Purdue University
Purdue e-Pubs

Cyber Center Publications | Cyber Center

2013

The Palm-tree Index: Indexing with the crowd

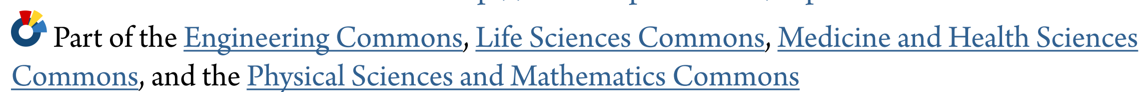
Aamer Mahmood
Purdue University - Main Campus

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Eduard Dragut

Saleh Basalamah
Umm Al-Qura University

Follow this and additional works at: http://docs.lib.purdue.edu/ccpubs


Part of the Engineering Commons, Life Sciences Commons, Medicine and Health Sciences Commons, and the Physical Sciences and Mathematics Commons

Mahmood, Aamer; Aref, Walid G.; Dragut, Eduard; and Basalamah, Saleh, "The Palm-tree Index: Indexing with the crowd" (2013). *Cyber Center Publications.* Paper 537.
http://docs.lib.purdue.edu/ccpubs/537

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

The Palm-tree Index: Indexing with the crowd*

Ahmed R. Mahmood
Purdue University
amahmoo@purdue.edu

Walid G. Aref
Purdue University
aref@cs.purdue.edu

Eduard Dragut
Purdue University
edragut@purdue.edu

Saleh Basalamah
Umm Al-Qura University
smbasalamah@uqu.edu.sa

ABSTRACT

Crowdsourcing services allow employing human intelligence in tasks that are difficult to accomplish with computers such as image tagging and data collection. At a relatively low monetary cost and through web interfaces such as Amazon's Mechanical Turk (AMT), humans can act as a computational operator in large systems. Recent work has been conducted to build database management systems that can harness the crowd power in database operators, such as sort, join, count, etc. The fundamental problem of indexing within crowdsourced databases has not been studied. In this paper, we study the problem of tree-based indexing within crowd-enabled databases. We investigate the effect of various tree and crowdsourcing parameters on the quality of index operations. We propose new algorithms for index search, insert, and update.

1. INTRODUCTION

Crowdsourcing is the practice of solving large problems by dividing them into smaller tasks, each of which is then solved by humans from an online community. The tasks are usually difficult to be performed automatically by a computer as they require human intelligence. Typical crowdsourcing tasks involve labelling, ranking, data cleaning, data filtering, data collection, and entity matching (e.g., see [11, 6, 2, 15, 5, 14]). Websites, e.g., Amazon's Mechanical Turk (MTurk) [1] provide an infrastructure for organizations to submit numerous micro-tasks and collect their results after they are fulfilled by human workers recruited at those websites. In a typical crowdsourced application, tasks are replicated and are answered by multiple people to avoid single user errors. Each person answering a task incurs a (monetary) cost. Thus, a challenging problem for crowdsourcing is that of optimizing the number of tasks while maintaining the accuracy of the results.

Consider the following scenario where a crowdsourcing tree-based index can be useful. Assume that a car repair shop wants to provide an online service that allows users to submit pictures of their damaged cars to get an estimate of repair cost of their car. The repair shop maintains images of cars previously repaired associated their actual repair cost. A good repair estimate would be the actual repair cost of previously repaired car of similar condition. Building a tree index on images of previously repaired cars sorted based on their repair cost allows performing crowdsourcing-based similarity queries on the index. These queries help users to find cars with similar conditions and hence anticipate the same repair cost to their own damaged car. The key to the success of this scenario is that the cost of repair is directly proportional to the condition of the car. The same idea can be applied when estimating the selling price of a used car, the cleaning cost of rental rooms, the placement of a new soccer player in player rankings or the ranking of new scientific publications, etc.

In this paper, we introduce the *palm-tree index*; a crowdsourcing tree-based index. We call our proposed crowdsourcing tree-based index *palm-tree index* as real palm trees also need *humans* in order to move the pollen from one male tree to the other female trees to produce fruits (dates). The palm-tree index aims at indexing keys based on properties that need human intelligence, e.g., to compare images against each other in order to descend and navigate the index. Crowdsourcing-based tree indexing is useful for (1) the ordering of a set of keys based on a subjective property, (2) performing index nested-loops joins, and (3) answering range queries, among other typical uses of an index. The problem of crowdsourcing-based indexing is challenging because of the following issues. First, crowd-based comparisons are subjective and are prone to error. Hence, we need to find strategies that give the most accurate and consistent results. Second, comparison tasks performed by the crowd incur a (monetary) cost. Hence, optimizing the number of tasks while preserving accuracy is very important.

