to joins in the algorithm. Experimental data support our changes to the concurrent algorithm and our choices on what operations to consider when using a join.

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Beth Trushkowsky, Cienn Givens, Israel Jones, Jasmine Seo, and Anna Serbent. 2016. Concurrency and Joins in Dynamic Filter. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 17 pages.
DOI: 10.475/123_4

1 INTRODUCTION

There are some questions that are difficult for a computer to answer, such as questions that are subjective or that require visual or semantic interpretation. Fortunately, humans are good at answering these questions, and we can take advantage of that talent through online crowdsourcing. Online crowdsourcing provides a great medium for large scale human computation, and therefore is perfect for answering many of these types of questions. Human error can also be minimized by asking multiple people each question, to ensure consensus, a common crowdsourcing practice [8][4]. Using a relational database, this data can be processed and used to ask the crowd even more questions. This way, we can use the crowd to break apart, piece by piece, larger, more complex questions. In doing so, we hope to reap the benefits of both machine scalability and human problem solving. This is the aim of Dynamic Filter, an algorithm that implicitly measures the difficulty and usefulness of various questions in respect to a single problem, all while processing responses from the crowd.

In this paper, we describe Dynamic Filter, some of its limitations, and a few proposed changes as presented in earlier work[5]. Before our changes, the core algorithm primarily simulated the case wherein the algorithm gave a worker in the crowd a question about one item, received an answer immediately, and continued. In reality, though, asking questions serially (one by one) can be extremely slow. Therefore, we want the algorithm to allow multiple workers to evaluate questions concurrently. However, this also means that the algorithm will need to make decisions before receiving feedback on all previous choices. One of our main goals is to optimize Dynamic Filter with this trade off in mind. We also intend to integrate joins, a type of database operation that Dynamic Filter does not currently implement, so that it asks easier, more time-effective questions to the crowd. Crowd-powered joins could significantly improve Dynamic Filter's function in the case of questions involving multiple objects, as they allow us to store and use information about objects that we can use to answer similar questions later, thus avoiding repeated work.

2 BACKGROUND

It is important to know what a database is in order to fully grasp Dynamic Filter and the work that we do to improve it. In the most general sense, a database is a store of data, something that holds information that many people need to access or that is too large to reasonably fit on a computer. In order to access and manage data appropriately, we usually need a database management system (DBMS) that serves as an intermediate between our computer and the raw data. The DBMS has the responsibility of making our access to the data as pleasant as possible, including increasing the ease of retrieval and ensuring that concurrent access to data is handled appropriately. Most DBMSs are relational, meaning they represent data in tables, where each column represents an attribute and each row represents an item. As these are the most common type of DBMS, these are the kind we focus on in Dynamic Filter. With this understanding of what a database is, we can examine the earlier version of the Dynamic Filter algorithm and then delve into some background on the two foci of this research: concurrency and joins.

2.1 Dynamic Filter Algorithm

A database query returns a set of items that pass a user defined series of database operations, such as filters and sorts. Filters eliminate objects from the set if they do not meet a certain condition. The Dynamic Filter algorithm [5] is aimed at optimizing the application of multiple filters by ordering them in a way which reduces the amount of work required to complete the query. In our system, a filter consists of a number of predicates, which are yes-or-no questions. For instance, for a set of hotels, the items are individual hotels, such as the Double Tree Hotel, and the predicates are hotel-related questions, like “Does this hotel have a gym?”. When a predicate is applied to an item, we refer to that as an Item-Predicate (IP) pair, such as “Does the Double Tree Hotel have a gym?”. We integrate online crowdsourcing into a database query by asking such questions to human workers on Amazon's Mechanical Turk (henceforth AMT). This is a platform that allows requesters, like us, to post Human Intelligence Tasks (HITs) for workers to complete, for small monetary compensation.

Each item has to be evaluated as true for all predicates to pass a filter. If an item fails one of the predicates, then it fails the entire filter, and no longer needs to be evaluated by other predicates. To take advantage of this fact, our system should place more selective predicates, which evaluate to false for many items, before less selective predicates. This way, it can eliminate more items early on and reduce the number of tasks to complete the filter. In addition, when predicates have varying costs, due to the difficulty or ambiguity of the question, placing cheaper predicates before expensive ones can reduce the total cost, in terms of both time and money. This is because when cheap predicates eliminate items, these items won't need to be evaluated for more expensive predicates. Previous research has shown methods of ordering predicates when their selectivities and cost are known [1].

However, in a crowdsourced database, the selectivity and cost of the predicate is unknown until the end of a query, making it difficult for the requester to order the predicates beforehand. The Dynamic Filter estimates the selectivity and cost of the predicates as it gathers responses from HITs and gives preference to selective and cheap predicates when deciding the next HITs. It does this using the concepts of *lottery tickets* and *predicate queues.* To pick a predicate to route an item to, the algorithm runs a random lottery. Initially, each predicate holds a single lottery ticket. When an item is routed to a predicate, that predicate receives a ticket, and if that IP pair is evaluated as true, that ticket is removed, penalizing predicates that do not fail items, and remove them from further processing. Thus, selective predicates, which tend to fail items, end up with more tickets than non-selective predicates and have more items routed to them. Each predicate also has a queue that holds a number of items that are waiting to be or are being evaluated by the crowd. Items are not assigned to predicates with full queues. Expensive predicates require more tasks and time to evaluate items, so they tend to have a full queue longer than cheap predicates, which reduces the number of tasks routed to expensive predicates. Dynamic Filter chooses tasks based on the behavior of predicates, rewarding those that have failed many items and those that completed items quickly.

2.2 Concurrency

Our previous work has often compared Dynamic Filter to the multi-armed bandit (MAB) problem. In MAB problems, an agent is given a slot machine with multiple arms, each of which give rewards based on unknown distributions, and often a time horizon after which no more pulls can be made, and then has to make a strategy for pulling arms. The goal of algorithms based on MAB is generally to minimize non-ideal choices, or regret, by using strategies that balance exploration and exploitation. Exploration refers to pulling different arms to get some estimation of how different arms pay out, and exploitation refers to pulling the arms that you estimate to have the highest payout. In the traditional (serial) setup, the agent knows the reward from their action immediately. However, we are interested in the delayed case, wherein the payout of an action is only returned after some time has passed [4] [3] [6]. The algorithm will generally make better decisions when it has feedback from its previous choices, but due to the horizon, it may not be worthwhile to wait for a response before making a new pull.

There are significant similarities between MAB with delay and the concurrent scenario for Dynamic Filter. With Dynamic Filter, choose between different predicates, each of which have an unknown selectivity. This operates as a “reward” in that different arms have different probabilities to return false, and consequently reduce the number of items and tasks that a query needs to complete. In the concurrent case, we also have delay, provided by workers, who have to spend time evaluating an IP pair before returning a response.

However, MAB's prioritization of reward in respect to a horizon is different than our system. The goal of our algorithm is to exhaust all items, and the removal of items can happen from a single IP pair associated with an item evaluating as false, or all IP pairs associated with an item evaluating as true, both based on ground truths. This means that, although we model reward as the failure of items, any ordering of predicates should always pass and fail the same sets of items. These conditions can't translate directly to MAB's concepts of reward or horizon, but we can still use algorithms for MAB problems as a guide. We discuss some of the particular algorithms that interest us in 3.1.1.

2.3 Databases and Joins

Before our recent adjustments, Dynamic Filter was only made to optimize for selections (which we usually call filters), in which a database would comb through every item in a table and select just those items that satisfy a given condition. Classical databases, however, are not usually confined to selections. often, databases use an operation called a join, in which two tables are combined based on a given condition called the join condition. To clarify, say two tables in a database have columns that represent the same data. We could request that the database join the two tables on the condition that those columns have the same value. The database could then go through every pair of rows $(r1, r2)$ where $r1$ is in one table and $r2$ is in the other table. If both had the same value in the column we are interested in, the database would add a new row to a resulting table that included all of the data from $r1$ and $r2$. This kind of join, in which we join based on equality within some column, is called an equijoin.

Joins are a useful tool for manipulating databases. We often want to find meaningful pairs of items or consolidate data about a single item, but this information is often stored in separate tables. To fix this, we need to combine the tables, but most of the time we want only a portion of the possible pairs. Thus we use a join, cutting the results down to only what we need. Often, we can evaluate a join without evaluating the entire cross product of the two tables, which is convenient because it can greatly decrease the time it takes to evaluate a join.

Our approach to joins will need to handle a few uncommon issues, both because Dynamic Filter is primarily meant to filter items and because we use crowdsourcing. For our purposes, joins are always crowd-powered, and are usually embedded in complicated filters, wherein we want to know which items in the database are related to other items that themselves satisfy some condition (e.g. what hotels have metro stations within a mile that have a direct line downtown). These involve taking each item h from the database and examining it in relation to other items $\{x_1, x_2, x_3, ...\}$. If the crowd agrees that a pair (h, x_i) satisfies the join condition, it is a match. Because this requires that we evaluate every possible pair, crowd-based joins can be incredibly cost-intensive [1]. In the worst case, evaluating a join is equivalent to running a filter on the cross product of two sets. Thus for an arbitrary cost c_1 of evaluating the condition for any pair, we know the total cost of joining the two lists H and M is at most that cost for every item in the cartesian product,

$$|H \times M| * c_1. \tag{1}$$

One might wonder why we use such an expensive operation. We have already explained that the joins we use are embedded in filters, and we already have machinery within Dynamic Filter to handle filters effectively. Put simply, we want to make the work easier for the crowd and at the same time avoid repeated work. It can be hard to figure out whether one item is related to another item in a potentially huge list of other items before also finding whether or not that other item fulfills yet another condition. We intend to make this easier by splitting joins into multiple steps, each of which should be relatively easy. Once we do so, we expect that workers will be able to complete the entire join more quickly, as they no longer need to keep switching between steps internally. Instead, they can do the same step on many items until they are all complete, and then move on to the next step. In addition, once we have information about some items, particularly items not already stored in the database, we can store that information and access it without asking the crowd. This can hypothetically reduce costs significantly, as we reduce the amount of processing and re-processing each worker needs to do.

[1]It should be noted here that, in the context of joins, cost can refer to either time or monetary value. We use the term arbitrarily for joins, as we intend to pay workers fairly, meaning more difficult tasks will take more time and cost more money. Because we expect the two to be roughly linearly correlated, we do not distinguish between them past this point.

