When Optimizer Chooses Table Scans: How to Make Them More Responsive

Lijian Wan

University of Massachusetts, Lowell
lwan@cs.uml.edu

Tingjian Ge

University of Massachusetts, Lowell
ge@cs.uml.edu

ABSTRACT

Recent studies show that table scans are increasingly more common than using secondary indices. Given that the optimizer may choose table scans when the selectivity is as low as 0.5% with large data, it is important to make initial query results faster for interactive data explorations. We formulate it as a query result timeliness problem, and propose two complementary approaches. The first approach builds lightweight statistics and judiciously determines an access order to data blocks for a given query. The second approach performs adaptive microscopic tuple reordering online without relying on pre-built statistics. Our systematic experimental evaluation further verifies the efficiency and efficacy of our approaches.

CCS CONCEPTS

• Information systems → Database query processing;

KEYWORDS

table scans; responsive; microscopic reorder

ACM Reference Format:

Lijian Wan and Tingjian Ge. 2018. When Optimizer Chooses Table Scans: How to Make Them More Responsive. In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*, October 22–26, 2018, Torino, Italy. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3269206.3271773>

1 INTRODUCTION

Recent studies show that table scans are increasingly more common than using secondary indices. In a SIGMOD'17 paper [13], Kester et al. use the term “crossover selectivity” to refer to the threshold τ of the query predicate selectivity (i.e., the fraction of tuples that satisfy the predicate), above which table scan performs better and below which index is better. This is shown in Figure 1. From the bottom table in Figure 1, it is remarkable that the crossover selectivity has been decreasing dramatically over the past four decades (from about 12% to around 0.5%), mostly due to the development of both hardware and database software. As indicated in the plot of Figure 1 (the experimental results of [13]), this crossover selectivity is a function of other variables such as table size—regardless, it is low, typically in the range between 0.4% and 1.2%. Consequently, table

Year	1980 (disk)	1990 (disk)	2000 (disk)	2010 (disk)	2016 (mem)
Crossover	12.4%	6.2%	5.0%	0.1%	0.6%

Figure 1: The crossover selectivity (query selectivity boundary) between table scan and using a secondary index is becoming very low over the years [13].

scans are increasingly more common in practice, which deserve the due attention.

Furthermore, in reality, a suitable index is often unavailable for a particular query. First, data analytics today deal with a large number of ad hoc and dynamic queries, and it is impossible or too costly to foresee and maintain all needed indices—as found in the deployment of BigQuery [22] and Cloudera Impala [21], one would need to build too many indices to cover all expected queries, and their size could become larger than original data. Second, indices are only applicable to simple predicates (equality and range predicates), but not on many other filters, such as those involving multiple columns, arithmetic expressions, user-defined functions, OR, NOT, “!=” operators, and many subqueries.

We consider an interactive data exploration setting. For example, a scientist has an initial hypothesis, issues an initial query to data, adjusts the hypothesis based on its result, issues another query, and the iterations go on. With table scans or not, for large datasets, query engines can easily take minutes or hours to answer the simplest of queries [19]. Slow responses can severely impact the scientist’s productivity, engagement, and even creativity [16, 18].

Particularly for table scans where the query selectivity is low (as low as 0.5%, or even lower if a suitable index is unavailable), since qualified tuples are typically not evenly distributed in the table, the wait time before seeing the first result tuples can be very long. Our goal is to improve the query result responsiveness, given that a table scan is used by the optimizer. This gives the interactive data explorer (the scientist in the example above) a better experience—the user may get information from the first few result tuples and abort the execution (or use a “limit k ” option). In addition, our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM '18, October 22–26, 2018, Torino, Italy

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6014-2/18/10...\$15.00

<https://doi.org/10.1145/3269206.3271773>

Figure 2: Main motivating factors and our approaches.

work could also accelerate the downstream operators in the query processing pipeline (if any), such as the progress estimate in online aggregation/join [8, 10, 15]. We summarize the motivating factors and the approaches of our work in Figure 2.

In this paper, we first formally define the notion of query result timeliness. Then we propose two complementary approaches (Figure 2). Our first approach builds *lightweight statistics* offline, analogous to histograms in traditional optimizers. Then online for each query, we determine the optimal order of data block traversal using an A* style algorithm [9]. Our second approach resorts to a *microscopic reorder* (i.e., piecemeal reorganization) of the tuples in the table. We perform a *competitive analysis* [17] of our online algorithm, and show that it converges very fast to a Markov chain's (optimal) stationary distribution using the Coupling Lemma [14].

Related Work. The lightweight statistics in our first approach is more similar to an optimizer histogram [3] than an index or a cracking index [11, 12] (which is essentially a half-built index amortizing index costs over query processing), as our statistics is maintained offline. The scope of our work falls in situations when the optimizer decides to do a table scan—improving its responsiveness. Our approaches can co-exist with indices, i.e., we simply provide additional features of table scans for the optimizer to choose from. In particular, our second approach moves blocks in the table, which is common in DBMS systems with a textbook solution [23]: using a sparse auxiliary B+ tree where a logical block-id is the key and the physical block-id is the value. All regular indices point to the logical block-ids—thus when table blocks are moved, only this auxiliary tree needs to update the mapping, and regular indices do not change. We implement this in the experiments.

Our work bears a resemblance to *data skipping* [25]. However, [25] assumes one knows the query workload in advance and solves a database design problem. It learns the query features and decides how to partition the data at database design time. Data analytics often deals with ad hoc and dynamic queries; we do not assume we know the queries beforehand. In particular, our second approach automatically adapts to dynamic query distributions. Our work may co-exist with data skipping, which does the initial database design, and our work further automatically adapts to the query/data changes during the deployment.

Our work is different from online aggregation (e.g., [8, 10, 15]), a major goal of which is to estimate a confidence interval or progress of the answer as the long query processing goes. Unlike our work, it does not attempt to purposefully change the order of traversing data blocks so as to minimize the time delay in seeing tuples that satisfy query predicates—when there is a long wait time for initial qualified tuples, the progress will remain 0% or very low. In this paper, we focus on lower levels of the query-tree plan, i.e., scanning the data source relations with the purpose of encountering result tuples earlier. If there are down-stream operators in the pipeline, such as online aggregations/joins, having source tuples earlier will

help accelerate the progress estimate. The details of this direction are beyond the scope of this paper and are left as future work.

2 PROBLEM STATEMENT

We consider scanning a set of tuples T for answering queries. The tuples of T are in n blocks $b_1, b_2, \dots, b_n$, which may either be stored on disk or in memory. The term “block” in this paper refers to a unit of data we schedule to read/write in its sequential order, and is different from the traditional DBMS blocks—we call the latter *pages* to differentiate them. Thus, a block in general contains one or more pages, and signifies the granularity of traverse order; any traverse order can be represented as a permutation of the n blocks.