The contributions of this paper are summarized as follows:

- We introduce the problem of crowdsourcing tree-based indexing as a technique for ordering a set of items with "subjective" keys.
- We present a taxonomy for crowdsourcing tree-based indexes.
- We introduce the palm-tree index, a crowdsourced index, along with several accompanying index traversal algorithms.
- We study the quality of the results retrieved using the palm-tree index.

The rest of this paper is organized as follows. Section 2 introduces the taxonomy for crowdsourcing tree-based indexes. Section 3 presents notations used throughout the paper. Section 4 presents the palm-tree index and its search algorithms. Section 5 analyzes the proposed algorithms. Preliminary experimental results

*This work was partially supported by the National Science Foundation under Grants III-1117766 and IIS-0964639.

Copyright © 2013 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

are given in Section 6. Related work is presented in Section 7. Section 8 contains concluding remarks.

2. PROBLEM TAXONOMY

In this section, we introduce a taxonomy for crowdsourcing tree-based indexes. This taxonomy can be seen as extension to the one presented in [8], which addresses only worker motivation, task replication and task assignment. Our taxonomy adds concepts related to tree indexing (e.g., tree height) and to crowdsourcing in general (e.g., worker experience). Figure 1 depicts the taxonomy.

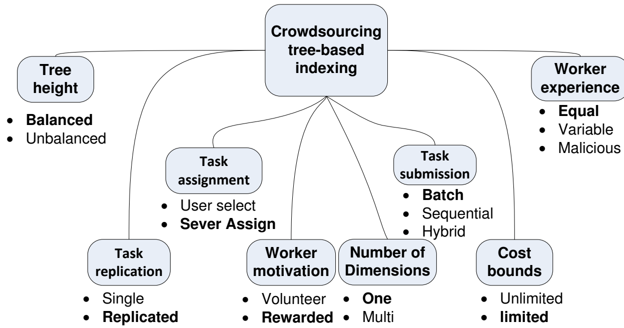


Figure 1: A taxonomy of crowdsourcing tree-based indexing. The concepts utilized in this paper are shown in boldface.

Tree height It specifies whether the tree is balanced or not. The proposed palm-tree index is a balanced B^+-tree-like structure to obtain predictive query cost. More detail about query cost is given in Section 3.

Number of dimensions An index can be either one-dimensional or multi-dimensional. In this paper we consider one-dimensional indexing.

Worker motivation In some situations, workers may be willing to perform tasks for free, e.g., see [8], However, in most instances workers seek rewards for their services. This raises several issues, e.g., budget estimation and the presence of low quality work due to workers solving tasks quickly to obtain more money. In this paper we consider rewarded workers.

Worker experience A simple model to describe workers is to assume that all workers have equal experience, e.g., as in [11]. An alternative and viable model is to differentiate workers based on their level of expertise, e.g., as in [3]. In the palm-tree index, workers are assumed to have indistinguishable, i.e., equal, expertise. We plan to extend our analysis to the general case when workers have variable experience. One special type of workers is a malicious worker. A malicious worker aims at degrading output quality by providing intentionally faulty and misleading answers. Coping with malicious workers is not addressed in this paper.

Task assginment There are two cases to consider: (1) The worker chooses the tasks to work on, or (2) the tasks are automatically assigned to the worker without the workers interference in the selection. This dimension is particularly important when workers are of variable expertise. For example, it is important to avoid assigning difficult tasks to inexperienced workers. In this paper, we assume Case 2, above.

Task replication In a *single* task no-replication model, workers are trusted to deliver high quality answers. In many other situations, workers' responses incur error. The *Replicated* task model allows aggregating multiple workers' opinions to increase the quality of the results. In this paper, we assume the replicated model.