Still, the cost of joins may be high for Dynamic Filter, and it would benefit us to make them as cost-effective as possible. One approach geared towards reducing crowd-based join costs is to use prejoin filters [7] [8]. Using this method, each item in both lists is placed into categories related to the join condition. These categories must be applicable to all items from both lists to be effective. Once the data are categorized, the categories of any two items can be compared before we ask the crowd to evaluate the join condition on them. If we have chosen categories that are simple and deeply connected to the join condition, we will be able to tell using only a computer that some pairs cannot match by the join condition based purely on the categories of each item. This can allow us to decrease the number of tuples that need to be processed by the crowd significantly, effectively performing some of the joins for free. When a prejoin filter is applied, the total cost of joining the two lists becomes

$$(|H| + |M|) * c_2 + f * |H \times M| * c_1, \tag{2}$$

where c_2 is the cost of evaluating the prejoin filter on one item and f is the fraction of pairs whose categories match in the prejoin filter. [2] Note here that the potential cost reduction is significant. With a selective prejoin filter, we can easily bring the cost to only a small fraction of what it would be otherwise.

As an example, say we have two sets of creatures and we want to know which of those are in the same genus. We could use a prejoin filter to ask the crowd to identify the subjects by kingdom. After doing so, the machine can check these results before we try to match a pair of creatures, and any without a matching kingdom can automatically be eliminated, as we can assume two creatures are not in the same genus if the crowd agrees they are of different kingdoms.

Multiple sources have demonstrated the effectiveness of prejoin filters. Mitsuishi et al. write about the use of human-powered prejoin filters for cases where computer generated prejoin filters are not applicable [8]. In this research they show the theoretical results as well as preliminary experimental results indicating the benefits of using human-powered prejoin filters. They consider cases where the results of the prejoin filter do not necessarily match for tuples that should be accepted by the join. For example, say our two lists are of metro stations and hotels and our join condition is that they are within 5 miles of each other. If our prejoin filter is the location of the metro or hotel, then a metro may be tagged at California while a hotel may be tagged as Los Angeles. In this case the two items are not necessarily an invalid pairing. Mitsuishi et al. handle this by introducing the idea of compatible and incompatible types to come up with property equivalence classes. In this way, they ensure a metro station whose location was "California" is marked as incompatible with a hotel in "Arizona," but not incompatible with a hotel in "Los Angeles". A tuple can only be successfully eliminated if the two tags are marked as conflicting by a worker, otherwise they continue and are joined by the crowd. This is a powerful functionality to have, and although we do not implement it right now, we hope to examine it in more detail in the future.

[2]In all cases f is less than 1 and approaches 0 with more selective prejoin filters.

3 APPROACH

In this section, we will explain our approach to our two issues: optimizing Dynamic Filter for concurrent workers and implementing the join operation in Dynamic Filter. The first subsection will detail how our revisions to Dynamic Filter attempts to assign tasks in a concurrent setting, and the second will explain how we introduce joins to better address problems that involve multiple entities in Dynamic Filter.

3.1 Concurrency

One of the major limitations of Dynamic Filter is that until now it has mostly been tested in the case with serial workers under the assumption that workers respond immediately. However, the algorithm is designed with concurrent workers in mind, and our intuition is that allowing multiple workers at once can speed up queries significantly. Therefore, in this section, we discuss existing structures of the algorithm we want to test in the concurrent case, as well as changes to the algorithm that we believe could improve concurrent performance.

3.1.1 Connections to the Multi-Armed Bandit Problem with Delay. Earlier, we discussed the similarity between our concurrent problem and the MAB problem with delay. We are particularly interested in black-box algorithms for the MAB problem with delay: algorithms that take a serial algorithm, and then used it, unchanged, inside of a larger concurrent algorithm. In the end, we were interested in Mandel et al.'s General Queue-Based Bandit Algorithm [6], which is based on Queued Partial Monitoring with Delays (QPM-D), described in Joulani, György, and Szepesvári's paper [4]. This algorithm uses another algorithm, the Stochastic Delay Bandit (SDB) algorithm [6]. We also took inspiration from c-delayed block policies, a concept defined in Guha, Munagala and Pal's paper [3].

In the General Queue-Based Bandit Algorithm, an instance of the serial algorithm, BASE, is used to generate a probability distribution for pulling each arm. This distribution is then modified by SDB, an algorithm that modifies the distribution based on various concurrency-specific variables, such as the maximum delay we've seen, the number of pulls that have not yet returned feedback for each arm (via queues), and other constants. Pulls are then made based on this altered probability distribution. The BASE algorithm is updated, as though results were received serially, whenever feedback is returned. Although the overall algorithm uses SDB to modify their BASE distribution, the algorithm is designed such that any function that modifies the serial distribution can be plugged in.

The way that the General Queue-Based Bandit algorithm works is in fact very similar to how Dynamic Filter is already implemented. In our algorithm, each predicate has a queue, limiting how many items can be active for a single predicate at a time, and each time step, instead of only pulling values from the serial algorithm's probability distribution, many other parameters dealing with concurrency generate a new distribution, based on the serial distribution. This is similar to how SDB is used to modify the probability distribution from BASE. In this way, the General Queue-Based Bandit algorithm didn't influence us to change Dynamic Filter, but validated our existing implementation, by showing that strong solutions to similar problems are similar to our solution.

Guha et al.'s discussion of c-delayed block policies motivated our changes to the algorithm a bit more concretely. On AMT, it is normal for HITs to be sent out in large batches. Although this is not necessary using the programmatic API, sending tasks out in batches would require the back-end to interact with the website less often, which is generally positive for performance, and a flexible enough batch assignment system could hopefully imitate a system that focuses on constant assignment. C-delayed block policies are strategies for deciding on many arms to pull in MAB at once (a block), then waiting until all pulls are made in a block before sending out another. Guha et al. provide a method which uses the distance to the horizon to calculate block sizes that bound a solution's regret. Although we route workers to IP pairs dynamically as they arrive, and never all at once on the backend, we do have to think about how many HITs we post on AMT, and when we post them, in which case the logic behind dynamically sizing blocks to improve performance is one we can borrow for batch assignment. It also tells us that there may be an algorithmic advantage to including some of the implicit delay that releasing tasks in batches causes. With this in mind, we attempt to create a logical stage-based batching system.

3.1.2 The Flow of the Concurrency Algorithm. Minimizing the number of active tasks allows Dynamic Filter to receive more feedback before it assigns new tasks, and therefore will generally assign more selective predicates, meaning it will take fewer tasks to complete the query. However, limiting the number of concurrent tasks increases the time required to complete a query. To try to find a balance between the speed and human cost of a query, we explore various methods of controlling the rate at which we make hits available to workers.

One approach is to maintain a constant number of active tasks by issuing a task immediately after an active task is completed. For instance, with 40 active tasks, 40 HITs are posted onto AMT. Every time a worker returns a response, a new HIT is posted, such that the number of active or available HITs is 40, until there are fewer than 40 tasks left to assign. Although this method is tested in our experiments, as previously discussed, constant assignment is not practical on AMT. Another method, as discussed in the previous section, is batch assignment. The current flow of the Dynamic Filter algorithm, with batch assignment, is shown in Algorithm 1. In batch assignment, many HITs are posted initially. We call the initial number of HITs we post the HIT refill cap. Then, when all HITs from this batch are assigned and the number of active tasks reaches a lower limit, which we call the HIT refill limit, we post HITs until the total number of tasks out matches the HIT refill cap. We experimented with different pairs of refill limits and caps: 10-40, 150-200, and 350-450. As an example, for 10-40 batch assignment, if there are only 8 active tasks, and all tasks from the previous batch have been assigned, a new batch of 32 HITs will be sent out to reach the refill cap of 40.

A variation to the batch method, staged batch assignment, combines these three settings, accounting for the gradual decrease in tasks to complete. As described in table 1, during the beginning stage when there are more than 50 incomplete IP pairs and thus more tasks to be worked on, the refill cap is 450, but as the query continues and there are fewer IP pairs to finish, the refill limit and cap each decrease to avoid assigning an excessive number of tasks.

Algorithm 1: Concurrent task assignment

```
Input: Set of items $D$ and set of predicates $Pred$
Let refillCap, refillLimit be the current batch refill settings
openHits = 0 ; /* Number of available HITs on AMT */
timeStep = 0;
Map activeWorkers ; /* map worker → (item,pred) */
Map ipPairs ; /* map pred. → items */
foreach predicate $p \in Pred$ do ipPairs($p$) ← D;
repeat
    foreach worker w ∈ activeWorkers do
        if w is done then
            (item,pred)← activeWorkers(w);
            ipPairStatus ← aggregateVotes(item,pred);
            if ipPairStatus is a consensus for false then
                foreach predicate $p \in Pred$ do
                    ipPairs($p$) ← ipPairs($p$) - item
                end
            else if ipPairStatus is a consensus for true then
                ipPairs(pred) ← ipPairs(pred-item)
            end
            activeWorkers($w$) ← activeWorkers($w$) - (item,pred)
    end
    if openHits = 0 and len(activeWorkers) < refillLimit then
        openHits = refillCap- len(activeWorkers);
        /* This number of HITs are posted to AMT */
    workers ← workersArrive();
    foreach worker w ∈ workers do
        item,pred ← chooseNextTask(w,ipPairs);
        w.startTask(item,pred);
        openHits-= 1;
    end
    ++timeStep;
until no item-predicate pairs remain in ipPairs and no workers remain in activeWorkers;
```

Table 1: Staged Batch Assignment

Stage	Num. Incomplete IPs	Refill Limit	Refill Cap
Beginning	more than 50	350	450
Middle	10-50	150	200
End	less than 10	10	40

Each of these ranges were determined by either experimental data or mathematical logic. The beginning range generally guarantees that we will also have the maximum number of possible active workers, based on experimental data with how often workers arrive and the length of tasks. For the middle stage, the maximum number of votes required to reach consensus in our current system is 21. So with a lower bound of 10 incomplete IP pairs, the maximum number of votes the active IP pairs will need is 210. Therefore this middle range will still allow many workers concurrently, but should never try to assign more tasks than are necessary. For the ending stage, research shows that workers on AMT generally are more likely to accept a task from a group with many available HITs, as discussed in the CrowdDB paper by Franklin et al. [2], so 40 was decided as a sufficiently large lower bound on HITs we put out, to ensure that we still attract workers.

We also considered the timing of the refill. While limit-based methods assign a batch of tasks when the number of active tasks hits a lower limit, the periodic method posts a batch of HITs after a fixed period, still assuming all HITs from the previous batch have been assigned. We used the total number of tasks and time required to complete a query to compare limit-based refill and period based refill, for several periods.