For a query q that has a boolean predicate P , let the records that satisfy P have a distribution D over the n blocks, where $D = \{r_1, r_2, \dots, r_n\}$, and r_i is a non-negative integer indicating that there are r_i records in block b_i satisfying P . We say that D is the *qualified-tuple distribution* (QTD) of query q .

A *traverse path* is a permutation π of the n blocks, indicating the visit order. That is, $\pi(1), \pi(2), \dots, \pi(n)$ is a sequence of block ID's in visit order (block $b_{\pi(1)}$ is visited first while $b_{\pi(n)}$ the last). For a query q and each π , the *timing function* $t_q(\pi, i)$ is given, which is the time it takes to get to block $b_{\pi(i)}$, following the traverse path π , where each block in the path is read and processed for query q during the traverse. Thus, both query processing time for each block and seek/latency times are included. We also say that $t_q(\pi, i)$ is the *required time of π up to step i* (where each step processes a block for a query q).

Definition 2.1. (Result Timeliness) The result timeliness of a query q with respect to a traverse path π is $\tau(q, \pi) = \sum_{i=1}^n r_{\pi(i)} f_q(\pi, i)$, where $r_{\pi(i)}$ is a tuple count from the qualified-tuple distribution D of q , and $f_q(\pi, i)$ is the state value at step i (typically a function of the required time $t_q(\pi, i)$).

For instance, a simple state function is $f_q(\pi, i) = \frac{1}{t_q(\pi, i)}$. Since $t_q(\pi, i)$ monotonically increases with i , this $f_q(\pi, i)$ monotonically decreases with i . Intuitively, a qualified tuple's “value” decreases over time; to have a higher result timeliness, a block that contains more qualified tuples should be placed earlier in the path π . In this paper, we assume that $f_q(\pi, i)$ decreases whenever $t_q(\pi, i)$ increases. When the context is clear, we often drop the subscript and write $t(\pi, i)$ and $f(\pi, i)$ for brevity.

Everything else being the same, visiting the data blocks in decreasing result-tuple-number order gives the maximum result timeliness. However, as the case with disk blocks, changing traverse order typically entails a timing function change as well. We then have the following optimization problem.

Definition 2.2. (Traverse Path Problem) Given a data structure with sequential blocks $b_1, \dots, b_n$, the qualified-tuple distribution D for a query q , and the timing function $t(\pi, i)$ for any traverse path π , find a traverse path π^* that gives the maximum result timeliness for query q .

Our second approach is to reorder/reorganize a small number of blocks in an online manner as we answer queries, in order to increase result timeliness for future queries.

Definition 2.3. (Online Reorganization Problem) Given an initial placement of n blocks $b_1, \dots, b_n$ of a data structure in the

storage, and a potentially infinite sequence of queries $q_1, q_2, \dots$ revealed to us one by one in an online manner, our goal is to adaptively reorganize the data blocks online, so as to improve the total query result timeliness values of all queries.

3 APPROACH 1: LEARNING QUALIFIED-TUPLE DISTRIBUTIONS

In Approach 1, we first need to have an estimated qualified-tuple distribution (QTD) D for a query q before we choose a traverse path. We begin with obtaining lightweight statistics—the distributions of attribute-value intervals. This can be done for any number of attributes that occur often in query predicates. From these precomputed distributions, we estimate the QTD D for a query q . The basic idea is that we first acquire a sample of the records, sort it, and partition the sample into β equidepth intervals. For each interval, we obtain its distribution over the n blocks. This algorithm is presented in GETATTRIBUTE DISTRIBUTION.

Algorithm 1: GETATTRIBUTE DISTRIBUTION(T, a)

Input: T : the table, a : the attribute
Output: the distributions of various value intervals of a

```

1 obtain a uniformly random sample of tuples  $S$  from  $T$ 
2 sort the tuples in  $S$  on attribute  $a$ 
3 split  $S$  into  $\beta$  equidepth intervals  $I_1, \dots, I_\beta$  based on sorted  $a$ 
4 for  $i \leftarrow 1, \dots, \beta$  do
5   for each block ID  $j \leftarrow 1, \dots, n$  do
6     initialize a counter  $c_{ij} \leftarrow 0$ 
7 for each block  $b_j \in T$  ( $1 \leq j \leq n$ ) do
8   for each tuple  $t \in b_j$  do
9      $c_{ij} \leftarrow c_{ij} + 1$  if  $t.a \in I_i$  ( $1 \leq i \leq \beta$ )
10 return  $I_1, \{c_{1,1}, \dots, c_{1,n}\}, \dots, I_\beta, \{c_{\beta,1}, \dots, c_{\beta,n}\}$ 
```

In lines 1-3, obtaining splitters is essentially the first step of Samplesort [7]. The *oversampling ratio* $\frac{|S|}{\beta}$ gives a tradeoff between efficiency and accuracy, and is well studied in [7] (following previous work, we choose a default ratio of 10). In lines 4-6, we initialize βn counters, as a distribution for each interval comprises n counters, one for each block. Subsequently in lines 7-9, we scan each tuple in each block, and increment the corresponding counter. The time complexity of the algorithm is $O(\beta n + |T| \log \beta)$, and the space complexity is $O(\beta n)$.

The distributions from Algorithm 1 are light-weight statistics, reminiscent of the histograms in query optimizers [4]. However, in a histogram, each interval has a count, while in our case each interval has a distribution over the blocks. Given the predicates of a query q , the QTD can be estimated from the distributions of attributes. For instance, consider a range predicate “*pickup_datetime* BETWEEN t_1 AND t_2 ”, where t_1 and t_2 are date/time constants. Suppose that we have built distributions for attribute *pickup_datetime* with β intervals, and that $[t_1, t_2]$ spans r full intervals, θ_1 fraction of the interval I_b around t_1 , and θ_2 fraction of the interval I_e around t_2 (without loss of generality). Then the estimated QTD is a linear combination of the counts. That is, the count at block i ($1 \leq i \leq n$) is the sum of those from the r full intervals, θ_1 fraction of that from I_b , and θ_2 fraction of that from I_e . Like the estimation using

histograms, it may be inaccurate for some predicates and “update statistics” is performed from time to time due to data updates.

Compressing Attribute Distributions. The amount of space that the distributions take may be significant. We devise a method to compress the distributions. The basic idea is as follows. Recall that a distribution of an attribute consists of β distributions D_i ($1 \leq i \leq \beta$), each of which consists of n counts. We normalize the D_i 's by dividing each count by the number of records $|T|$. Thus, these βn normalized values form an empirical probability distribution. For each D_i , we try to merge the frequencies of adjacent blocks into a pair (*block_interval*, *frequency_sum*). For decompression, an individual block's frequency can be recovered approximately through dividing the *frequency_sum* by the number of blocks in *block_interval*.