Task submission model A task submission model describes how replications of the same task are submitted to workers. One alternative is to submit all task replicas in a single *batch*. This technique can reduce the response time to get the final results as task replicas are solved by multiple workers concurrently. Nonetheless, one may end up submitting more tasks than needed. In order to reduce the cost needed for simple tasks, the replicas of same task can be submitted *sequentially*. In the sequential model, a replica of a task is submitted only when the outcome of the previous replicas does not produce results that are of acceptable quality. The sequential technique increases the task response time as replicas wait for each other. A *hybrid* submission model is a middle-ground between both alternatives. In the hybrid model, small-sized batches of the same task are submitted sequentially. In this paper, we only consider the batch task submission model, i.e., that replica tasks are submitted in parallel to the designated workers at the same time.

Cost bounds Having *unlimited* cost per worker is a rare occurrence. It may occur when the quality of the result is of major importance. Typically, cost for tasks are *limited* and optimizing the overall task error within cost bounds is of major importance. In this paper, we consider the limited cost case.

3. PRELIMINARIES AND INDEX MODEL

3.1 Problem definition

Let S be a set of N items and q be a query item. The keys of these items are subjective or imprecise (e.g., the item is the photo of a damaged car and the key is the cost of repair for this car). We need to address the following issues:

- Order the items in S according to their keys.
- Construct (build) an index on S.
- Submit query q on S to the crowd.
- Aggregate crowds responses to query q to produce a final result .

3.2 Index model

The structure of the palm-tree index is composed of two components: the index manager and a B^+-tree. Figure 2 gives the main components of the palm-tree index. We distinguish between two main concepts in the palm-tree; *jobs* and *tasks*. A *job* is the unit of work submitted by a palm-tree user to perform either a query or an insertion on the indexed data. A *task* or *HIT* (*H*uman *I*ntelligent *T*ask) is the smallest unit of work assigned to a worker. Typically, a job involves generating multiple tasks.

3.2.1 The B^+-tree Component

The palm-tree index is built on top of a B^+-tree index. Each node in the B^+-tree has n ordered keys and represents one task. [11] observes that human beings are capable of performing n-ary operations with ease. Therefore, we give to a worker a sorted list A of n items, e.g., images or videos, along with the query item q,

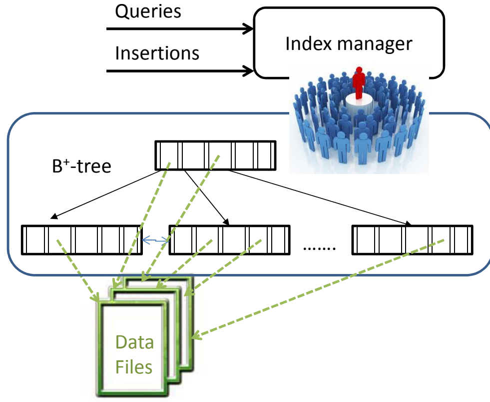


Figure 2: palm-tree index model.

e.g., a query image or video, and ask the worker to place q in A based on how q compares to the items in A. For instance, A can be a list of 5 pictures of cars sorted according to the extent to which the car is damaged, while q can be the picture of a new damaged car. The fanout f of the B$^+$-tree (the maximum allowed children of a node other than the root) naturally captures the maximum n-ary operations that a human can accomplish with ease. Clearly, a human being can handle a lot easier a list A with 5 pictures than one with 20. Determining the optimal f for a given problem requires an initial phase of training. We discuss this issue in more detail in Section 5.

Construction (building) of the palm-tree depends on the type of indexed keys. We have two types of keys: *quantitative* and *qualitative* keys.

DEFINITION 3.1. ***A quantitative key*** *is a key to be indexed that has two proportional properties, namely, a subjective property and an assessed value. For quantitative keys, the palm-tree index can be constructed by sorting keys based on the assessed values then bulkload the index.*

DEFINITION 3.2. ***A qualitative key*** *is a key to be indexed based on a subjective property only. It does not have any associated assessed value. For qualitative keys, a palm-tree index can be constructed only using successive insertions and human-based comparisons.*

One example of a quantitative key is images of damaged cars associated with actual repair costs. The degree of car damage is the subjective property of the key while the repair cost is the quantitative key. An example of a qualitative key is images of butterflies with an ordering based on beauty as a subjective property. Regular B$^+$-tree splitting/merging algorithms are used during insertion/deletion into/from the B$^+$-tree. For both types of keys, queries use human intelligence to make comparisons needed to search the B$^+$-tree. All comparisons are based on the subjective property only.

3.2.2 Index manager

The palm-tree index manager is responsible for tree construction and query processing within the palm-tree. It initiates jobs for users' requests, generates tasks for every job, assigns tasks to workers, and aggregates workers' responses. We use majority voting to aggregate responses of workers.