3.1.3 Rewarding Selective Predicates with Lottery Tickets. One of the first few concurrency adjustments we made to the Dynamic Filter algorithm was changing the ticket awarding system. Avnur and Hellerstein's Eddy system for dynamic query processing assigns tickets to database operations when an item is routed to it, and removed it if the item passes the operation [1], thus giving bias over time to operations that remove items. Bias here refers to how many tickets and tasks a certain predicate receives. Ideally, we want more bias for predicates that are highly selective, and as much as possible to not route items to low-selectivity predicates until highly selective predicates are exhausted. However, in the beginning of a query, the original algorithm might be rewarding less selective predicates if items were routed to them before any information is collected. This is not an issue in the serial case, because feedback is received instantaneously. But in the concurrent case, due to the stochastic nature of the assignment and feedback delays, the first two or three predicates selected by the lottery have a high chance of having their ticket numbers spike, before any feedback is received, and tickets removed. Instead of awarding tickets to predicates when an item is routed to them, and potentially removing them later if the item does not fail, we now only award them when an IP pair fails. This ticketing method seems more intuitive to us, and preliminary tests indicate that it performs similarly to or better than the original method.

Taking inspiration from MAB research, we test an adaptive batching system, and make adjustments to the existing ticketing system. We will describe the experimental results for these changes in section 4.1.

3.2 Joins

Since we will be implementing our own joins, we have full control over how the join is built and when the pairs are filtered. This can lead to what seems like a huge number of processing options. So we first will define some key terms. Then we will walk through an example of how one join can have many approaches, identify the relevant approaches (we will call these "paths") to explore and discuss one way to go about choosing which path to use.

3.2.1 Join terminology. As we will see, completing an entire join can require workers to do several different types of work. Thus, from this point forward, we will refer to a full crowd-based join as a "join process" within our algorithm. By this we simply mean to emphasize that there are many parts to consider. Let us take a closer look at what will happen when we execute this join process.

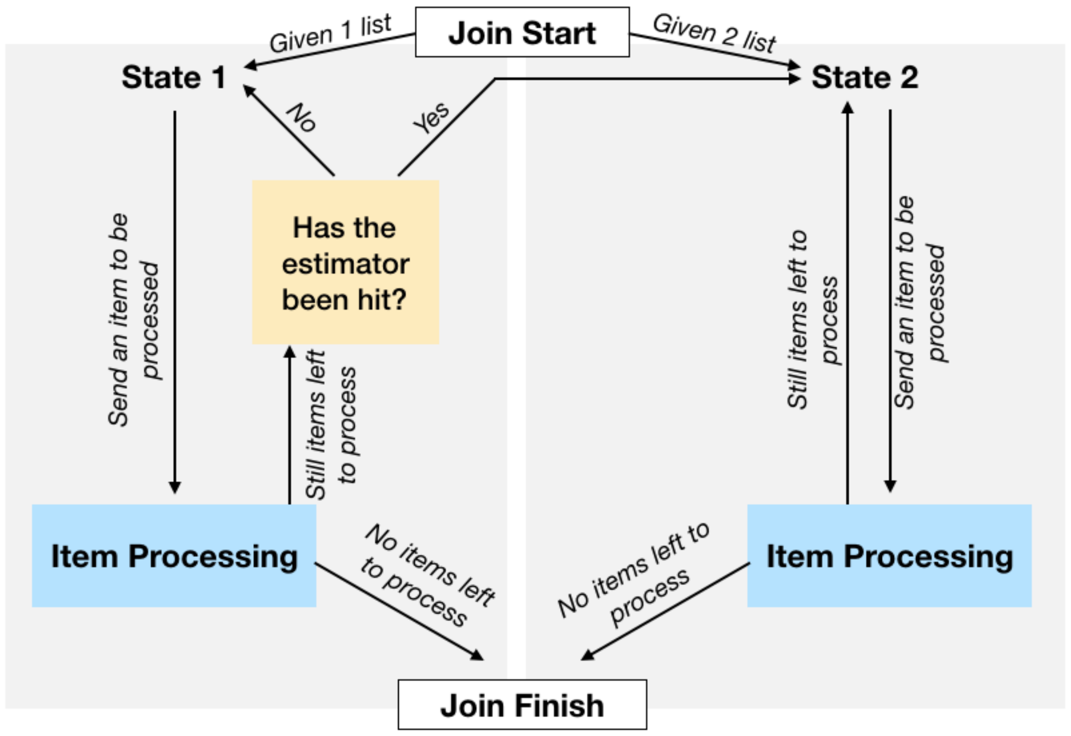


Figure 1: Two processing states in a join process

For a particular question, or predicate, we know that we can ask the crowd to treat that predicate as a filter –which is what Dynamic Filter has done until our changes. Depending on the structure of the predicate, however, it might be in our best interest to evaluate that predicate using a join process. We say that a predicate is "joinable" if it requires Dynamic Filter to use items from a secondary (usually crowd-generated) list of items to create pairs that satisfy a join condition. While we are in the context of a join process, we will call the joinable predicate the "primary predicate", because it is in reference to the primary (given) list. Primary predicates can be broken down into two components.

The first component is the join condition, which we have already discussed. The second component is what we call the secondary predicate. The secondary predicate is a simple filter but unlike the other predicates we have seen, it applies to the secondary list rather than the primary list. To help consolidate these definitions, we will examine how they manifest in an example.

3.2.2 Approaching a join in Dynamic Filter. Say we ask the crowd to filter a list of restaurants by the question "Does the restaurant serve any American dishes with potatoes?". In this case, we know that this predicate is joinable because it requires workers to come up with American dishes (the secondary items) and decide if they think the restaurant serves that dish (the join condition). In the join process, this question is the primary predicate. The requirement on the dishes (that they contain potatoes) is the secondary predicate, because it refers to the secondary items.

Given a restauraunt, it might be difficult for a worker to first think through American dishes with potatoes on their own, then decide if the restaurant has said dish. Instead, we anticipate that we can make the process more cost-efficient by providing a list of American dishes and asking them to determine if the restaurant serves any of them. Using this approach, we have the flexibility to assign the secondary predicate ("Does the dish have potatoes?") early in the process or later depending on which option reduces cost the most.

With flexibility also comes a large space of choices. If there are not many restaurants and each has only one or two dishes, we might supply the crowd with a restaurant name and ask them for dishes it serves, and then ask them to filter those by the secondary predicate. If the secondary predicate is costly and we already know what all of the dishes are, we might give workers dish-restaurant pairs and only apply the filter on pairs that the crowd confirms passes the join. One can imagine how different circumstances (different numbers of primary or secondary items, more costly filters, etc.) might lead to different optimal approaches. Before we can choose the best process to use to evaluate the join, we need to formulate the overall framework of our join process and identify all the paths we can follow to reach our final answer.

3.2.3 A framework for a join process. At the start of the join process all the algorithm has is a primary predicate and a primary list of items. Given that the primary predicate is joinable the algorithm breaks it into the secondary predicate and join condition. Since the algorithm has no information about the items of the secondary list at this point, its processing options are limited. One way to start the join process is by sending a primary list item out to the crowd and asking them to return the secondary-list items that match it by the join condition. This requires each worker to generate some secondary-list items on their own. After it receives matches for a particular primary item, the algorithm will ask the crowd if the secondary items pass the secondary predicate. If a secondary item passes, the algorithm recognizes the pairs that contain that secondary item as valid answers (these satisfied the join condition and the secondary predicate, thus satisfying the entire primary predicate). All primary list items in these valid answers "pass" the primary predicate, all other primary list items (those that do not have pairs having failed the join condition or those whose pairs' secondary items failed the secondary predicate) "fail" the primary predicate and are filtered out.

This is how the algorithm functions when it only has access to the primary list throughout the join process. In this state, the algorithm is restricted to only one approach (the one described above). If we are provided with two lists there are many approaches to the join process that we can take. Before we dive into just what those approaches, or paths, look like, let's explore how we might get from the one-list case to the two-list case. Suppose we are only provided with one list. While the algorithm runs, it is receiving secondary-list items. In addition to seeing if they pass the secondary predicate, the algorithm saves them and builds an internal copy of the secondary list.

In biology it is common to use sampling to estimate the number of species in an area, this is species enumeration. In previous research by Trushkowsky, Kraska, Franklin, and Sarkar at U.C. Berkeley, similar enumeration techniques were used in conjunction with crowd-sourcing to estimate when workers had enumerated everything in a particular list[9]. Applying this enumeration estimator to our algorithm, Dynamic Filter can detect when all of the secondary list has been enumerated and for the remainder of the join process use any of the paths that we could use if we were given both lists. Essentially, when this enumeration estimator is hit we can move to our two-list state. It is important that we do have the entire secondary list before we move to the two-list case, as our cost estimates (described in the next subsection) depend on the sizes of the two lists, and having only a portion of the list could greatly hamper the effectiveness of our algorithm.[3] Once we have, though, there are a few options we can consider.

3.2.4 *Identifying paths.*

We have identified five main paths listed in the flowchart in figure 2. In the process of constructing this diagram and identifying the possible join paths, we made some assumptions that helped us eliminate paths that would almost never be selected as the fastest path. One such assumption is that whenever it is possible for us to use a prejoin filter it is beneficial, an assumption we will address in section 4.2.1. This is evident in figure 2, where the join is never present without a preceeding prejoin filter.

[3]because our cost estimates are sensitive to the size of the secondary list, attempting to operate on only the parts we have received so far would likely skew the process towards options that are in fact much less effective, notably paths 3 and 5.

3.2.5 *Choosing between paths.*

Once we eliminate any paths that seem unlikely to be beneficial, we face the challenge of coming up with an effective way to choose which path is best in executing the join for a particular query. From figure 2, we see that the choice between the five identified paths comes into play only once we have identified all of the second list. In this case, there are still items from the first list that need to be processed but we now have two lists that we can exploit to reduce our costs. To take advantage of the most cost-effective path we use cost estimates that are dynamically updated during runtime. In Dynamic Filter, these costs are forward-looking estimates of how long the entire algorithm will run, a factor we explain more in section 4.2.6. Below, we have a naive estimate of the cost of each path, where Cost_i is the cost of Path i.

$$\text{Cost}_1 = C_{\text{sp}}(|L_2|) + C_{\text{pjf}}(|L_1| + |L_2|) + C_{\text{join}}|L_1||L_2|, \tag{3}$$

$$\text{Cost}_2 = C_{\text{pjf}}(|L_2| + |L_1|) + C_{\text{join}}|L_1||L_2| + C_{\text{sp}}L_2, \tag{4}$$

$$\text{Cost}_3 = (C_{\text{iw}} + N_{\text{match}}C_{\text{match}}\frac{|L_1|}{|L_2|}) * |L_2| + C_{\text{sp}}|L_2|, \tag{5}$$

$$\text{Cost}_4 = (C_{\text{iw}} + N_{\text{match}}C_{\text{match}})|L_1| + C_{\text{sp}}|L_2|, \tag{6}$$

$$\text{Cost}_5 = C_{\text{sp}} * |L_2| + (C_{\text{iw}} + N_{\text{match}}C_{\text{match}}\frac{|L_1|}{|L_2|}) * |L_2| \tag{7}$$

where

$$\begin{aligned}
L_1 &= \text{List 1 (primary list)},\\
L_2 &= \text{List 2 (secondary list)},\\
C_{\text{sp}} &= \text{Cost of evaluating the small predicate},\\
C_{\text{pjf}} &= \text{Cost of evaluating the PJF},\\
C_{\text{join}} &= \text{Cost of evaluating the join},\\
C_{\text{match}} &= \text{Cost of finding one match},\\
C_{\text{iw}} &= \text{Base cost of an item-wise join task},\\
N_{\text{match}} &= \text{Average number of matches}
\end{aligned}$$

For the sake of terminology, note that list 1 is the list we are filtering by the primary predicate, "PJF" should be read as "prejoin filter", and an item-wise join is a task in which we give an item from one list and ask for its matches.

