Crowdsourced Joins for Filter Queries

Cienn Givens
Harvey Mudd College
Claremont, CA
cgivens@hmc.edu

Anna Serbent
Harvey Mudd College
Claremont, CA
aserbent@hmc.edu

Beth Trushkowsky
Harvey Mudd College
Claremont, CA
beth@cs.hmc.edu

1. INTRODUCTION AND MOTIVATION

Human computation, also called crowdsourcing, leverages people to complete computational tasks. Web platforms such as Amazon's Mechanical Turk provide a programmatic interface for recruiting crowd workers to work on small tasks that typically take less than a few minutes; e.g., labeling an image. Our work investigates using crowdsourcing within query processing systems such as a database management system (DBMS) to broaden the scope of questions users can ask about their data.

This project builds on previous work on Dynamic Filter [3], an algorithm for filtering a list of items using criteria that each require human knowledge, perception, or experience; the algorithm uses the crowd to evaluate each item to see which satisfy all filtering criteria. For example, a user's query could be to filter a list of hotels to find which ones have (1) a nice view from the room and (2) a gym that is open 24 hours.

In our current work, we aim to improve the efficiency of the Dynamic Filter algorithm by giving special consideration to filtering criteria that have a particular form. As an example of this form, consider again the example use case of filtering a list of hotels. Suppose there is a third filtering criterion: the hotel should be within one mile of a metro station with an elevator. The general form is when a filtering criterion refers to a secondary list of items, in this case a list of metro stations, that in turn has its own filtering criterion (has elevator). Connecting items from one list (hotels) to items from a second list (metro stations) using some constraint ("within one mile") is referred to as a *Join* operation in database systems. We call a filtering criterion that has this form "join-able", meaning that it can be treated as a join with a secondary list with its own filtering criterion, the secondary filter.

Our work investigates join-able filters in order to reduce the amount of total crowdsourcing work needed to complete the query. In the hotels example, overall worker effort could be reduced if multiple hotels are close to the same station: after the station is checked for having an elevator for the first hotel, it will not have to be checked again for other hotels.

In this poster we describe our approach to some key challenges for realizing the benefits of join-able filters. The first is determining the possible approaches for processing the join and the secondary filter using the crowd. As is typically the case in query processing systems, different approaches may be better for different queries. Thus the second challenge is determining how the amount of overall crowd worker effort, representing query *cost*, should be characterized and then used to decide on an approach for a given query. Finally, there is the challenge of how to estimate the cost of different approaches for a query because in Dynamic Filter the true costs are unknown at query time.

2. BACKGROUND AND RELATED WORK

The process of determining the best, low-cost approach for executing a query in a database system is called query optimization. Related work in query optimization for crowd-powered database systems assumes query costs are known before the query runs [2]. Our work investigates when these costs are unknown at query run time. Dynamic Filter is an example of adaptive query processing [1]: the execution approach for a query is modified as it runs. The algorithm learns estimates of the query's possible execution approaches while the query is running and then adapts its strategy.

Conceptually, a DBMS Join pairs every item from its first input list with every item from its second input list, called the cross-product; pairs that satisfy the join *condition* are the result of the join. Our work addresses *crowd-powered* joins, where the join condition is evaluated by the crowd. Join algorithms in a traditional DBMS aim to more efficiently evaluate the same result as the conceptual evaluation. Part of our approach to efficient crowd-powered joins involves the idea of a *pre-join filter* [4]. A pre-join filter asks the crowd to categorize each item from the two input lists before the join condition is tested on pairs; two items in the same category have a much higher likelihood of matching by the join condition (in many cases, to the extent that two items from different categories cannot match) [5]. For example: if the lists were hotels and metro stations, each could be categorized with the city where they are located.

3. APPROACH

Our work focuses on identifying approaches to a crowd-powered join, characterizing their costs, and determining how to estimate those costs while a query is running. Our adaptive algorithm would use cost estimates to shift its execution strategy over time.

3.1 Defining Join Paths

We call each approach for a crowd-powered join a *path*: a sequence of evaluation steps each done by the crowd. Join-able filters always have the secondary filter as one of the steps; the join takes at least one step but may be broken into multiple steps.