Over all β value intervals, we can represent the compressed distribution as a number of (*value_interval_id*, *block_interval*, *frequency_sum*) triples, where $1 \leq \text{value_interval_id} \leq \beta$. The accuracy loss of individual blocks is quantified by variation distance [14]. Our algorithm requires a discretization—let the precision of variation distance be a small constant ϵ (e.g., $\epsilon = 0.001$). Then a variation distance constraint can be represented as an integer multiple δ of ϵ . For example, $\delta = 100$ and $\epsilon = 0.001$ correspond to a maximum variation distance of $\delta\epsilon = 0.1$. Thus, we seek to minimize the number of triples as described above, subject to a maximum variation distance constraint δ .

Define $f(i, d)$ as the minimum number of triples needed to represent the distribution from value intervals i to β with a variation distance budget of d . Further define $g(i, j, d)$ as the minimum number of triples needed to represent the block-distribution for interval i and blocks j to n , with a variation distance budget of d . Then:

$$f(i, d) = \min_{d_1 + d_2 \leq d} g(i, 1, d_1) + f(i + 1, d_2) \quad (1)$$

$$g(i, j, d) = 1 + \min_{j \leq k \leq n} g(i, k + 1, d - \frac{1}{2} \sum_{j \leq l \leq k} \left| p_{il} - \frac{p_{ij} + \dots + p_{ik}}{k - j + 1} \right|) \quad (2)$$

where p_{ij} is the normalized frequency (c_{ij}). In Equation (2), we merge blocks i to k as the first triple; thus the recursive call deals with the rest of the blocks with a reduced variation distance budget. From Equation (2), we can first have a dynamic programming algorithm to compute all $g(i, j, d)$ values with a complexity of $O(\beta \delta n^2)$. Then Equation (1) entails another dynamic programming algorithm to compute all $f(i, d)$ values with a complexity of $O(\beta \delta^2)$. Thus, our original problem is to solve $f(1, \delta)$ with an overall complexity of $O(\beta \delta (\delta + n^2))$.

4 APPROACH 1: FINDING TRAVERSE PATH

First Try: Optimal Traverse Path. For a given query q , based on the qualified-tuple distribution learned in Section 3, we solve the Traverse Path problem. We first show that this problem is NP-complete. For succinctness and clarity, the proofs of all the lemmas and theorems are in our technical report [27].

THEOREM 4.1. *The decision version of Traverse Path problem is NP-complete.*

The NP-completeness result in Theorem 4.1 indicates that it is unlikely to devise an algorithm that finds the optimal traverse path efficiently for large n . In fact, n does not need to be large even for

Figure 3: Illustrating the OPTIMALTRAVERSEPATH algorithm. (a) is the priority queue Q (typically implemented as a heap), where the root (with the maximum key) is popped each time. Zooming into the root node we see 5 fields: π , already committed list of blocks, the *head* of the current scan cursor, τ_1 , the exact result timeliness of π , τ_2 , an upper bound of the result timeliness of future blocks, and the key for Q , $\tau_1 + \tau_2$. (b) shows the lineage of the root node in (a). It started from π being one block 8, and τ_1 always increases. The key is always an upper bound of the final result timeliness.

a very large dataset—we may just need to have a large block-size, which, however, coarsens the granularity of traverse order. We first propose an algorithm that does aggressive A* style [9] pruning and bounding and finds the optimal traverse path. The reader may refer to [27] for some background on A* search.

The basic idea is a “best-first” search through the space of traverse paths. An intermediate traverse path π has a *finished* part in the beginning (a sequence of blocks already scheduled), and an *unfinished* part at the end (other blocks to be scheduled). For the finished part, we can calculate its exact contribution to the result timeliness. For each block i in the unfinished part, we obtain an upper bound of its contribution to the result timeliness through an upper bound of block i ’s $f_q(\cdot, \cdot)$ value (Definition 2.1), which in turn is associated with a lower bound of block i ’s timing function $t_q(\cdot, \cdot)$. This is so since a longer time delay only lowers the result timeliness. The end of the last block j in the finished part indicates the current scan position (i.e., “disk head” if we read from disk). Thus, a lower bound of time to an unfinished block i is simply the time it takes to go directly from the end of block j to block i . Our work and experiments target on-disk data. Modulo a constant factor, this lower bound can be approximated by the distance between the two blocks. The idea of A* style pruning is that we always retrieve a candidate intermediate traverse path π with the maximum result timeliness upper-bound through a priority queue. If π still has unfinished blocks, we expand π by one block (i.e., schedule/finish it), and put the expanded paths into the queue. If π does not have unfinished blocks, the algorithm just returns π , since its result timeliness is the exact value and other paths in the queue cannot be better (even though they use optimistic upper bounds). The algorithm is shown in OPTIMALTRAVERSEPATH.

We explain the algorithm while illustrating it with an example in Figure 3. In lines 2-7, we add into Q initial traverse paths with only a single block in the finished part. The *head* in line 3 indicates the disk “head”, i.e., the current scan position. τ_1 in line 5 is the exact result timeliness contribution from the finished part of π . The RESULTTIMELINESS function invoked in line 6 expands the current traverse path π by one block i , and returns the new τ_1 (exact timeliness contribution from finished part) and τ_2 (upper-bound of timeliness contribution from unfinished part). This function first adds to τ_1 the value contributed by the next block i , which, according to the

Algorithm 2: OPTIMALTRAVERSEPATH (D)

Input: D : qualified-tuple distribution for query q
Output: optimal block traverse path π^*

```

1 initialize a priority queue  $Q$ 
2 for  $i \leftarrow 1, \dots, n$  do
3    $head \leftarrow 0$ 
4    $\pi \leftarrow$  empty path
5    $\tau_1 \leftarrow 0$ 
6    $(\tau_1, \tau_2) \leftarrow \text{RESULTTIMELINESS}(\pi, head, i, \tau_1, D)$ 
7   add  $(\pi + i, i, \tau_1, \tau_2)$  into  $Q$  with key  $\tau_1 + \tau_2$ 
8 while (true) do
9   pop  $(\pi, head, \tau_1, \tau_2)$  from  $Q$ 
10  if  $\tau_2 = 0$  then
11    return  $\pi$ 
12  for each  $i \in \text{unfinished}(\pi)$  do
13     $(\tau'_1, \tau'_2) \leftarrow \text{RESULTTIMELINESS}(\pi, head, i, \tau_1, D)$ 
14    add  $(\pi + i, i, \tau'_1, \tau'_2)$  into  $Q$  with key  $\tau'_1 + \tau'_2$ 
```

Result Timeliness definition, is r_i (block i ’s qualified-tuple count, available in D) divided by the distance between the end of block *head* and block i . Now treating block i as the new current scan position (*head*), we can compute the new timeliness upper-bound of each unfinished block based on its qualified-tuple count and distance to block i ; the sum of these values is the new τ_2 . Finally, RESULTTIMELINESS returns the new τ_1 and τ_2 .