These equations take into account the different costs in the order in which they arrive, all evaluated on the two full lists. However, we are still missing important information. If the equations above were correct, then paths 1 and 2 and paths 3 and 5 should cost the same, so there would be little reason to have both. This is because we have not yet accounted for the selectivity of each operation, which reduce the amount of work later operations have to do. In order to account for this, we introduce some more variables:

$$\begin{aligned}
L_{\text{eval}} &= \text{List of things evaluated by PJF},\\
S_{\text{sp}} &= \text{Selectivity of evaluating the small predicate},\\
S_{\text{pjf}} &= \text{Selectivity of evaluating the PJF},\\
S_{\text{join}} &= \text{Selectivity of evaluating the join},\\
losp &= \text{Likelihood of some pairs},
\end{aligned}$$

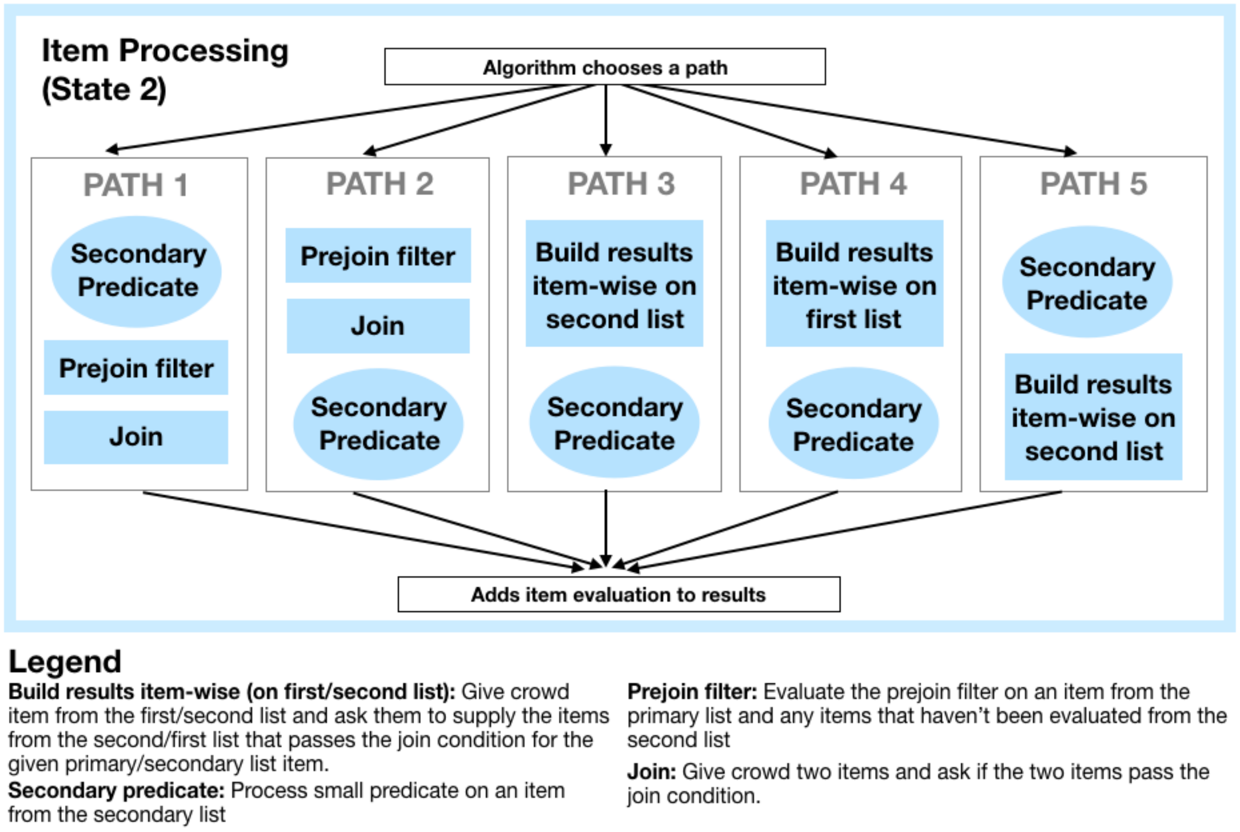

Item Processing (State 2)
Algorithm chooses a path
PATH 1
Secondary Predicate
Prejoin filter
Join
PATH 2
Prejoin filter
Join
Secondary Predicate
PATH 3
Build results item-wise on second list
Secondary Predicate
PATH 4
Build results item-wise on first list
Secondary Predicate
PATH 5
Secondary Predicate
Build results item-wise on second list
Adds item evaluation to results
Legend
Build results item-wise (on first/second list): Give crowd item from the first/second list and ask them to supply the items from the second/first list that passes the join condition for the given primary/secondary list item.
Secondary predicate: Process small predicate on an item from the secondary list
Prejoin filter: Evaluate the prejoin filter on an item from the primary list and any items that haven't been evaluated from the second list
Join: Give crowd two items and ask if the two items pass the join condition.

Figure 2: Possible paths in a two-list join process

Note that the "likelihood of some pairs" represents the probability that an item from the second list has at least one item from the first list that matches it by the join condition. This metric essentially acts as the selectivity of a join or itemwise task, as only secondary items that have one or more matches need to be tested by the secondary predicate after we have found pairs. With these in place, our equations become

$$\begin{aligned}\text{Cost}_1 &= C_{\text{sp}}(|L_2| - |L_{\text{eval}}|) + C_{\text{pjf}}(|L_1| + S_{\text{sp}}|L_2|) \\ &\quad + C_{\text{join}}|L_1||L_2|S_{\text{sp}}S_{\text{pjf}}, \qquad (8)\end{aligned}$$

$$\text{Cost}_2 = C_{\text{pjf}}(|L_2| + |L_1|) + C_{\text{join}}|L_1||L_2|S_{\text{pjf}} + C_{\text{sp}}losp * L_2, \qquad (9)$$

$$\text{Cost}_3 = (C_{\text{iw}} + N_{\text{match}}C_{\text{match}}\frac{|L_1|}{|L_2|}) * |L_2| + C_{\text{sp}}losp|L_2|, \qquad (10)$$

$$\text{Cost}_4 = (C_{\text{iw}} + N_{\text{match}}C_{\text{match}})|L_1| + C_{\text{sp}}losp|L_2|, \qquad (11)$$

$$\text{Cost}_5 = C_{\text{sp}} * |L_2| + (C_{\text{iw}} + N_{\text{match}}C_{\text{match}}\frac{|L_1|}{|L_2|}) * |L_2| * S_{\text{sp}} \qquad (12)$$

Note that, unlike the first set of equations, these have S terms multiplied by L_1 and L_2 in the later terms of each cost.

A couple important notes about our cost calculations: First, we assume that the cost of evaluating the prejoin filter on an item from list 1 is approximately the same as it is for an item from list 2. This is why, in Costs 1 and 2, only one prejoin filter cost appears. We also assume an approximately linear relationship between the time it takes to find matches for an item and the number of true matches for the item, as we expect it will take longer for a worker to find multiple matching items than a single matching item. This is why costs 3 through 5 represent the cost of an item-wise task as linear expressions related to the number of matches.

3.2.6 *Accounting for the Cost of Consensus.*

In each equation the terms appear in the order they occur in the join process. So in the equation for *Cost*$_1$ the first term corresponds to the cost of the small predicate tasks and the second and third terms are for the prejoin filter and join costs respectively. While these are correct in a way, they lose some information by assuming that it only takes one worker's response to find an answer. In reality, we require many workers' responses until we reach consensus, the number of which can vary greatly between different tasks and items. We can generalize the cost that it takes to get an answer to each task (small predicate, prejoin filter, or join) in the join process as the individual task cost times the number of tasks it normally takes to reach consensus. In other words we know these equations,

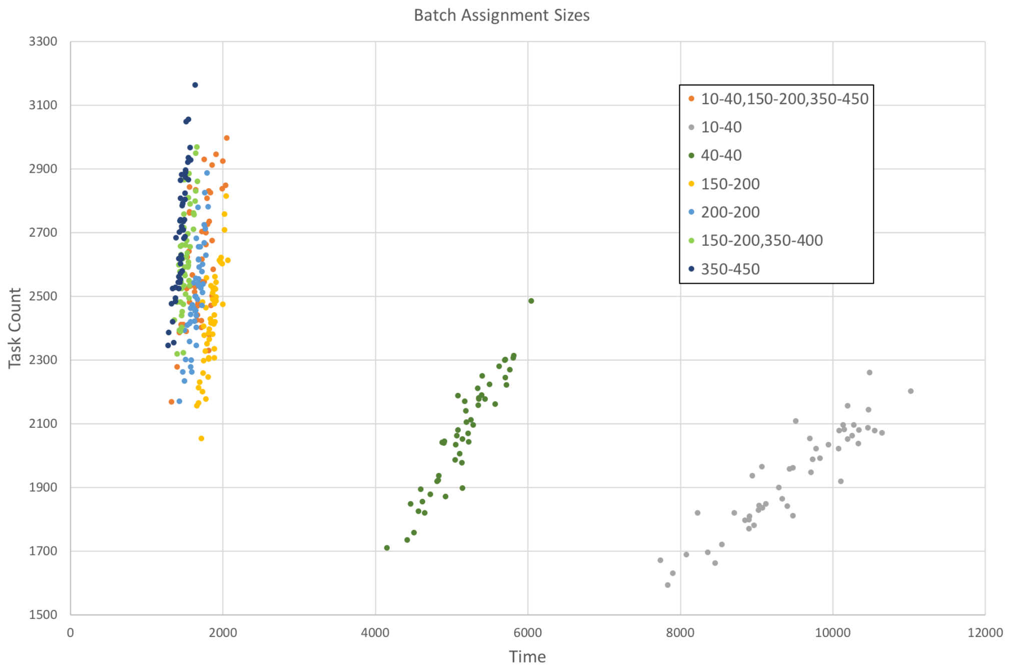


Figure 3: Each dot represents the total tasks and time for a single simulation with the relevant batch setting.

