Workload-Aware Cache Management of Bitmap Indices

Julia Kaeppel
Mathematics and Computer Science
University of Puget Sound
Tacoma, WA, USA

Jason Sawin
Computer and Information Sciences
University of St. Thomas
St. Paul, MN, USA

David Chiu
Mathematics and Computer Science
University of Puget Sound
Tacoma, WA, USA

ABSTRACT

Big-data management systems must handle multiple concurrent queries over multi-dimensional data sets. To achieve high throughput, such systems could implement various techniques to avoid redundant computations and data fetches. One such approach is to cache a subset of the query results and reuse these results to (partially) fulfill future query requests. This approach can be quite effective for query-at-a-time processing. However, we suspect that even greater performance is being left on the table if queries are only optimized in isolation, and that higher throughput can be extracted through a systematic examination of the relationships between queries in a given workload.

This paper describes a framework that captures inter-query relationships to reveal increased opportunities to exploit caching. We present a heuristic used for scheduling queries and a novel workload-informed cache replacement policy. When these methods are applied in combination, our system is able to extract impressive speedup of the total execution time of batches of queries, using only modest cache sizes. In this paper we show that the proposed replacement algorithm easily outstrips the performance of the classic algorithms FIFO and LRU. Under certain conditions, our system was able to achieve roughly 2 to 4 time speedup over these traditional replacement schemes.

ACM Reference Format:
Julia Kaeppel, Jason Sawin, and David Chiu. 2023. Workload-Aware Cache Management of Bitmap Indices. In *IEEE/ACM 10th International Conference on Big Data Computing, Applications and Technologies (BDCAT '23), December 4–7, 2023, Taormina (Messina), Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3632366.3632386

1 INTRODUCTION

Achieving high throughput over data-intensive processing workloads is among the key challenges in the management of big data. One method of enabling efficient filtering and processing of large data sets is to summarize the underlying data using bitmap indices and their suite of related algorithms [39]. Bitmaps are a 2-dimensional array of bit values built over the underlying data to enable a wide range of selection and filtering operators to be applied efficiently. To do this, common data transformation expressions (*e.g.*, select, project, join, etc.) can be equivalently converted into fast bitwise expressions such that, when carried out over bitmaps, can greatly reduce data processing time. The processing performance achievable through bitmap indices explains why they can be found in many of today's big data applications, databases, and warehouses [19, 27, 35–37].

While bitmaps can achieve sublinear-time processing for many types of queries, their performance is known to greatly degrade as data dimensionality increases. Thus, accelerating bitmap processing has garnered a great deal of attention, spurring the creation of new bitmap-processing algorithms [8, 10, 17, 40], support of distributed/parallel systems [4, 29], among many other approaches. To this end, we previously designed and implemented a bitmap caching framework that specializes in accelerating query processing over high-dimensional bitmap indices. Our caching system stores metadata alongside cached query results, *e.g.*, expressions describing each corresponding cached item. By exploiting the available metadata, we demonstrated that a provably optimal set of cached results could be identified in log-linear time and reused to help satisfy future queries.

Our existing bitmap caching framework, however, had assumed an interactive workload environment, in which queries are processed as soon as they arrive to minimize response times. Big data analytical workloads, on the other hand, are generally less concerned with the response times of individual queries in favor of optimizing for throughput [34]. When queries are processed in isolation, the dependencies *between* them become obscured. We posit that inter-query dependencies within query sets could be exploited to improve overall cache performance and achieve higher throughput of batched query workloads.

This paper reports on the study of various techniques applied to the cache management of bitmap indices to improve the overall execution time of batched queries. We examine a two-pronged approach: First, we propose a scheduling heuristic to reorder queries inside each batch prior to their dispatch for execution. We also propose a novel cache-replacement algorithm, *Least Aggregated Coverage*, that is informed by the cost and dependency analysis of the batched query set.

This paper makes the following contributions:

- We consider the problem of maximizing query throughput of workload batches. Our framework performs a static analysis of a set of enqueued query requests to determine semantic interrelationships that inform multiple decision points on upkeeping a bitmap cache that produces high hit rates within the batch.
- We propose a simple yet provably optimal query scheduling algorithm. The query-dispatch sequence determines the recent state of the cache, which greatly impacts the performance of future queries.

BDCAT '23, December 4–7, 2023, Taormina (Messina), Italy

ACM ISBN 979-8-4007-0473-4/23/12…$15.00
https://doi.org/10.1145/3632366.3632386

- We propose a novel cache-replacement algorithm, *Least Aggregated Coverage*, which evicts the cached result(s) with the least perceived value for accelerating future queries.
- We test our system using the TPC-H benchmark. Our results show that our scheduling algorithm produces a 1.75× speedup over dispatching queries using their natural times of arrival. Furthermore, when using our cache-replacement policy in tandem with our scheduler, we were able to extract 2.85× to 3.41× speedup over the use of classic replacement policies FIFO and LRU, respectively.
- On a modest 1 GB cache, our algorithms can achieve performance that is only slightly slower (36%) than the use of an infinite sized cache.

The remainder of this paper is organized as follows. Section 2 covers the basics of bitmap indexing and processing. Section 3 details the proposed models and algorithms to optimize usage of our cache. A detailed experimental design and the discussion and analysis of findings are given in Section 4. Related works are summarized in Section 5, and we conclude our results in Section 6.