In line 7, $\pi + i$ denotes the new intermediate traverse path with block i appended to the end of the finished part of π . The structure of each element in the priority queue Q is a 4-tuple $(\pi, head, \tau_1, \tau_2)$, and the key for priority is $\tau_1 + \tau_2$, an upper bound of the timeliness of π . When π is a complete path, $\tau_2 = 0$ and τ_1 is the exact value of result timeliness. One such node added to Q is illustrated as the top node in Figure 3(b), where π has a single block 8, which is also the *head*. Its key is $\tau_1 + \tau_2 = 3.1 + 19.5 = 22.6$.

In lines 10-11, $\tau_2 = 0$ implies that the path π is complete. Lines 12-14 expand π by appending block i . Figure 3(b) illustrates the history of this expansion.

Final Version: Approximation + A*. We now relax the optimality of the traverse order with the goal of a more efficient algorithm for a larger n value, hence smaller block-size and finer granularity of traverse order. The basic idea of our algorithm is illustrated in Figure 4. In general, suppose we intend to find an optimal traverse path in the interval from block i to block j . We break up this task into two dichotomic order steps at some optimal point, say block k , in the interval. In addition to this split point, there is also an option of “left-right” or “right-left”, i.e., recursively visiting the interval [block i , block k] as the first step, or [block $k + 1$, block j] as the first step. Figure 4 shows the latter choice.

One complication here is that the computation of our target function, result timeliness, carries state from one step to the next, namely the timing function’s value and the location of the current scan-position. As such, we write a recursive function incorporating as parameters: *pre*—the previous scan-position (last block) before jumping into this interval (blocks i to j), *post*—the block/scan-position

Figure 4: Illustrating the algorithm to find an optimal dichotomic order traverse path.

the plan for this interval will end up at, and t_0 —the thus-far incurred time (as determined by the timing function) before jumping into this interval. Figure 4 illustrates *pre* and *post*.

We are now ready to define the following recursive functions:

$$f(i, j, pre, post, t_0) = \max_{i \leq k < j} g(i, k, j, pre, post, t_0) \quad (3)$$

The function f denotes the maximum result timeliness for the interval from blocks i to j , when the scan-position is previously at block *pre*, and will finish at block *post*, and the previously incurred time delay (before moving to this block interval) is t_0 . g is an auxiliary function, and is defined as follows. If $post > k$, then

$$g(i, k, j, pre, post, t_0) = \max_{i \leq post_1 \leq k} [f(i, k, pre, post_1, t_0) + f(k+1, j, post_1, post, t_0 + t(i, k, pre, post_1, t_0))] \quad (4)$$

Otherwise,

$$g(i, k, j, pre, post, t_0) = \max_{k < post_1 \leq j} [f(k+1, j, pre, post_1, t_0) + f(i, k, post_1, post, t_0 + t(k+1, j, pre, post_1, t_0))] \quad (5)$$

In Equation (3), block k is the dichotomic order split point as illustrated in Figure 4. In Equations (4-5), we have two cases depending on the relative positions between block *post* and block k . Since *post* must be the last block, if $post > k$, the order must be block interval $[i, k]$ followed by $[k+1, j]$ (Equation (4)), and the other way around otherwise. Figure 4 illustrates the situation in Equation (5). The variable $post_1$ denotes the final block resulted from Step 1's sub-interval (which is the *pre* of Step 2). In addition, each $f(i, j, pre, post, t_0)$ entry is associated with a *time function* entry $t(i, j, pre, post, t_0)$ recording the time it takes for this interval; recursively, it is the sum of the time function values from the two sub-intervals. This time function is used in Equations (4-5). Equations (3-5) give us a dynamic programming algorithm, and our goal is $\max_{1 \leq post \leq n} f(1, n, 0, post, 0)$, the maximum result timeliness for block interval $[1, n]$ where the last visited block (*post*) can be anywhere. To analyze the complexity, it is useful to realize that the number of distinct t_0 is $O(n^2)$, because any traverse path has at most n blocks to process, and the transition time between two blocks is $O(n)$ depending on the distance between them. Thus, this algorithm has a total cost of $O(n^8)$. This is a fundamental reduction in search space compared to OPTIMALTRAVERSEPATH.

Further cost reduction. Furthermore, we observe that the search space is quite sparse. For example, block *pre* has to be outside the interval $[i, j]$, and *post* has to be inside; otherwise the f function value is $-\infty$. Thus, we only fill in the dynamic programming function table as needed (top-down). We still use A* search (analogous to OPTIMALTRAVERSEPATH), but an intermediate traverse path that is put in the priority queue $\mathcal{Q}$ in general has a sequence of block intervals (rather than single blocks as in OPTIMALTRAVERSEPATH).

An interval is refined from left to right. Unless an interval contains only one block, an upper-bound of result timeliness for each block in an interval is derived through its shortest distance to the boundary of the previous block-interval in the traverse path. Therefore, essentially this A* search guides us the top-down dynamic programming as needed, with significant pruning and bounding.

5 APPROACH 2: MICROSCOPIC REORDER/REORGANIZATION

In this section, we devise a completely different approach. We do not build any statistics before processing a query. But after processing a query and having the “ground-truth” qualified-tuple distribution, we may move some records around in order to enhance the result timeliness for subsequent queries. First, the following is quite intuitive:

LEMMA 5.1. *We achieve the maximum result timeliness for a query q when the n data blocks $b_1, \dots, b_n$ are placed sequentially on disk with qualified-tuple distribution satisfying $r_1 \geq r_2 \geq \dots \geq r_n$.*

Based on Lemma 5.1, our reorganization algorithms will attempt to place data blocks in decreasing qualified-tuple cardinality order (approximately) for the pool of queries $\mathcal{Q}$ that arrives in an online manner.

Principle of Locality for data. One of the most important properties of data repeatedly used in Computer Science and databases is the Principle of Locality [5, 6]. *Locality* means that the probability of reference for recently referenced pages is higher than the average reference probability [6]. As a matter of fact, this is the principle underlying the designs of database buffer management [6] and data caching; it is used to manage record-flows between levels of memory [5]. As Denning puts it, “during any interval of execution, a program favors a subset of its pages, and this set of favored pages changes slowly” [5]. Data locality applies to multiple concurrent transactions and users [20]. For example, in a data source of news and blogs [24], there is temporal locality of querying news/blogs of hot topics within a time period. In an online movie database such as IMDb [2], queries from many users frequently access the current popular movies, which dynamically, but relatively slowly, change over time.