In the palm-tree index, a task consists of items (e.g., pictures) in a node in the B$^+$-tree. A worker is asked to choose the best location for a query item among these items. Figure 3 gives an example task

Table 1: Palm-tree index operations

Operation	Quantitative keys	Qualitative keys
Query	Crowd-search	Crowd-search
Insert	B$^+$-tree insert	Crowd-search then B$^+$-tree insert
Build	Bulk loading or Successive inserts	Successive inserts
Delete	Query then B$^+$-tree delete	Query then B$^+$-tree delete
Update	Delete then insert	Delete then insert

for estimating the repair cost of a car. The image on top represents the query image, while the images below are the items in a node in the tree. The worker assess where the query key fits among the current nodes keys.

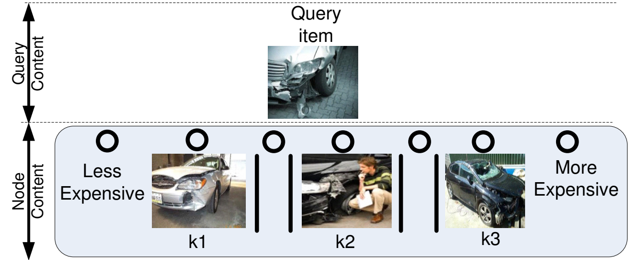


Figure 3: Sample task for choosing the best branch to estimate the car repair cost.

4. ALGORITHMS

Table 1 describes how the build, insert, update, delete, and query operations are performed in the palm-tree index. Regular B$^+$-tree algorithms (i.e, insert-split) are performed directly and do not involve the crowd. We focus on the crowd-search operation (i.e., querying the palm-tree index using the crowd) because the other operations depend on it.

The original search algorithm for the B$^+$-tree index is not directly applicable for the crowdsourced palm-tree index. The palm-tree search algorithm is dependent on the way the palm-tree is traversed and the way the workers answers are aggregated. We identify three search strategies for the palm-tree search algorithm. All three search strategies share the following two steps. The first step is *root_task_generation* that generates tasks at the root level and is common for all three strategies. The second step is the *response_handler* that generates tasks at the lower levels of the tree and varies according to the three strategies. Algorithm 1 describes the *root_task_generation* step.

Algorithm 1: $generate_root_tasks$(Tree tree,Key q)

```
k← get_replications(tree,h)
solved← 0 // zero solved so far
for i = 1 to k do
    w ← choose_worker()
    h← tree.height // tasks at the root level
    t← create_task(tree,h,q,w)
    submit_task(t,response_handler)
end
```

4.1 Leaf-Only Aggregation Strategy

In this technique, a job is replicated to K workers. A worker performs the index search by descending the tree from the root to the leaf based solely on this workers own decisions or assessments to find the best match to the query item. All workers responses and final results are aggregated only at the leaf level. Algorithm 2 gives an outline of the leaf-only aggregation strategy.

Algorithm 2: $handle_leaf_only_aggregation$(Task t)

```
if t.level ≠ 0 then // task at non-leaf level
    tree ← get_subtree(t.response,t.tree)
    t← create_task(tree,t.level-1,t.q,t.w)
    submit_task(t,handle_leaf_only_aggregation)
else
    solved++
    if solved == k then // all workers finished
        result = ← aggregate_all_results
        return result
    end
end
```

4.2 All-levels Aggregation Strategy

An alternative search strategy is to move down the tree level by level. The path from the root to the final search position is collectively determined by the crowd as follows. The search is started at the root where K_0 tasks are generated at this level. After all workers complete their tasks at the root level, their answers are aggregated, say by majority voting, and the child node that corresponds to the item with the highest vote is selected. Let this node be v. We generate K_1 new tasks for v. We aggregate the answers from the K_1 workers and decide which of the children of v to go to next. We repeat the process until we reach the leaves.

There are two variations of this search technique: *uninformed* and *informed* workers. In the former, a worker is not aware of the other workers' decisions. In the latter, a worker is allowed to view a summary of the other workers' responses to the current task. Algorithm 3 provides an outline of this search procedure.