$$C_{\text{sp, tot}} = N_{\text{sp}} * C_{\text{sp}}, \tag{13}$$
$$C_{\text{pjf, tot}} = N_{\text{pjf}} * C_{\text{pjf}}, \tag{14}$$
$$C_{\text{join, tot}} = N_{\text{join}} * C_{\text{join}}, \tag{15}$$
$$C_{\text{iw, tot}} = N_{\text{iw}} * C_{\text{iw}}, \tag{16}$$
$$C_{\text{match, tot}} = N_{\text{iw}} * C_{\text{match}} \tag{17}$$

where,

N_{sp} = Number of tasks to reach consensus on items for the small predicate,
N_{pjf} = Number of tasks to reach consensus on items for the PJF,
N_{join} = Number of tasks to reach consensus on items for the join,
N_{iw} = Number of iw tasks to reach consensus on all matches for an item,

When we replace every cost instance with our new total cost variables defined above we get the following final cost equations:

$$\begin{aligned}\text{Cost}_1 = {} & N_{\text{sp}}C_{\text{sp}}(|L_2| - |L_{\text{eval}}|) \\ & + N_{\text{pjf}}C_{\text{pjf}}(|L_1| + S_{\text{sp}}|L_2|) \\ & + N_{\text{join}}C_{\text{join}}|L_1||L_2|S_{\text{sp}}S_{\text{pjf}},\end{aligned} \tag{18}$$

$$\begin{aligned}\text{Cost}_2 = {} & N_{\text{pjf}}C_{\text{pjf}}(|L_2| + |L_1|) \\ & + N_{\text{join}}C_{\text{join}}|L_1||L_2|S_{\text{pjf}} \\ & + N_{\text{sp}}C_{\text{sp}}\mathbf{\mathit{losp}} * L_2,\end{aligned} \tag{19}$$

$$\text{Cost}_3 = (C_{\text{iw}} + N_{\text{match}}C_{\text{match}}\frac{|L_1|}{|L_2|})N_{\text{iw}}|L_2| + C_{\text{sp}}\mathbf{\mathit{losp}}|L_2|, \tag{20}$$

$$\text{Cost}_4 = (C_{\text{iw}} + N_{\text{match}}C_{\text{match}})N_{\text{iw}}|L_1| + C_{\text{sp}}\mathbf{\mathit{losp}}|L_2| \tag{21}$$

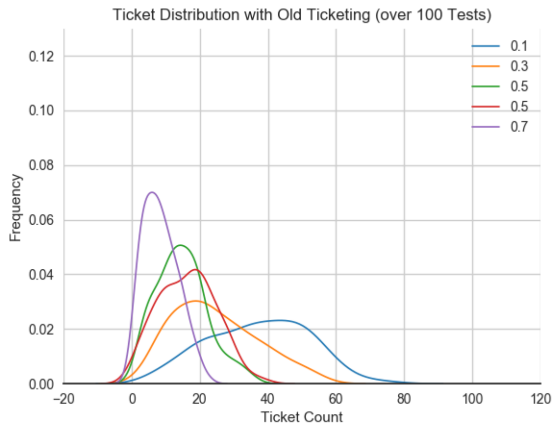
3.2.7 Collecting information implicitly eliminates some need for exploration. The nature of these costs and the fact that their equations share so many of the same variables simplifies the task of collecting cost estimates. For example, we know that one iteration through "path 1" (small predicate, prejoin filter then join) tells us something about the C_i's, S_i's and N_i's associated with that path. We can then use those preliminary estimates to get an idea of how costly "path 2" might be, even though we haven't explored path 2 yet. Thus, it takes fewer iterations down each respective path to get an accurate estimate of the cost of that path. Of course, there are some variables where this sort of implicit exploration is not possible (and we need to try a path several times before we have a working estimate) thus we still choose to implement a buffer-like period where exploration is valued over exploitation. For example, if there is a variable in the estimation that has not been calculated (there is not data for it) then we set the cost to zero, forcing the algorithm to attempt paths it has not attempted yet. In addition, we have a minimum "trial threshold", which is the minimum number of times that we need to attempt each path.

4 EXPERIMENTAL RESULTS

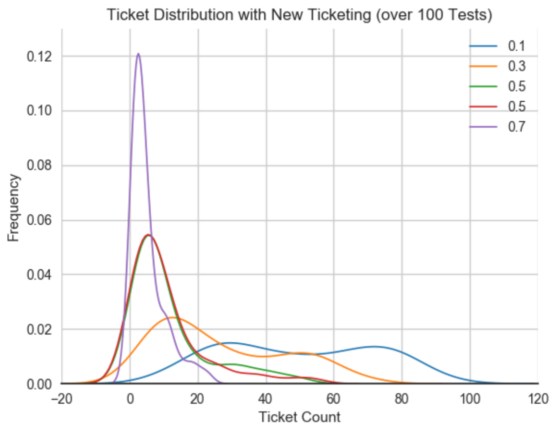
4.1 Concurrency

Our adjustments to the Dynamic Filter attempt to reduce the cost of a crowdsourced database query in terms of the total number of tasks and time steps passed before completing a query, particularly in the concurrent setting. We also look at how our changes affect bias towards selective predicates.

4.1.1 Experimental Setup. As we implemented changes to the Dynamic Filter algorithm, we ran simulations with synthesized



(a) Original Ticketing Method



(b) New Ticketing Method

Figure 4: The labels indicate the selectivity predicates. Only 5 out of 6 tested predicates were included in the ticket distribution graphs above for better visibility of the graphs. The full histograms with 6 predicates are in figure 11 of the appendix.

data based on input settings and data collected from HITs on AMT. For example, the task response time is selected from a distribution of response times of completed HITs from previous requests on AMT. Our standard test consists of 100 items and 6 predicates with varying selectivities and same ambiguities and difficulties as described in table 2. While previous research on Dynamic Filter has tested with 2 predicates, we tested with 6 predicates to observe the algorithm's bias on a range of predicate selectivities. We also have a configuration of simulation settings that we know generally performs well and seems realistic. For realistic simulations, we assume a total of 1000 active workers, 3 of who are assigned tasks every time step.

Table 2: Selected Predicates and Their Properties

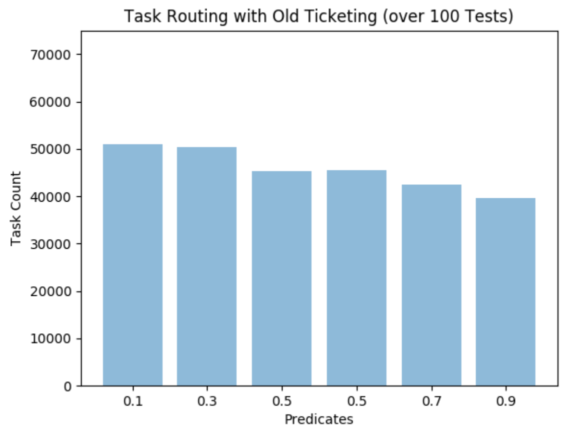
Predicate No.	Selectivity	Ambiguity	Difficulty
1	0.1	0.25	1
2	0.3	0.25	1
3	0.5	0.25	1
4	0.5	0.25	1
5	0.7	0.25	1
6	0.9	0.25	1

4.1.2 Batch Assignment. Figure 3 graphs the time and task count for queries using our default test configuration over 50 simulations for different task assignment methods: 2 constant assignment (40 constant active tasks, 200 constant active tasks), 3 fixed batch assignment (at 10 refill to 40, at 150 refill to 200, at 350 refill to 450), and our staged batch assignment settings (table 1). While most settings are concentrated in a clump, settings with refill caps of 40 take significantly longer than other settings. Although these settings also have fewer tasks overall, the trade off does not appear to be equal. Although the time for these settings is at least double the very worst time for other settings, the reduction in tasks as compared to the average of other settings is less than half.

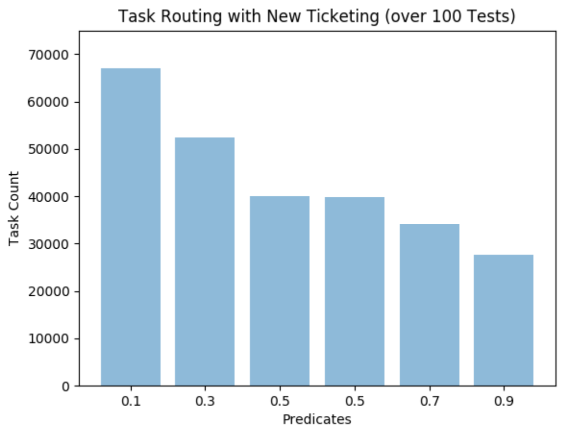
Due to randomness of the task assignment method, each setting has a varying number of completed tasks. It is interesting to note that batch assignments with refill cap of 40 have a greater variation in time versus tasks. Also, there is positive correlation between the task count and time, since it requires more time to complete more tasks. Settings that take less time have a higher task per time ratio, which is equivalent to the slope of the trend line of each setting. Observing the patterns of the tested batch methods, settings with high task-per-time ratio are more appropriate to apply when many tasks need to be complete, in order to reduce the total time of the query.

We also evaluated periodic batch assignment, which, instead of using a refill limit, refills after a certain number of timesteps. This method still used the the stages specified in table 1. A single-tailed t-test with p=.001 confirmed that there is no statistically significant difference in number of tasks or time among staged, 50-periodic, and 100-periodic batch methods. However, with longer periods, such as 150 and 200 time steps, time was significantly greater than the other methods, because the algorithm waits too long to refill batches.

4.1.3 Ticketing System. We ran 100 simulations for old and new ticketing systems, using our default 6 predicate distribution. Figure 4 depicts the number of tickets for each predicate after 100 simulations. We recognize that the ticket count in the original ticketing method does not fully represent the number of tickets each predicate holds throughout a simulation, since the old algorithm can take away tickets as items are completed. Under the new ticketing method, although the number of tickets for a predicate has higher variance, the distributions of ticket counts for selective predicates are shifted to the right. This indicates that the selective predicates on average receive more tickets and have a higher chance of receiving more tasks. Also, predicates with the same selectivity (0.5), represented by the green and red lines, have a similar distribution in figure 4b and are in fact directly on top of each other, meaning



(a) Original Ticketing Method



(b) New Ticketing Method

Figure 5: Number of tasks assigned to each predicate with different selectivities under the original and new ticketing methods

that the new ticketing system accurately differentiates the number of tickets by the predicate selectivities.