Baseline: Complete Reorganization. We leverage the Principle of Locality to reorder table blocks after processing a query, so that subsequent queries benefit from it and improve their responsiveness. As a first cut, we start with a simple algorithm that does complete reorganization as follows: after we process a query, we get n pairs of (*block_id*, *qualified_tuple_cnt*), which is the ground-truth as we have processed it. Then we sort these n pairs in descending order of *qualified_tuple_cnt*. Finally, we read the blocks in this order, and write them consecutively in a new file in the storage.

This is an *online algorithm*, as we do not know all the queries and the *qualified_tuple_cnt* of each block for each query in advance. But rather, the queries are revealed to the algorithm one by one. Thus, to analyze how good our online algorithm is, it is useful to analyze its *competitive ratio* [17]. That is, we compare it with an all-powerful ideal offline algorithm that dictates all queries and their *qualified_tuple_cnt* distributions. Some background on online algorithms and competitive analysis can be found in [27].

We model the cost of the reorganization by the number of *pair-wise adjacent-block swaps* needed to transition all blocks in sorted order. This intuitively makes sense as it reflects the effort required to correct the order. This happens to be the number of *inversions* in the original block list, where an *inversion* is a pair of blocks (not necessarily adjacent) that are in the inverse of sorted order, as shown in Lemma 5.2.

LEMMA 5.2. *The number of adjacent-block swaps required to correct an order is the same as the number of block-pair inversions.*

In our competitive analysis, in addition to (1) the algorithm cost itself, we must also consider (2) the cost of query result timeliness. A competitive analysis (i.e., comparison with the ideal offline algorithm) requires a uniform metric of algorithm “cost”. As discussed above, part (1) is depicted by the number of block-pair inversions for a query.

For a single metric, we define *disturbance ratio* (k) as a multiplicative factor indicating how “disturbed” the user is to a block-pair inversion (i.e., seeing a block with fewer result-tuples before seeing one with more). Thus, if there are in total p block-pair inversions, then part (2) of the cost (for result timeliness) is $k \cdot p$. Clearly, if there are no inversions, cost (2) is 0; the more inversions we have, the higher cost (2) is. Thus, the disturbance ratio k is application specific and is tailored to a specific user with respect to his/her sensitivity to result timeliness, relative to the algorithmic cost of data reorganization after getting results (cost (1), which is p). We have the following result.

THEOREM 5.3. *The competitive ratio of our online reorganization algorithm is $\frac{k+1}{\min(1, \frac{k}{2})}$, where k is the disturbance ratio. Moreover, any deterministic online reorganization algorithm, as well as any randomized online algorithm with the adaptive offline adversary model, has a competitive ratio at least k .*

The result in Theorem 5.3 is very informative. First, our deterministic online reorganization algorithm has a competitive ratio that is a bounded constant, and does not depend on the data size or number of blocks n . Note that it is competing with an all-powerful offline algorithm, which we do not even know. Second, this competitive ratio performance is nearly optimal—if $k \geq 2$ it is $k + 1$ (while the best one can hope for is $\geq k$); otherwise it is $\frac{2(k+1)}{k}$.

Piecemeal Reorganization. The complete reorder may have a significant overhead; we now devise a *piecemeal* reorganization (i.e., microscopic reorder) algorithm. The key idea is that we partition the n data blocks into equal-sized chunks, each of which has size α . Thus, we have $m = \lceil \frac{n}{\alpha} \rceil$ chunks. For each query q , after we process q and know the qualified-tuple distribution over the blocks, we stochastically pick *one chunk* (following some distribution, as detailed in the algorithm) and swap its blocks with blocks elsewhere, based on the qualified-tuple distribution of q . In this way, we show that the qualified-tuple distribution over time closely traces that based on the query pool Q . The algorithm is presented in PIECEMEALSIFT.

We explain the algorithm while looking at an example in Figure 5. In this (simplified) example, each chunk has 3 blocks (i.e., $\alpha = 3$). In line 1 of the algorithm PIECEMEALSIFT, we initialize a set of *clean* chunks to be empty. “clean” here means that the blocks in the chunk

Algorithm 3: PIECEMEALSIFT (T, Q)

Input: T : the table with m chunks,
 Q : the pool of queries to answer
Output: nothing (T is re-organized over time)

```

1  $clean \leftarrow \emptyset$ 
2 for query  $q \in Q$  do
3   if  $clean \neq \text{whole } T$  then
4      $c_{new} \leftarrow \text{next chunk in } T \text{ not in } clean$ 
5     if  $clean = \emptyset$  then
6        $p_{new} \leftarrow 1$ 
7     else
8        $p_{new} \leftarrow \theta$ 
9   else
10     $p_{new} \leftarrow 0$ 
11   for  $i \leftarrow 1, \dots, |clean|$  do
12      $c_i \leftarrow i\text{'th chunk in } clean$ 
13      $p_i \leftarrow (1 - p_{new}) \cdot (\frac{1}{2})^i$ 
14     if  $i = |clean|$  then
15        $p_i \leftarrow 2p_i$ 
16   randomly pick a chunk  $c$  based on  $p_{new}, p_1, \dots, p_{|clean|}$ 
17   if  $c = c_{new}$  then
18     swap into  $c$  top- $\alpha$  blocks (based on  $q$ ) not in  $clean$ 
19      $clean \leftarrow clean \cup c$ 
20   else
21     swap into  $c$  the blocks that should be in (based on  $q$ )

```

have been “swapped” at least once — for example, the 3 blocks in Chunk 1 position should be swapped with the top-3 blocks with the highest qualified-tuple counts based on the current query. The *clean* set starts from Chunk 1 and only extends to an adjacent chunk each time until all chunks are clean.

The algorithm loops through each query (line 2). If the *clean* set has not extended to the whole table yet (lines 3-10), we set the probability p_{new} that a new (adjacent next) chunk will be added to *clean*. It is either 1 (if no chunk is clean) or θ (a tradeoff between the speed of swapping a new chunk versus updating a top/clean chunk according to the current query). By default, we use $\theta = 0.5$ in this work. p_{new} is 0 once all chunks have been cleaned once.