The space of possible paths includes different algorithms for implementing the crowd-powered join as well as differ-

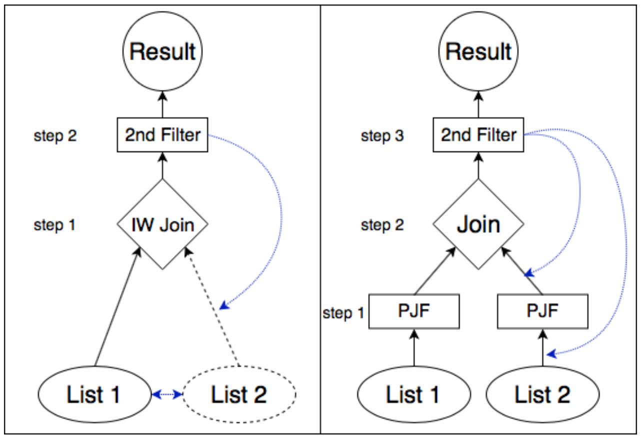

Figure 1: Join base paths. Blue dotted arrows show ways to reorder steps, creating new paths. Dashed lines denote a list that is created as the query runs.

ent orders of the steps in a path. We highlight two join techniques, forming two *base* paths as shown in Figure 1.

The first path, on the left in Figure 1, has two steps: perform what we call an item-wise join and then apply the secondary filter. The item-wise join takes each item from the primary list (e.g., hotels) and asks the crowd for every secondary-list item (metro station) that satisfies the join condition (within one mile) for the primary list item. Having the crowd find the contents of secondary list is unique to crowd-powered joins because the secondary list can be unknown at the start of the query. The item-wise join only evaluates as much of the cross-product between the two lists as necessary. Reordering of steps in this base path include doing an item-wise join starting with items from the secondary list, or evaluating the secondary filter before the join.

The second base path could be used when some or all items in the secondary list are known; see right in Figure 1. It has three steps: first, apply a pre-join filter to both lists, then give possible pairs to the crowd to be verified, then apply the secondary filter. The pre-join filter can greatly reduce the number of pairs that must be compared by the crowd. We can create new paths from this one by applying the secondary filter before the join, either before or after the pre-join filter.

3.2 Determining Path Cost

In order to be able to choose the best path for a given query, we must characterize each path's cost. The item-wise base path must find all pairs for each item in the primary list and then run the secondary filter on each secondary-list item that is joined to a primary-list item. The cost of the item-wise base path can be expressed as

$$C_{p1} = C_{fp}N_{pr} + C_{sf}(N_{sec}S_{jp}),$$

where C_{fp} is the average cost of finding all pairs for one item, N_{pr} is the number of primary-list items, C_{sf} is the average cost of applying the secondary filter to a single item, N_{sec} is the number of secondary-list items, and S_{jp} is the fraction of secondary-list items that have a match in the primary list. The cost of the pre-filter base path is

$$C_{p2} = C_{pj}(N_{pr}+N_{sec})+(N_{pr}\times N_{sec})S_{pj}C_{jp}+C_{sf}(N_{sec}S_{jp}),$$

where C_{pj} is the average cost of applying the pre-join filter to one item, S_{pj} is the likelihood that a pair shares the same pre-join filter category, and C_{jp} is the cost of evaluating the join condition on a pair.

3.3 Estimating Cost to Choose a Path

In our setting these costs will not be known at query time, so the query processing algorithm will need to form estimates while the query is running. To do so, it can start with an initial path and form cost estimates as it observes some cost information in real time. Notably, only a few terms in the two path cost expressions differ: C_{fp}, C_{jp}, C_{pj}, and S_{pj}. With sufficient estimates of these values, the algorithm can learn which path is optimal by running either path.

Our current approach is to begin with an exploratory period that establishes an estimate of some of the relevant costs by running paths equally; the algorithm then switches to a strategy in which it just uses the path with the lowest estimated cost. We aim to shorten the exploratory period by noting when observations made for one path's cost inform the other's cost, beyond the shared terms in their cost expressions. For example, the cost of finding pairs is related to the number of pairs and the cost of applying the join condition to a pair. Future work includes further exploring these relationships.

4. CONCLUSION AND FUTURE WORK

Transforming filters to crowd-powered joins can improve the efficiency, and thus the utility, of crowd-powered query processing systems. We have shown two approaches to crowd-powered joins, discussed their costs, and described a way to adaptively choose the best path for a given query.

Next steps include identifying additional join paths, as well as different approaches to estimating cost. For example, we will explore adding a step that asks crowd workers to estimate the relative size of the two lists. Finally, we plan to test our approach using Amazon's Mechanical Turk.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

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

[2] J. Fan et al. Crowdop: Query optimization for declarative crowdsourcing systems. *IEEE TKDE*, 27(8):2078–2092, 2015.

[3] D. Lan, K. Reed, A. Shin, and B. Trushkowsky. Dynamic filter: Adaptive query processing with the crowd. In *HCOMP*, 2017.

[4] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *Proc. VLDB Endow.*, 5(1):13–24, Sept. 2011.

[5] T. Mitsuishi, A. Morishima, N. Shinagawa, and H. Aoki. Efficient evaluation of human-powered joins with crowdsourced join pre-filters. In *ICUIMC*, pages 7:1–7:6, New York, NY, USA, 2013. ACM.