We found that queries with new ticketing took somewhat fewer tasks on average than old ticketing, with 2637.14 tasks for new ticketing and 2759.98 tasks for old ticketing. We determined this was a significant result using a single-tailed t-test with p=.001. Under the same statistical test, the times for each method were not found to be significantly different. This indicates that new ticketing, across the board, performs at least similarly to old ticketing. Beyond task and time steps, one of the other data points that interests us is the influence of the ticketing method on the bias in predicate routing. To try to characterize biases towards selective predicates, we recorded the overall routing of tasks to each predicate over the course of our tests. These can be seen in figure 5. In both graphs, we see that the ordering of task assignment more or less corresponds to the true predicate ordering. However, in the new ticketing graphs, there is a much more stark difference between between the highly and lowly selective predicates, which fits more into our intuition of how the algorithm's behavior should look.

A big part of our theory on why the new ticketing only does similarly to old ticketing, despite fitting what we would think is a more ideal pattern of bias, comes from looking over ticketing graphs from the full body of our testing data, with batches, ticketing, and preliminary tests for interesting future work. In many of our tests that perform particularly well, when we look at the ticketing over time, there isn't just one predicate that dominates all other predicates, but a grouping of highly selective predicates on top and a grouping of lowly selective predicates on bottom. The theory is that, although strongly favoring a single predicate can improve query performance early on, once most items have been routed to that predicate, then we need to start routing more items to other predicates. However, often in our experiments, by the time the most selective predicate is exhausted, few tasks have been routed to other predicates, so we have less information on their selectivities. We believe there may be a "sweet spot" between exploration and exploitation that we're not hitting, but only for the case of mixed selectivities. Ticketing can be tricky because it is both how the algorithm collects information and how it schedules it actions. The algorithm, in a way, doesn't collect information on what predicates are lowly selective. A very lowly selective predicate that has had many items routed to it might have a similar ticket count to a predicate that the algorithm has never visited. This means, in the case where there is only one highly selective predicate, more exploration of the lowly selective predicates won't result in much. Similarly, if there is one highly selective predicate, and any number of other predicates with very similar selectivities, all options after the most selective predicate is exhausted are equal, so exploration has at most minor benefit.

4.1.4 Overall Algorithm. With all of our alterations, we're interested in seeing how our algorithm compares with other benchmarks for simulations in the concurrent setting. Particularly, we were interested in two sets of algorithms. First are algorithms that choose items to work on using the same (or less) information than we use in Dynamic Filter. These algorithms are random assignment, which selects an IP pair randomly, and Rank MAB, which is described in the 2017 Dynamic Filter paper [10] and explicitly estimates a rank for each predicate. The second set of algorithms are "clairvoyant" algorithms, which choose IP pairs to assign based on knowledge of the exact selectivities of predicates as well as IP pairs' truth values. They are as follows:

Best Predicate First only assigns tasks to the most selective predicate that can still receive tasks. Worst Predicate First only assigns tasks to the least selective predicate that can still receive tasks. Proportional uses 1 minus the selectivities for the set of all predicates to create a probability distribution with which arms are pulled. For instance, with predicates .3 and .7, on average .3 would receive 70% of all tasks and .7 would receive 30% of all tasks. Best Pick selects a pair randomly from the set of all IP pairs whose ground truth is false. And finally, Worst Pick selects a pair randomly from the set of all IP pairs whose ground truth is true, and tries to assign to active IP pairs first. Notably, we don't have any benchmarks

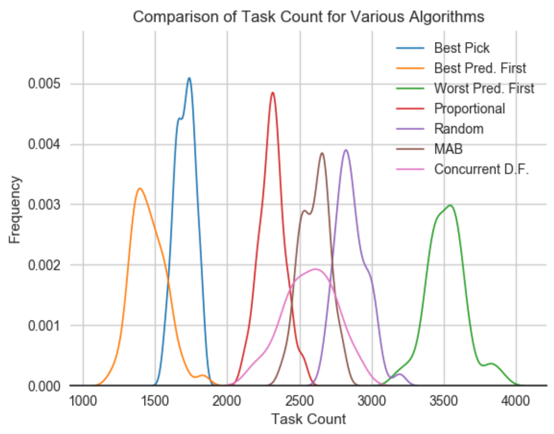


Figure 6: Comparison with Benchmark Algorithms: The Worst Pick Algorithm is omitted from the graph for better visibility of the graph. The full histogram with all benchmarks is in the appendix.

that modify the batch assignment system. Therefore, the results here mostly reflect the changes to ticketing, and test the overall algorithm against benchmarks in the concurrent setting, which was not done in previous work.

Notably, in figure 6, we see that all the positive clairvoyant algorithms appear to do better than our algorithm, and the negative algorithms do worse, as we would expect. This result was found to be significant with a one-tailed t-test where p = 0.001. We also see that both MAB and Dynamic Filter do better than Random, which we found was significant with the same test. However, a two-tailed t-test, with both p = .05 and .001 did not find that MAB and our algorithm performed with a significant difference. This makes sense, because the core algorithm shares many similarities with MAB, which is why it's a benchmark in the first place.

One particularly interesting thing from the data is the fact that Best Pick- which knows every pair that is false -does worse than Best Predicate First. We think that this is primarily because Best Predicate First, as much as it can, only works on one predicate at a time. This means that we rarely need to ignore tasks that we sent out because the item in their IP pair failed in another queue, which is currently a significant number of tasks for our concurrent algorithm. This is a result that we hope to explore more in future work, and hopefully is something we can use to make our algorithm perform more similarly to the clairvoyant algorithms.

4.2 Joins

Because joins are a relatively new introduction, we have not had time to do much testing. In this section, we explain our tests of prejoin filters and the conclusions that we came to from them. In addition, we explain some preliminary information from testing the newly-complete join system.

4.2.1 Our use of prejoin filters. Before we explain our tests on using prejoin filters, it is important to understand how we use them. We adapt previous work in prejoin filters by making some simplifying assumptions. We assume that two items passed through a prejoin filter only continue through the join process ("pass" the prejoin filter) if they are in the same prejoin filter category. Since we are checking for equality, we have no need to use the compatibility marking that Mitsuishi et al. do to avoid falsely rejecting tuples. In addition, to simulate runs for a variety of different joins and prejoin filters for data that is not necessarily available, we instead mimic the human-powered prejoin process by randomly deciding whether two items share the same prejoin category based on a preset selectivity. The selectivity in this case measures the likelihood of a pair having the same prejoin filter attribute, which captures the behavior we want by allowing us skip work when the two items should not match. We also provide an element of uncertainty, mirroring the tendency for human workers to give an incorrect answer. We are not aware of any significant inaccuracies in this approach.

4.2.2 Preliminary testing on when to use prejoin filters. We can imagine that a possible path for a join process would be to apply the small predicate then simply join items, avoiding the prejoin filter (In the interest of simplicity, for the rest of this section, we refer to matching two items by the join predicate as "joining" them. Recall that we are always in a join process while we run the prejoin filter and these paired "joins"). We reasoned that this might be preferable for lists of certain sizes or size disparities, or if the prejoin filter was costly to evaluate.

We ran preliminary tests to determine whether an adaptive algorithm could be effective in choosing between using a prejoin filter and not using a prejoin filter. We calculated cost estimates for both options and used these estimates to choose what was the "cheaper" path. We used a running average of the costs so far to determine which path to use on the next tuple. In our final algorithm we use a similar methodology of calculating a cost estimate, but we solve some of the limitations that we observe in this preliminary algorithm.

4.2.3 Limitations of preliminary algorithm. One such limitation is that it uses cost calculations that do not account for the uneven distribution of work over time that occurs when using a prejoin filter. The nature of a prejoin filter is to do a significant amount of work ahead of time categorizing items and later benefit by eliminating items in incompatible categories. Because our simulation works item-by-item, we run the prejoin filter on an item the first time we attempt to join it with something, which drives the price of prejoin-filtered joins early in the simulation higher. Later on, the cost will greatly decrease, but we do not capture this advantage because we use a backwards-looking average – simply the average cost thus far – to decide our path.

4.2.4 Expected vs. actual results. Since this is how the preliminary algorithm chose between two paths, there tends to be a bias away from using a prejoin filter when choosing a path dynamically. Since the algorithm allows us to toggle between two different paths, we expect the overall cost of the dynamically chosen run to be between the costs of each respective path run on all the tuples. In summary, we know that one path will be our "worst case" and the other will be our "best case," and we hoped to show that our algorithm was always better than the worst case and asymptotically approached the best case.

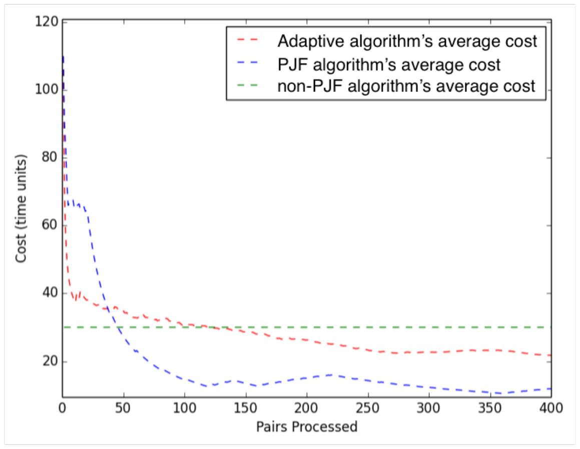


Figure 7: An experiment on adaptively choosing whether to prejoin filter where doing so is optimal

In figure 7 we see an example of a run we expected to see using our algorithm. This shows how the average cost of simply joining the two items, depicted in green, is consistent throughout the entire join process. Meanwhile, the average cost of using prejoin filters is high at first, as the prejoin filter is evaluated mostly at the start of the algorithm, driving up average cost early on. Later, we might encounter two items that have already been evaluated by the prejoin filter when they were considered as part of different tuples. In this case we can reap the benefits of the prejoin filter results we had saved from earlier and in some cases eliminate a tuple without needing to pass it through the join itself. This causes the average cost of the prejoin filter cost to drop significantly and rather quickly. In the beginning we see our adaptive algorithm (in red) joining items without using a prejoin filter since the average cost is high at first. The adaptive algorithm retains some explorative aspect as it attempts a few prejoin filter tasks while the bulk of the tasks are simply joined. When it becomes clear that using prejoin filter is more cost effective (given the lists are long enough to capitalize on the work done by the prejoin filter) it starts using that path more often and dips below the average join-only cost. Note that, because of its bias against using a prejoin filter, it takes significantly longer to drop than the prejoin filter.