Lines 11-15 allocate the remaining probability $1 - p_{new}$ to each existing clean chunk. The idea is that we give more probability to more “important” chunks. A chunk with a smaller index (i.e., towards the left in Figure 5) should have more qualified-tuples; hence we update it more often based on the current query. The probabilities we allocate to each clean block thus follow an exponential series $\frac{1}{2}(1 - p_{new}), \frac{1}{4}(1 - p_{new}), \dots$, except for the last clean chunk, for which we need to double its probability from this series (lines 14-15), so that all these probabilities sum to $1 - p_{new}$. For example, for the four clean chunks in Figure 5, chunks 1 to 4 get probabilities $\frac{1}{2}(1 - p_{new}), \frac{1}{4}(1 - p_{new}), \frac{1}{8}(1 - p_{new})$, and $\frac{1}{8}(1 - p_{new})$, respectively. These 4 probabilities, together with cleaning a new chunk probability p_{new} , add up to 1.

Line 16 then randomly picks a chunk position based on the distribution we just get. If this choice happens to be extending a new chunk to *clean* (line 17), we swap into this new chunk the top- α

Figure 5: Illustrating the PIECEMEALSIFT algorithm. Here each chunk has 3 blocks, and clean set has 4 chunks. A stochastic choice now is to grow clean by one more chunk. The 3 blocks at chunk 5 location are swapped with the chunks not in clean but with the top-3 qualified-tuple counts for current query.

blocks with the highest qualified-tuple counts that are not already in the *clean* set (line 18). Otherwise, in line 21, we just update the clean chunk c by swapping into it the α blocks that should be in those positions (based on the qualified-tuple count distribution of the current query). These α blocks may come from either the *clean* set or outside of it. Note that we treat it differently depending on whether the chunk is already in *clean* set or not. The intuition is not to spoil a block already in the *clean* set by swapping it with a random “stranger” block originally located in c_{new} . This is required in our theoretical proof for fast convergence of the algorithm’s behavior to a desired distribution (Theorem 5.5).

Figure 5 shows an example for the case of extending the *clean* set by a new chunk (i.e., chunk 5) for the current query q . The top-3 blocks with the highest qualified-tuple counts (w.r.t. q) outside the *clean* set are identified (three green blocks in Figure 5) and swapped with the blocks originally located in chunk 5. The *clean* set now has 5 chunks, and the algorithm advances to the next query.

We have the following result. Some background on Markov chains, mixing time, and coupling [14] is in [27].

THEOREM 5.4. *Let a permutation of the n data blocks during the PIECEMEALSIFT algorithm be a state. Let each query q be a step. Then the states (i.e., permutations) of the n data blocks form a Markov chain. Moreover, this chain has a unique stationary distribution that is the same as the probability distribution of permutation of n blocks generated by the current query pool Q .*

We further show that, not only the algorithm behaves like a Markov chain converging to a desired stationary distribution, but it also converges very fast — called *rapidly mixing* in Markov chain coupling theory [14].

THEOREM 5.5. *The Markov chain of Theorem 5.4 is rapidly mixing,*

$$\text{with a mixing time } \tau(\epsilon) \leq \frac{m + \ln \frac{1}{\epsilon}}{\theta} + \sqrt{\left(\frac{m + \ln \frac{1}{\epsilon}}{\theta}\right)^2 - m^2}.$$

A Markov chain is called *rapidly mixing* if its mixing time $\tau(\epsilon)$ is polynomial in $\log \frac{1}{\epsilon}$ and the size of the problem. Thus, our chain is rapidly mixing. Theorem 5.5 gives us great confidence in the performance of the algorithm. Furthermore, note that the current query pool Q is really a function over time, i.e., $Q(t)$. Thus, the algorithm is adaptive to the query stream over time, since it continuously reorganizes as queries come in.

6 EXPERIMENTS

Datasets and Setup. We use the following datasets: (1) **The New York City taxi trip data** [28]. The decompressed and normalized trip data is about 30GB, containing the information of all taxi trips in NYC in 2013. It has 14 attributes including information such as medallion number, vendor ID, pick-up or drop-off date, time and location, passenger count, trip time, and trip distance. (2) **TPC-H benchmark data** [26]. The commonly used TPC-H benchmark data have been chosen to have broad industry-wide relevance. We use dbgen to get 20GB lineitem data, which has attributes such as order key, quantity, extended price, discount, ship date/mode, and receipt date.

We implement all the algorithms presented in this paper in Java. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

Experimental Results. Inspired by a project that uses Google BigQuery [1] on the NYC taxi trip data, we issue the following queries on the taxi dataset:

```
Q1: SELECT medallion
FROM trip
WHERE (HOUR(TIMESTAMP(pickup_datetime))
      BETWEEN ? AND ?)
      AND (DAY(TIMESTAMP(pickup_datetime))
      BETWEEN ? AND ?)
      AND (MONTH(TIMESTAMP(pickup_datetime))
      BETWEEN ? AND ?)

Q2: SELECT vendor_id
FROM trip
WHERE LEFT(pickup_longitude,7) BETWEEN ? AND ?
      AND LEFT(pickup_latitude,6) BETWEEN ? AND ?
GROUP BY vendor_id

Q3: SELECT medallion
FROM trip
WHERE INTEGER(passenger_count) = ?
```

Q1 queries information of any taxis whose pick-up times are within some interval. Q2 similarly retrieves counts of trips by vendors whose pick-up locations are within some range, while Q3 selects based on the passenger count (“?” is randomly chosen from 5, 6, and 7). These queries could be issued by taxi companies or even customers. At each time-step t , we choose one of the query templates uniformly at random. For Q1’s parameters, we pick a random-length interval within two weeks around the month, day, and hour of the time t when the query is issued (and the year is 2013, as in the dataset). This is to simulate real-life environments where the Principle of Locality takes effect, but the sets of frequently accessed data change over time. For each Q2, we select its location parameters randomly from 20 regions in the dataset, and change this set of regions every minute. The functions in the predicates of these queries are taken from BigQuery [1]. The diverse set of functions is one of the reasons for which one may not be able to use indices and table scans are used.

Note that, for the performance and result timeliness of the algorithms, what really matters is where the qualified tuples are, rather than the specific queries themselves. For TPC-H data, we issue the following queries:

```
Q4: SELECT l_receiptdate
FROM lineitem
WHERE l_shipdate BETWEEN ? AND ?
      AND l_shipmode = ?

Q5: SELECT l_linenum, l_quantity
FROM lineitem
WHERE l_orderkey IN (?, ?, ?)

Q6: SELECT l_suppkey
```

```

FROM lineitem
WHERE l_extendedprice > ?
GROUP BY l_suppkey

```

Like for the taxi data, at each time-step t we choose one of the query templates randomly. For Q4 we pick the $l_shipdate$ interval in a similar manner as we do for Q1's $pickup_datetime$ interval. For Q5, we randomly pick $l_orderkey$ values from an interval of size 1,000,000, while the interval shifts by 300,000 after each query. The reason for the data access pattern migration is the same as for the taxi data. Q6 queries items at the high-price range (top 5%).