Algorithm 3: $handle_all_levels_aggregation$(Task t)

```
solved++
if solved == k then // all workers finished at
this level
    result = ← aggregate_all_results
    if t.level ≠ 0 then // task at non leaf level
        k← get_replications(tree,t.level-1)
        tree ← get_subtree(result,t.tree)
        for i = 1 to k do
            w ← choose_worker()
            t← create_task(tree,t.level-1,t.q,w)
            submit_task(t,handle_all_levels_aggregation)
        end
    else
        return result
    end
end
```

4.3 All-levels Aggregation Strategy with Backtracking

The two search procedures above, namely the leaf-only and all-levels aggregation strategies, cannot compensate for an error. Backtracking is one way to correct potentially wrong decisions of the workers. A priority queue of pointers to the unexplored nodes in the index. Unexplored nodes are ordered in the priority queue according to their height (or level) in the tree. Nodes at the same level in the B^+-tree index are ordered by the percentage of votes given to these nodes. To detect bad decisions, workers are shown progressively the leaf node bounds of the current sub-tree at hand. If the workers indicate that the position of the query key falls outside the range of the keys in leaf nodes, then the current sub-tree is abandoned and another node is picked from the top of the priority queue to resume the search.

Figure 4 gives an example of the backtracking search strategy. In Figure 4(a), a palm-tree of fanout 2 is used to index keys 1 through 8. Figure 4(b) illustrates the steps of traversing the tree. The search starts at the root node A. Node B is at the top of the priority queue withas 60% of the workers indicate this node. The priority queue keeps track of other alternatives besides B. In next step we show the range (i.e., the min and max) of the keys in the sub-tree rooted at B, that is from 1 to 4. Assume that the workers indicate that the query key is larger than 4; then we have an error. Hence, we backtrack to node C. We insert in the queue node D. Algorithm 4 outlines the search steps.

Algorithm 4: $handle_backtrack_all_levels_aggregation$(Task t)

```
solved++
if solved == k then // all workers finished at
this level
    result = ← aggregate_all_results
    if result outside range of current subtree then
        tree ← pop_queue()
    else
        tree ← get_subtree(result,t.tree)
    end
    push_queue(other voting alternatives)
    if t.level ≠ 0 then // task at non-leaf level
        k← get_replications(tree,t.level-1)
        for i = 1 to k do
            w ← choose_worker()
            t← create_task(tree,t.level-1,t.q,w)
            submit_task(t,handle_backtrack_all_levels_aggregation)
        end
    else
        return result
    end
end
```

5. ANALYSIS

In this section, we study how to set the parameters of the palm-tree index: **tree order** and **cost distribution**. We also introduce the main performance metrics of the palm-tree index, mainly, **error** and **cost**.

5.1 Performance metrics

The location of a key is the rank of the key in the ordered list of keys at the leaf level of the tree. Let q be a query key with a ground truth location, say loc_{truth}, at the leaf level, and loc_{ret} be the retrieved location using one of the search strategies.

DEFINITION 5.1. ***Error*** *E is the distance between ground truth location and retrieved location of the query key*
$E = \mid loc_{truth} - loc_{ret} \mid$

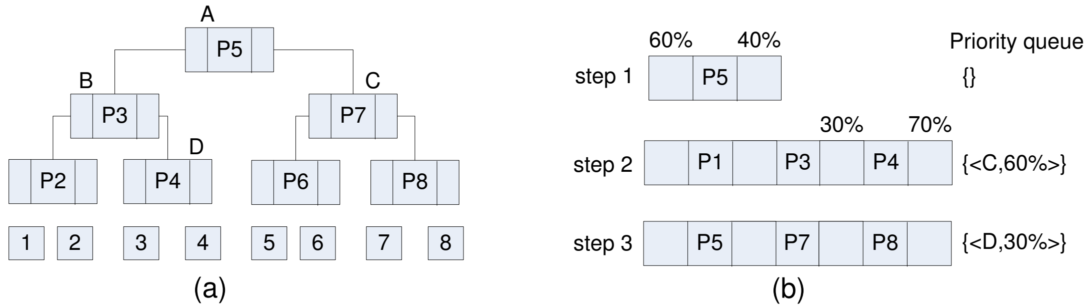


Figure 4: Backtracking all-levels aggregation strategy.

DEFINITION 5.2. ***Cost*** *C is the total number of tasks to complete a job.*

5.2 Tree order and height

The order (fanout) of a regular B^+-tree index is usually constrained by the size of the disk page. In contrast, in the palm-tree index, the order of the B^+-tree depends on the ability of workers to process at once (in a single task) a specific number of keys (see Section 3).