For another configuration of settings, which include join selectivity, join cost, prejoin filter selectivity, prejoin filter cost and sizes of the two lists, we saw the results displayed in figure 8. In this case, the high cost of the prejoin filter and the short lengths of the lists were such that simply computing the join was better than doing all the pre-processing with a prejoin filter then joining. We can see that the average cost of the prejoin filter, depicted in blue, is dropping over time, but the short lengths of the lists causes the join to finish before the benefits of using a prejoin filter can influence the average cost. Meanwhile, the adaptive algorithm attempts to use the prejoin filter but strays away from the high prejoin filter cost and towards the relatively lower cost of computing the join outright.

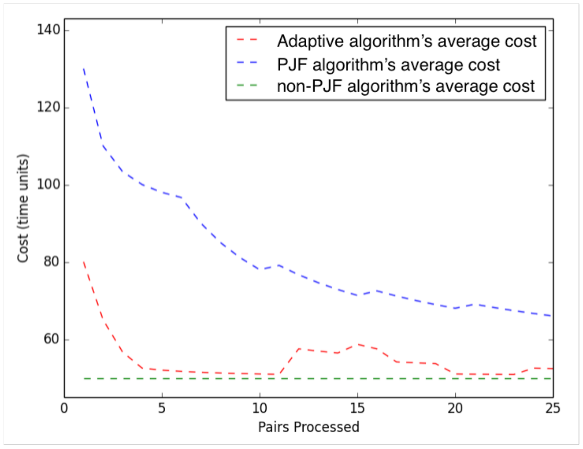


Figure 8: An experiment on adaptively choosing whether to prejoin filter where doing so is not optimal

Table 3: Settings in Representative Runs

Trial No.	Join selectivity	PJF selectivity	PW time	PJF time	Size 1	Size 2
1	0.3	0.3	10	10	50	40
2	0.3	0.9	10	100	100	20
3	0.3	0.6	10	10	50	40
4	0.3	0.6	10	10	100	20
5	0.3	0.6	10	10	100	20
6	0.3	0.6	10	100	100	20
7	0.3	0.3	10	100	100	20
8	0.3	0.9	10	10	50	40
9	0.3	0.9	10	100	50	40

These were the two types of behaviors we were expecting to see in the conception of the algorithm, which we expected to see in some reasonable proportion to one another. However, after running these sorts of tests across a range of settings, the data showed that the cases where we see results like in figure 8 were extremely rare, and almost exclusively occurred with lists of such small size that total time was relatively small. After some deliberation, we believe that these cases are not a major concern, as we do not expect Dynamic Filter to be deployed in use cases where the items are so few as to be easily handled by a single person. In a great majority of the cases, it was better to always use the prejoin filter before joining. After running hundreds of tests we came up with the following nine cases (shown in table 3) which we felt were representative of both the more extreme and moderate settings we might encounter within the scope of DynamicFilter.

Table 4: Results of Representative Runs

Trial No.	PJF Mean	Adaptive Mean	PJF StDev	Adaptive StDev	Test statistic	P value
1	3.33	5.57	0.89	0.98	-5.36	< .00001
2	14.89	15.32	0.71	0.29	-1.78	0.0511
3	6.28	8.04	0.53	0.40	-8.36	< .00001
4	6.69	8.43	0.84	0.57	-5.40	< .00001
5	6.82	8.51	1.06	0.72	-4.15	0.0004
6	12.09	15.01	0.49	0.21	-17.41	< .00001
7	9.16	14.68	0.52	0.19	-31.58	< .00001
8	9.49	10.00	0.35	0.19	-4.02	0.0008
9	13.52	14.31	0.45	0.12	-5.38	0.0004

Table 3 shows the settings for these nine representative cases. We expect the best case for a prejoin filter to be when the two lists are of similar size and both somewhat large, with a high prejoin filter selectivity and a low prejoin filter cost. The worst case is the opposite, with the two lists having very different sizes, one of them small. We ran a one-tailed t-test on the times of each setting, with a sample size of ten each. As can be seen in table 4, of the nine cases we looked at only one of the cases had a borderline insignificant difference in cost (using a confidence interval of 0.05). This difference is seen in trial number two, where the p value was 0.0511, just over our confidence interval of 0.05. Since this case was so close despite being our worst-case trial, we decided to rerun trial number two, but with twenty runs instead of the original ten runs. With this expanded testing, we reran our statistical analysis and found that the new p value was 0.03.

4.2.5 Mathematical analysis of using prejoin filters. To understand why it was that for all of our representative cases were better with prejoin filters, we took a look at what conditions make the overall cost of not using prejoin filters better than using prejoin filters. So taking equation (1) and setting it less than equation (2), we get equation (22) with outlines the condition that must be true for not using prejoin filters to be a more cost effective option.

$$\begin{aligned} |H \times M| c_1 &< |H+M| c_2 + f|H \times M| c_1, \qquad (22)\\ |H \times M| &< |H+M| \frac{c_2}{c_1} + f|H \times M|,\\ 1 &< \frac{|H+M| c_2}{|H \times M| c_1} + f,\\ 1 - f &< \frac{|H+M| c_2}{|H \times M| c_1},\\ \frac{(1-f)c_1}{c_2} &< \frac{|H+M|}{|H \times M|}, \qquad (23) \end{aligned}$$

In the scope of this project, we are assuming that no prejoin filter we use will cost more than the cost of the join itself. Under this assumption the worst case scenario for prejoin filter (the scenario that gives preference toward not using prejoin filters) would be when $c_1 = c_2$. Plugging these costs into equation (23) we get,

$$1 - f < \frac{|H+M|}{|H \times M|} \qquad (24)$$

Next we wanted to come up with some sort of bound on what we thought were realistic values for f, the fraction of the join pairs that pass the prejoin filter, which we also call selectivity [4]. We assume that a realistic upper limit of f would be $f = 0.9$, which would correspond to a very low selectivity for the prejoin filter. This gives us our final simplified equation for the worst case for our prejoin filters given the scope of the data used in Dynamic Filter.

$$0.1 < \frac{|H+M|}{|H \times M|} \qquad (25)$$

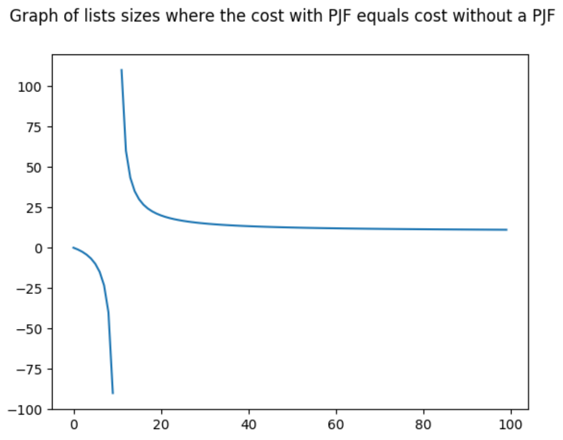


Figure 9: The curve of possible solutions where prejoin filter would be equal to no prejoin filter with j = 0.1

In figure 9, every point between the lines (since we cannot have negative list lengths this is the same as the area under the upper curve) represents the sizes of the two lists that would make using a prejoin filter non-optimal. Each axis stands for the size of one list, and as the graph is symmetric, it does not matter which is which. If we look at these solutions, we see that it is only when lists are small or of extremely disparate size that not using a prejoin filter is preferable. For example, two lists of size 8 and size 40 are different enough sizes that it would be better to not use prejoin filters. Again, this is under the more extreme case where $f = 0.9$ and $c_1 = c_2$. As these values become more reasonable, when $f < 0.9$ and $c_2 < c_1$, the lists need to be even smaller and more differentiated for the non-prejoin filter cost to be less. For example, for $f = 0.6$ and $c_2 = 0.5c_1$, we get much more restrictive solutions. For these settings, neither list can be larger than two elements before the prejoin filter is beneficial. To capture the effect of changing the selectivity and the relative costs we define a variable j:

$$j = \frac{(1-f)c_1}{c_2} \qquad (26)$$

which captures the factor that shifts the curve shown in figure 9. Figure 10 shows how the cost equality curve shifts as j changes. Increasing j could be a result of increasing the cost of the join

[4]Note that as stated previously, highly selective prejoin filters would have a f value closer to zero. So a value of $f = 0.9$ would be less selective since 90% of the pairs continue to the join task and thus also incur the cost of the join in addition to the prejoin filter.

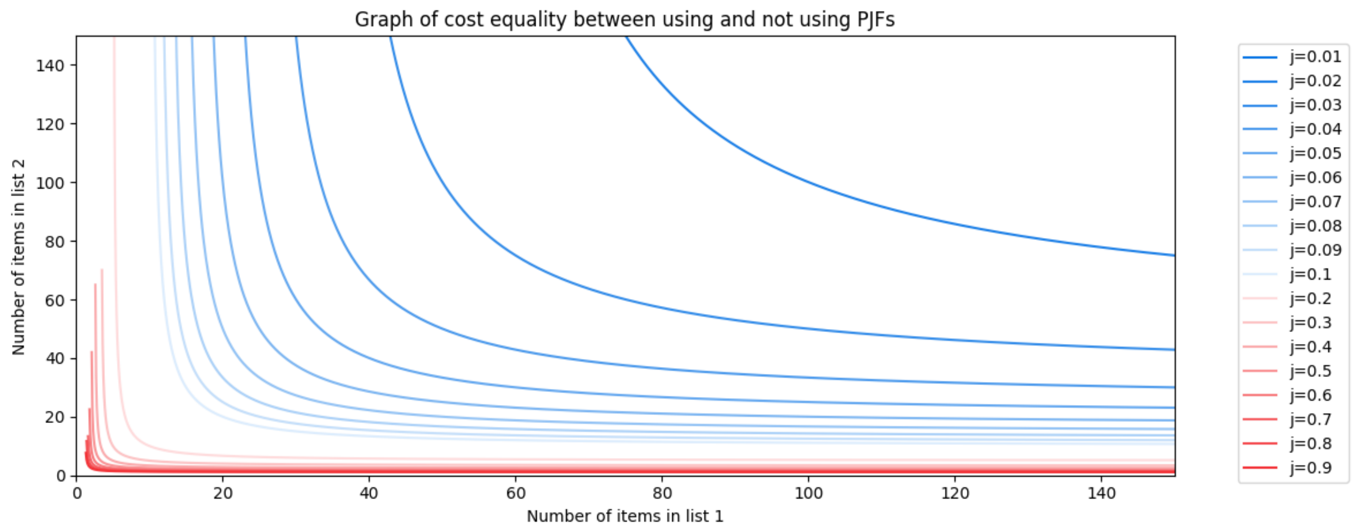


Figure 10: The curves of possible solutions where prejoin filter would be equal to no prejoin filter over various values for *j*

filter relative to the prejoin filter or decreasing the selectivity of the prejoin filter. In figure 10, keep in mind that (realistically) we assume j will not be less than 0.1 (which appears as a blue line with asymptotes at a size of 10). Additional lines for j values above that are displayed to show the general behavior of the equation.