The first approach of our solution is to learn the distributions of relevant attributes over blocks, from which we obtain the optimal traverse path of a query. Thus, in our first experiment, we evaluate the efficiency of building/learning the distributions. Starting with the Taxi dataset, we learn the distributions for the attributes required in the predicates of queries Q1 to Q3. In Figure 6, we show the execution times for two attributes, $pickup_datetime$ (pd) and $pickup_longitude$ (pl), varying the number of partitioned blocks n , while the total data size is always 30GB (thus block size varies). Our default value for the number of attribute value intervals (β) is 10. In Figure 6 we also show the cost of compressing these two attributes' distributions, where the allowed variation distance is $\delta = 0.1$ (with precision $\epsilon = 0.001$).

Figure 6 indicates that the execution time for learning attribute value's distributions is independent of the number of partitioned blocks n . This is because it only depends on the table/data size, which does not vary. For a large table of 30GB, the learning time is about 2 to 3 minutes, which is feasible, as it is offline and performed infrequently. On the other hand, the compression of distributions has a cost that is a quadratic function of n (consistent with our analysis in Section 3), but it is much faster — orders of magnitude faster for the block sizes we consider.

Figure 7 shows the result for the TPC-H dataset's $l_shipdate$ and $l_orderkey$ attributes, where we vary the number of attribute value intervals β between 6 and 25 while fixing the number of partitioned blocks at $n = 20$. The learning time is slightly faster than that of the Trip dataset, and is independent of the number of intervals β . The compression time increases (somewhat slowly) with the number of intervals, which is consistent with our analysis. We then examine the compression ratio, shown in Figure 8 for both datasets, where we vary the allowed variation distance δ (between the compressed and the original distributions) from 0.05 to 0.15. We can see that the distribution size (i.e., the number of frequency counts we need to maintain) decreases as δ increases. This is because a greater δ permits more approximation, and hence more attribute value intervals may be merged.

After learning the attribute distributions, we use them to estimate each incoming query's QTD, from which we determine the block traverse path. In the next set of experiments, we first examine the efficiency of the two algorithms, finding the *optimal* traverse path and finding a *dichotomic* order traverse path. We first show the result for the Taxi data in Figure 9 while varying the number of partitioned blocks n . As n increases, the optimal path algorithm slows down very fast. This is because the problem is NP-complete, as analyzed in Theorem 4.1. The algorithm in general has a cost exponential to n . When $n \geq 30$, finding optimal path is too costly (slower than answering the whole query itself, which we show in

the forthcoming Figure 11); hence it is shown as “out of bounds” in Figure 9. Finding a dichotomic order path, on the other hand, is an approximation and much faster — it finishes within 3 seconds even for $n = 50$.

We repeat the same experiment over the TPC-H dataset, and show the result in Figure 10. Interestingly, finding the optimal path is feasible here compared to query processing time. Even though it is for an NP-complete problem and in general has an exponential worst-case cost, our algorithm performs an A^* search and space pruning, which may cut the cost significantly. Overall, finding optimal path is still much more expensive than finding a dichotomic order path.

In the next three figures, we show the effectiveness of the traverse paths found by the two algorithms. Figure 11 shows the query execution timing information for the Taxi dataset and Figure 12 for the TPC-H dataset. Recall that, for each dataset, we continuously run the queries with randomly chosen qualified data. In Figure 11, we show the average total query processing time and the expected time to get a result tuple (i.e., average time delay of seeing a qualified tuple since the start of query processing), for the Taxi dataset. This information is displayed for three query processing algorithms: the baseline algorithm that just does plain sequential scan of the table (marked “plain” in the figure), the optimal traverse path found by our first algorithm (marked “optimal”), and the dichotomic order path found by our second algorithm.

Figure 11 indicates that the total query processing times for the three algorithms are nearly the same — slightly over 2 minutes for the Taxi data. The expected tuple time is around 100 seconds for the plain algorithm, while it ranges from 10 seconds to 30 seconds for the optimal and dichotomic order path algorithms. As mentioned earlier, when $n \geq 30$, finding optimal path is an infeasible plan; hence we cannot show its execution times. When the number of partitioned blocks increases, the average discovery time of qualified tuples slightly decreases for the optimal and the dichotomic order path algorithms. This is because the estimated query block distribution has a finer granularity when n is greater, resulting in a slightly better plan in getting the qualified tuples earlier. Finally, the expected result tuple time for the dichotomic order path is almost as good as the optimal one, demonstrating its effectiveness. The result for the TPC-H dataset is shown in Figure 12. It further verifies our findings. The only difference with this dataset is that all the execution times are slightly faster, and that the optimal path is available for $30 \leq n \leq 50$ (unlike the Taxi data), as mentioned earlier.

In Figure 13, we further measure the *result timeliness* as defined in Definition 2.1. For each qualified tuple (i.e., where predicates are satisfied) encountered during query processing, we add $\frac{1}{t}$ to the result timeliness, where t is the time (in seconds) when this tuple is encountered since the start of query processing. Finally, this result timeliness value is divided by the total number of qualified tuples for normalization. Thus, a higher result timeliness value is more preferable as it indicates that we get results sooner. Figure 13 shows that, for both datasets, both optimal path and the dichotomic order path algorithms have nearly two orders of magnitude higher result timeliness values than the baseline “plain” algorithm that does a sequential scan of the table. The optimal path algorithm has

Fig 6 Learning distributions (Taxi) Fig 7 Learning distributions (tpc-h)

Fig 8 Compression ratio

Fig 9 Finding traverse path (Taxi)

Fig 10 Finding traverse path (tpch) Fig 11 Query result times (Taxi) Fig 12 Query result times (tpch) Fig 13 Query result timeliness metric

a slightly higher result timeliness value than the dichotomic, except that the optimal one is unavailable for the Taxi data when $n \geq 30$. This further verifies the effectiveness of our approach.

In the next set of experiments, we evaluate data reorganization. We first examine the overhead of the complete reorganization (the baseline) and the piecemeal reorganization (final solution), as compared to the query processing cost itself. The result of this is shown in Figure 14 for the Taxi dataset and Figure 15 for the TPC-H dataset. In Figure 14, we fix the number of partitioned blocks at $n = 5, 300$ and vary the chunk size parameter α between 3 and 500 (applicable to the piecemeal reorganization). In Figure 15, we vary n between 500 and 100,000 while fixing $\alpha = 10$ (the default value in subsequent experiments). Note that in the data reorganization approach, we can afford a much higher n parameter value, unlike the finding optimal traverse path approach, which is an NP-complete problem and is already infeasible when $n \geq 30$ for the Taxi data.