Notice that increasing the order of the tree increases a worker's probability of making a wrong decision at a given node. However, from a different point of view, in order to get a correct final result, correct decisions must be made at all levels of the tree. Increasing the tree order reduces number of levels required to be correct and hence reduces the final probability of error.

5.3 Cost selection

We adopt majority voting to aggregate workers decisions. Therefore, increasing the number of replicated tasks increases the quality of the aggregated decisions. However, there is almost a saturation point beyond which, increasing the number of replicated tasks would be of little, if any, benefit.

5.4 Cost distribution

Assume that there is a cost bound, say C, to search the index. An important issue is how this cost is distributed among tasks in terms of the number of replicas per tree level as tasks are assigned to workers. In the leaf-only aggregation strategy, every worker has to perform exactly h tasks from the root level to the leaf level. Hence, we can only have $k = \lfloor \frac{C}{h} \rfloor$ workers at most.

In the all-levels aggregation strategy, we propose two techniques to distribute tasks among level, namely *even* and *probabilistic* distribution. In the *even* distribution technique, every level in the palm-tree index is assigned $\lfloor \frac{C}{h} \rfloor$ tasks. *Even* cost distribution assumes equal importance of all the levels of the palm-tree index. However, tree levels are of different importance e.g., when a wrong decision is made at the higher levels (closer to the root level) of the palm-tree index, it would result in a higher final error. This is not the case when wrong decisions are made at the lower levels of the palm-tree index. On the other hand, wrong decisions are more likely to occur at the lower levels than at the higher levels. The reason is that spacing in-between the keys within a node decreases as we descend the tree.

To accommodate for the effect of nodes level on error severity is to replicate tasks in a way that is proportional to the *expected distance error per level*.

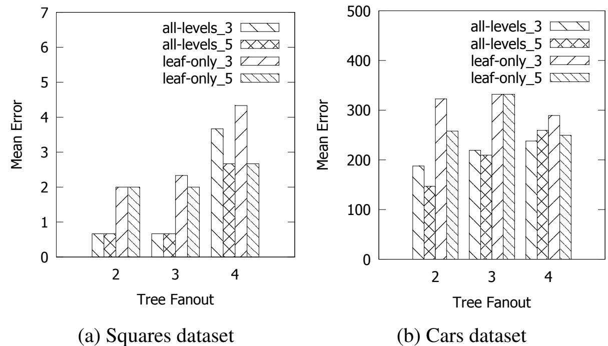
DEFINITION 5.3. ***Probability of distance d error at level l*** *(P_{dl}) is the probability of deviating d branches from the ground truth branch at level l.*

We estimate the final error that would result from a distance d error at level l to be $d \times f^{l-1}$. For example, a distance 1 error at level 1 (i.e., the leaf level) deviates the final result from the correct one by distance 1. A distance 1 error at level h (i.e., the root level) deviates final result about the number of leaf keys in a child subtree of the root node, that is f^{h-1}. We calculate the *expected distance error at level l (EE_l)* to be $\sum_d d \times P_{dl} \times f^{l-1}$.

Techniques for cost distribution for level-by-level voting with backtracking is left for future investigation.

6. EXPERIMENTAL EVALUATION

In order to test the palm-tree index and its search strategies, we implement a web site to submit tasks to workers. This helps in further studying the problem under controlled settings. Two datasets are used in the experiments; the first dataset is a set of 200 square images of different sizes. The second dataset is a set of 1300 used cars images with desired selling prices. A custom-built web crawler is used to collect the dataset of cars from a website of used car ads.

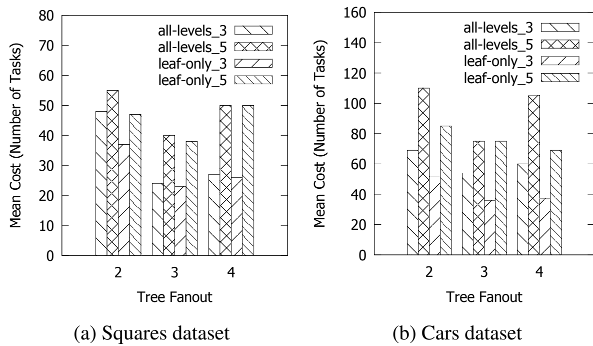


(a) Squares dataset (b) Cars dataset

Figure 5: Mean error while changing tree fanout.