4.2.6 Takeaways from Preliminary Testing. From this data, we came to the conclusion that the prejoin filter is better than the join-only approach (and thus also the adaptive approach) so often that the optimal adaptive solution, even a clairvoyant one, would almost always choose the prejoin filter anyway. Thus, since the use cases for no prejoin filter are so extreme, we eliminated this approach as one of our "viable" paths to consider in the overall join process.

A key conclusion from our preliminary testing was that future algorithms with cost estimators should take into account completed work and use forward-looking estimators. In other words, when we estimate how much paths with prejoin filters (and by extension small predicates) cost, we should use items we evaluate early on to find costs, but this time for how long each part of the process will take. Then we can use this information to focus on finding the cost of the entire join process. In addition, we need to account for how much the work we've done contributes to the work we will do in the future (for instance, a prejoin filter needs to be evaluated only once per item). This is something that we took into consideration when we came up with the cost calculations for paths in a join process.

4.2.7 Testing the complete join algorithm. Due to time constraints we were not able to test our join implementation rigorously. We will discuss some of the interesting behavior we notice in the initial tests and suggest areas for future investigation.

In an effort to test the functionality of all the paths, we ran tests that purposefully bypassed the dynamic selection of different costs and simply forced the algorithm to use just one path throughout the duration of the test. When we did this for path 5 (building results item-wise from the second list) we observed some interesting behavior. For example, we realized that due to the current ticketing system, finishing tasks from path 5 (which uses items from the secondary list rather than the primary list) does not affect the distribution of tickets to predicates. As a result, even though a particular predicate might be very selective, it does not get any probabilistic preference in the lottery draw. It is not until the predicate has been evaluated for all items from the second list that the algorithm knows the predicate is finished, and tickets are gained for the primary items that never matched a secondary item. This causes a huge spike in tickets for that predicate at the end of processing, which is inconsequential since it has already been evaluated on all items.

We also noticed that there were few items finished completely until the end of most runs. This is because there was continuous incremental progress being made on tasks across all IP pairs, leaving them to all be completed around the same time at the end of the join. Our runs that focused on paths 1 and 2 resulted in exponential spikes in ticketing at the very end of processing.

As these problems show, there remains significant work to be done on integrating joins into Dynamic Filter.

5 CONCLUSION

In this paper, we have expanded on Dynamic Filter to better leverage human computing. Our work in concurrency revised the algorithm to better suit the use case by allowing many workers to answer questions simultaneously. However, our algorithm still performed very similarly to Rank MAB, which we consider a reasonable and realistic benchmark for comparing our algorithm to existing work. Therefore, our work didn't necessarily result in making an algorithm that significantly outperforms existing algorithms, but instead establishes a strong foundation for future development of concurrency in the Dynamic Filter and better reflects the intended use case. In joins, we focused on using human powered computing to ask workers different, often more detailed questions about our data to reduce the number of tasks that a solely filter based query may ask. [JOIN TEAM!!]

6 FUTURE WORK

For concurrency, future work includes modifications on queue lengths, limiting the number of predicates an item is active in, and better simulating variation in task lengths. In our research, we did some testing with varying queue lengths. It would be helpful to have a better understanding of how queue lengths affect the overall performance of the algorithm, especially in respect to different numbers of items and predicates. Adaptive queues would allow the Dynamic Filter to dynamically change the lengths of each predicate queue throughout a query based on the overall number of predicates and items, as well as the number of lottery tickets each predicate holds, which is an estimation of predicate selectivity and cost. Another issue of interest is, as seen in the Best Predicate First algorithm in 4.1.4, applying a limit to the number of predicates an item can concurrently be evaluated for, which we call a "predicate limit", might reduce the overall task count. In regards to running more realistic simulations, we would like to define an additional predicate attribute, "difficulty", which represents the time it takes to complete a task for the predicate. This could be useful, because the previous implementation does not account for the fact that different predicates may require a different amount of time for a worker to evaluate. Finally, we want to work more to identify the exact trade-off between task and time, to help us determine the *best* adjustments to the Dynamic Filter.

For joins, there are many changes we would like to investigate. In the future, changes can be made to give preference towards finishing join tasks already in progress so that the algorithm can finish items more serially, allowing the join to gather more information from responses and take better advantage of Dynamic Filter's adaptive nature.

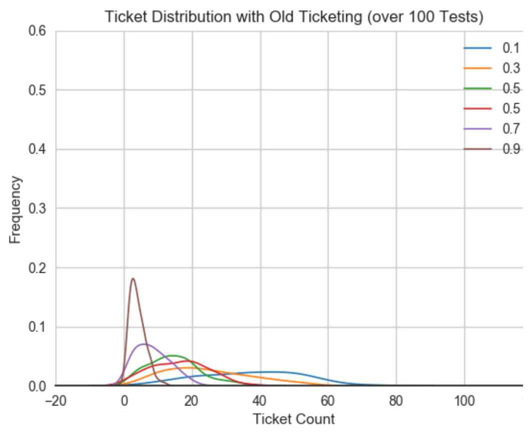
We plan to experiment with using the crowd to generate a cardinality estimate for the two lists. This would allow the algorithm to consider other, potentially more cost-efficient, processing paths. For example, if the size of the second list was estimated to be fairly small then the algorithm might consider simply enumerating the second list [5] before moving immediately to state 2 as depicted in figure 1. A cardinality estimator could also potentially be used to find which paths should be preferred before processing even begins.

We intend to experiment with what we will call "partial enumeration". By partial enumeration, we mean that instead of switching to state 2 when the entire list has been enumerated, instead we could allow some processing in state 2 before the entire secondary list is enumerated. The addition of a cardinality estimator would be beneficial for this purpose, as we would be able to estimate certain enumeration-dependent variables in the cost estimates with the results of a cardinality estimate.

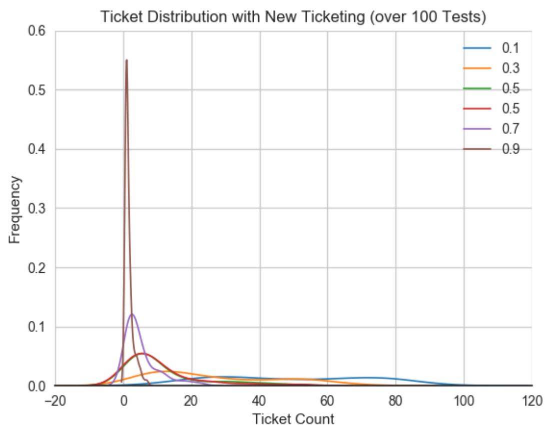
It would be in our best interest to gather data on join tasks on the Mechanical Turk, which can be used as a basis for setting cost means and standard distributions in simulations. Once we have a better idea of how long, difficult, and selective various join tasks are, we can provide more interesting data on when various paths will be preferable.

There is also room for more research into other path choosing metrics in state 2. For example, we would like to consider a ticket-based system that rewards paths for various successes. We would also like to examine whether or not it would be feasible to predict the best path for individual items rather than the entire join.

[5] This would entail sending out tasks specifically designed for list enumeration, rather than join tasks that can collect enumeration information as a secondary role.



(a) Original Ticketing Method



(b) New Ticketing Method

Figure 11: Full 6 predicate ticket distributions. These results are discussed in section 4.1.3

A APPENDICES

This appendix includes some graphs excluded from the paper due to scale.

A.1 Experimental Results

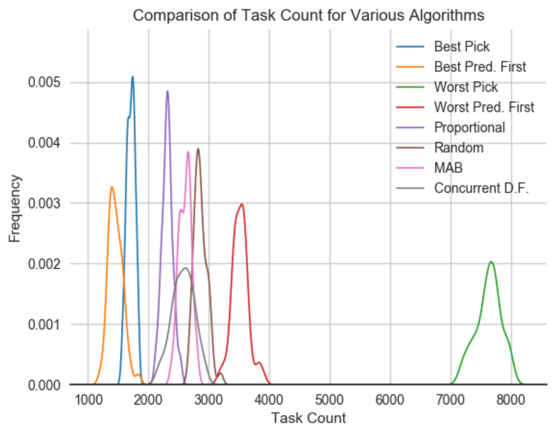


Figure 12: Full task count histogram for our algorithm and the benchmarks. These results are discussed in section 4.1.4

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant No. 1359170 and Grant No. 1657259.

REFERENCES

[1] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *SIGMOD Rec.* 29, 2 (May 2000), 261–272. https://doi.org/10.1145/335191.335420
[2] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. CrowdDB: Answering Queries with Crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/1989323.1989331
[3] Sudipto Guha, Kamesh Munagala, and Martin Pal. 2010. Multiarmed Bandit Problems with Delayed Feedback. *CoRR* abs/1011.1161 (2010). arXiv:1011.1161 http://arxiv.org/abs/1011.1161
[4] Pooria Joulani, András György, and Csaba Szepesvári. 2013. Online Learning under Delayed Feedback. *CoRR* abs/1306.0686 (2013). arXiv:1306.0686 http://arxiv.org/abs/1306.0686
[5] Doren Lan, Katherine Reed, Austin Shin, and Beth Trushkowsky. 2017. Dynamic Filter: Adaptive Query Processing with the Crowd. In *HCOMP*.
[6] Travis Mandel, Yun-En Liu, Emma Brunskill, and Zoran Popović. 2015. The Queue Method: Handling Delay, Heuristics, Prior Data, and Evaluation in Bandits. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI'15)*. AAAI Press, 2849–2856. http://dl.acm.org/citation.cfm?id=2886521.2886718
[7] Adam Marcus, Eugene Wu, David Karger, Samuel Madden, and Robert Miller. 2011. Human-powered Sorts and Joins. *Proc. VLDB Endow.* 5, 1 (Sept. 2011), 13–24. https://doi.org/10.14778/2047485.2047487
[8] Tomomi Mitsuishi, Atsuyuki Morishima, Norihide Shinagawa, and Hideto Aoki. 2013. Efficient Evaluation of Human-powered Joins with Crowdsourced Join Pre-filters. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication (ICUIMC '13)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2448556.2448563
[9] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. 2013. Crowdsourced enumeration queries. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 673–684. https://doi.org/10.1109/ICDE.2013.6544865
[10] Palanker Melaku Schmit Trushkowsky, Gallina-Jones. 2017. DynamicFilter: Adaptive Query Processing with the Crowd. (2017). Unpublished paper.