Figure 14 (Taxi data) shows that a complete reorganization is about 1/5 the cost of the complete query processing. While still affordable, it is much higher than that of piecemeal reorganization, which is 1 to 3 orders of magnitude faster. Moreover, a greater chunk size α entails a higher piecemeal reorganization overhead. This is because the algorithm needs to swap more blocks of data. The benefit of a greater α is that more data gets reorganized, and, as analyzed in Theorem 5.5, the Markov chain's mixing time is smaller (a greater α implies a fewer number of chunks m), i.e., the algorithm converges sooner to a preferable stationary distribution (Theorem 5.4).

Figure 15 (TPC-H data) indicates that a greater number of partitioned blocks (i.e., a smaller block size) entails a higher cost for complete reorganization. This is because the algorithm needs to collect a finer-grained block-distribution for query result, and subsequently sort more blocks. For piecemeal reorganization, when n increases at a low-value range, the cost decreases since the amount of data in a chunk to be swapped decreases. However, this is not always the case when n increases at a high-value range (50,000 or higher), since the cost of sorting the blocks based on their numbers of qualified tuples gradually dominates the total cost.

In the next two figures, we examine the effectiveness of the two reorganization algorithms. In Figure 16, we show the expected result tuple times for the three algorithms: *plain* (just a sequential scan without reorganization), *complete* (a sequential scan followed by a complete reorganization for each query), and *piecemeal* (a sequential scan followed by a piecemeal reorganization for each query). We can see that, for both datasets, complete and piecemeal reorganizations are quite effective, significantly shortening the expected result tuple time. Moreover, perhaps surprisingly, piecemeal reorganization is more effective than complete reorganization. The reason is that, although a complete reorganization reorganizes all data according to the result of the current query, it is often an overkill — as the next query may have result tuples very different from the current query, but more or less resembles a previous query. Piecemeal reorganization strikes a balance between “adjusting to trend changes” and “staying put”, and is more robust to multiple query result clusters in the data. Our experiments also indicate that

Fig 14 Reorg. overhead (Taxi)

Fig 15 Reorg. overhead (tpch)

Fig 16 Expected result tuple times

Fig 17 Result timeliness after reorg.

a wide range of block count values n works well, with the optimal between 5,000 and 50,000 (a small n gives a coarse granularity, while an n that is too large increases the overhead and makes a chunk too small for a fixed α).

Finally, fixing the number of partitioned blocks at $n=5,000$, we vary the chunk size α between 3 blocks and 500 blocks, and measure the result timeliness values for the three methods as above, for both datasets. The result is shown in Figure 17. The chunk size parameter α is only relevant for piecemeal reorganization, which is also relatively insensitive to α (a value in the range between 10 and 100 gives the highest result timeliness). Overall, reorganization improves result timeliness by about two orders of magnitude, with piecemeal reorganization slightly better than complete reorganization, due to the reasons explained above for Figure 16.

Summary. We have evaluated two approaches that address query result timeliness: (1) learning lightweight statistics offline, and (2) microscopic reorder. Both approaches improve query result timeliness by about two orders of magnitude. In approach (1), the learning is performed offline infrequently and is feasible. The optimal-order algorithm gives slightly better result timeliness than our preferred approximate solution (dichotomic order path), but it is sometimes infeasible to find the optimal one. Approach (2) does not require offline learning, but leverages the Principle of Locality. Our experiments suggest that piecemeal reorganization is usually superior to complete reorganization, since it has significantly less overhead and performs slightly better in result timeliness.

7 CONCLUSIONS

Recent studies suggest that the crossover selectivity for the optimizer to choose table scans is getting lower and lower (about 0.5%), in addition to the fact that a suitable index is very often unavailable. Thus it is important to improve the responsiveness of table scans for interactive data exploratory queries. We propose two approaches—one prebuilds lightweight statistics and the other does online microscopic reorder. Our theoretical analysis and experimental evaluations prove the efficiency and effectiveness of our two approaches in solving this timely problem.

ACKNOWLEDGMENTS

This work is supported by NSF grants IIS-1149417 and IIS-1633271.

REFERENCES

- [1] 2017. https://bigquery.cloud.google.com/table/imjasonh-storage:nyctaxi.trip_data. (2017).
- [2] 2017. <http://www.imdb.com/>. (2017).
- [3] A. Aboulmaga and S. Chaudhuri. 1999. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD*.
- [4] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *PODS*.
- [5] Peter J. Denning. 2006. The Locality Principle. *Communication Networks and Computer Systems* (2006).
- [6] W. Effelsberg. 1984. Principles of Database Buffer Management. *ACM Transactions on Database Systems* (1984).
- [7] W. Frazer and A. C. McKellar. 1970. Samplesort: A sampling approach to minimal storage tree sorting. *JACM* (1970).
- [8] Peter Haas and Joseph Hellerstein. 1999. Ripple joins for online aggregation. *SIGMOD* (1999).
- [9] Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE TSSC* (1968).
- [10] Joseph Hellerstein, Peter Haas, and Helen Wang. 1997. Online aggregation. In *SIGMOD*.
- [11] S. Idreos, M. Kersten, and S. Manegold. 2007. Database Cracking. In *CIDR*.
- [12] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. In *VLDB*.
- [13] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *SIGMOD*.
- [14] David Levin, Yuval Peres, and Elizabeth Wilmer. 2009. *Markov chains and mixing times*. American Math. Soc.
- [15] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*.
- [16] R. B. Miller. 1968. Response time in man-computer conversational transactions. In *ACM fall joint computer conference, part I*.
- [17] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized algorithms*. Cambridge University Press.
- [18] Barzan Mozafari. 2017. Approximate Query Engines: Commercial Challenges and Research Opportunities. In *SIGMOD*.
- [19] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. 2017. Snappydata: A unified cluster for streaming, transactions, and interactive analytics. In *CIDR*.
- [20] J. Rodriguez-Rosell. 1976. Empirical Data Reference Behavior in Data Base Systems. *Computer* (1976).
- [21] John Russell. 2013. *Cloudera Impala*. O'Reilly Media, Inc.
- [22] Kazunori Sato. 2012. An Inside Look at Google BigQuery. (2012).
- [23] A. Silberschatz, H. F. Korth, and S. Sudarshan. 2010. *Database System Concepts (6th edition)*. McGraw-Hill.
- [24] A. Sun, M. Hu, and E. Lim. 2008. Searching Blogs and News: A Study on Popular Queries. In *SIGIR*.
- [25] L. Sun, M. Franklin, S. Krishnan, and R. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *SIGMOD*.
- [26] TPC. 2017. <http://www.tpc.org/tpch/>. (2017).
- [27] Lijian Wan and Tingjian Ge. 2018. http://www.cs.uml.edu/~ge/paper/timeliness_tech_report.pdf. (2018).
- [28] Chris Whong. 2017. http://chriswhong.com/open-data/foil_nyc_taxi/. (2017).