Figures 5 and 6 give the experimental results of the mean error rate and mean cost for the uninformed all-levels aggregation and leaf-only aggregation search strategies. The parameters of the palm-tree index are set as follows. The fanout ranges from 2 to 4 and task replication is set to 3 and 5.

Figure 5 gives the mean error rates for the two search algorithms when applied to the cars and squares datasets. The error rate for either algorithm is much higher on the cars dataset than on the



(a) Squares dataset (b) Cars dataset

Figure 6: Mean cost while changing tree fanout.

squares dataset. The reason is that it is easier for ordinary users to compare images of squares based on their sizes than it is to compare cars based on their prices. It is also clear that all-levels aggregation outperforms leaf-only aggregation in both datasets. The reason is that in the leaf-only aggregation strategy, if a worker makes a wrong decision at a higher level of the tree, this error affects more significantly his/her answer in the lower levels of the tree.

In the all-levels aggregation, if a worker makes a wrong decision, then other workers can usually compensate for it (We use the majority principle). It is also evident from Figure 5 that increasing the number of workers reduces the error rate.

Figure 6 gives the mean cost needed to perform a query on the palm-tree. The cost for search algorithms is higher on the cars dataset than on the squares dataset. The reason is that the size (and accordingly tree height) of the cars dataset is larger than the that of the squares dataset. It is evident that increasing the number of replications increases the cost needed. It is also evident that increasing the tree fanout generally reduces the cost as the height of the tree decreases. One technique used to reduce the cost is to allow workers to stop at not leaf levels if they agree that a key at a non-leaf node matches the search key.

We are in the process of implementing more extensive experiments that will include larger crowds and other search strategies (i.e., informed all-levels aggregation and all-levels aggregation with backtracking). We plan to study the effect of cost distribution alternatives (even and expected distance errors). We also plan to study the quality of sorts and joins using the palm-tree index.

7. RELATED WORK

Lately, there has been an increasing interest in crowdsourcing. Several crowd-powered database systems, e.g., [6, 15, 12, 11] have been proposed to use human intelegance to perform database operations. These operations mainly focus on sorts and joins, e.g., [11], counting, e.g., [10], and max-finding, e.g., [7, 16]. Other works focus on algorithms to answer top-K, e.g., [4], skyline queries, e.g., [9], and spatial queries, e.g., [8].

The problem of optimizing the number of questions needed to find a set of target nodes within a general directed acyclic graph (DAG) using yes/no questions is studied in [13]. However, the B^+ tree within the palm-tree index posseses more specific properties than a general DAG (i.e., balanced height, and multiple ordered keys within nodes) that requires more specific strategies for tree search and task aggregation methods (Section 4) than in the case when yes/no questions are used.To our knowledge, this is the first work to address the problem of indexing with the crowd.

8. CONCLUSION

In this paper, we study the problem of crowdsourcing-based indexing. We motivate the problem with several application scenarios. We propose a taxonomy of the problem and highlight its main challenges. We propose techniques for index construction and querying. We intend to complete this study with an extensive analysis and experimental evaluation.

Our current work focuses on one-dimensional indexing. We plan to study other variations of crowdsourced indexing, such as multidimensional indexing, spatial and spatio-temporal indexing and fuzzy indexing.

9. REFERENCES

[1] The Amazon Mturk website. `https://www.mturk.com`, 2013.

[2] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with CrowdSearcher. In *WWW*, pages 1009–1018, 2012.

[3] A. Bozzon, M. Brambilla, S. Ceri, M. Silvestri, and G. Vesci. Choosing the right crowd: expert finding in social networks. In *EDBT*, pages 637–648, 2013.

[4] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*.

[5] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, pages 469–478, 2012.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[7] S. Guo, A. G. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, pages 385–396, 2012.

[8] L. Kazemi and C. Shahabi. Geocrowd: enabling query answering with spatial crowdsourcing. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 189–198. ACM, 2012.

[9] C. Lofi, K. El Maarry, and W.-T. Balke. Skyline queries in crowd-enabled databases. In *EDBT*, pages 465–476, 2013.

[10] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. In *PVLDB*, pages 109–120, 2013.

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

[12] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. CIDR, 2011.

[13] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: it's okay to ask questions. *VLDB Endow.*, 4:267–278, 2011.

[14] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.

[15] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5, 2012.

[16] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 989–998, 2012.