2 BACKGROUND

This section provides a brief overview of bitmap index creation and query processing.

A bitmap index is created by first selecting a subset of a database relation's attributes. The domains of these attributes are then partitioned to form a set of bins. A bin can represent a discrete value, such as a single name, or a range of values, such as ages between 30 and 50. Each indexed attribute for each tuple in the relation is then discretized into the appropriate bin. This process places 1 in the bin that specifies the tuple's attribute value. As long as the bins' ranges do not overlap in some way, all other bins for that tuple receive a value of 0. This process creates Y bit vectors, where Y is the number of bins. Each bit vector has X rows, where X is the number of tuples in the underlying relation. The set of all bit vectors forms an $X \times Y$ binary matrix or bitmap. Each row in the bitmap is associated with a pointer to the underlying tuple's location on disk.

The binary format of a bitmap index allows for fast hardware-enabled logical operations to aid in query processing. In its simplest form, bitmap query processing is $v_i \circ v_j = r$ where v_i and v_j are bit vectors, $\circ$ denotes a bitwise logical operation, and r is the result. For example, consider an employee relation where an `age` attribute is discretized into four bins: $bin_1 = [18, 30)$, $bin_2 = [30, 40)$, $bin_3 = [40, 50)$, and $bin_4 = [50, \infty)$. The number of rows needed to be fetched from disk for the query:

```
SELECT * FROM employee as e WHERE e.age > 40
AND e.age < 60
```

is reduced by performing the bitwise operation $b_3 \vee b_4 = result$ where b_3 and b_4 are the bit vectors associated with bin_3 and bin_4, respectively. Every row in $result$ that contains a 1 corresponds to a tuple that needs to be retrieved from disk for further candidate checking. All other rows on disk can be ignored.

Various forms of bitmap indices have been used to aid in processing many types of queries, including point queries [23, 30], range queries [29, 38, 41], joins [25], skyline queries [12], among others. Our work focuses on range queries, though we believe it is easily extendable to other query types. Bitmap range query processing takes the form $z = x_1 \vee x_2 \vee \ldots \vee x_n$, where x_i is a bit vector representing attributes value within the desired range, and z is the result vector indicating the tuples to be retrieved from disk. A simple iterative algorithm can be used to solve range queries. First, z is initialized to $z \leftarrow x_1$, and then the operation $z \leftarrow z \vee x_j$ is repeated for all $j \mid 2 \leq j \leq n$. In spite of the improvements to query-processing time that bitmaps can provide, it is a well-known problem that performance degrades for high-dimensionality data, in which n is assumed to be large.

The next section introduces our framework, which uses cached partial query results to significantly reduce the overhead of range queries applied to high-dimensional data sets.

3 SYSTEM DETAILS

In this section, we present the overall system design and details on the models and algorithms we have developed. Figure 1 presents a diagram depicting our complete system. This architecture is an expansion of our previously introduced bitmap caching framework [28]. Our old framework accepts single dynamically issued queries in the form of ordered pairs of bitmap bit vector IDs $(start, end)$, where $start \leq end$. These pairs encode sequential range queries of the form $b_{start} \vee b_{start+1} \cdots \vee b_{end}$. Upon receiving a request, the system exploits the associative nature of range queries by probing its cache for results of queries that were entirely within the range of $[start, end]$. From its current cache configuration, it builds an optimal set of partial results that maximally covers the new query request. It then fetches all the bit vectors not covered by cached results and coalesces the two sets. The new resulting vector is then added to the cache.

To illustrate the function of this framework, consider the system in its initial state. The first request it receives, q_1, is $(15, 32)$. Since the cache is empty, it will have to retrieve, from disk, all the needed bit vectors $b_{15}, b_{16}, \ldots b_{32}$ and perform the sequential range query algorithm described in Section 2. It will cache the result as r_{q1}. Assume the second request, q_2 is $(12, 33)$. Since the range of $q1$ is fully contained in q_2, the system can use the cached result. It will fetch bit vectors $12 - 14$ and 33 as they are not included in r_{q1}. It will then perform $b_{12} \vee b_{13} \vee b_{14} \vee \boldsymbol{r_{q1}} \vee b_{33}$. By reusing r_{q1} the framework saves on both IO operations and query processing.

Notice that if the query order was switched in the above example, our old framework would not have been able to realize any benefit from the cache. This is because q_2 is not a subrange of q_1. Thus, the result of q_2 could not be used to aid in the processing of q_1. This paper presents enhancements to our caching framework to better leverage inter-query dependencies.

Depicted as `Queries` in Figure 1, queries submitted to the new system are initially queued instead of being immediately dispatched for execution. When the batch size reaches a specified limit, a `Query Dependency Graph` is generated over the set of queued queries. Concurrently, the batch is sent to the `Query Scheduler`, which reorders the queries in an effort to maximize cache hits.

When queries are dispatched for processing, the cache performs as before by probing for cached results for coverage and retrieving uncovered bit vectors from the disk. The new system uses a novel replacement algorithm to decide which of the existing entries to remove from the cache when its memory limit is reached.

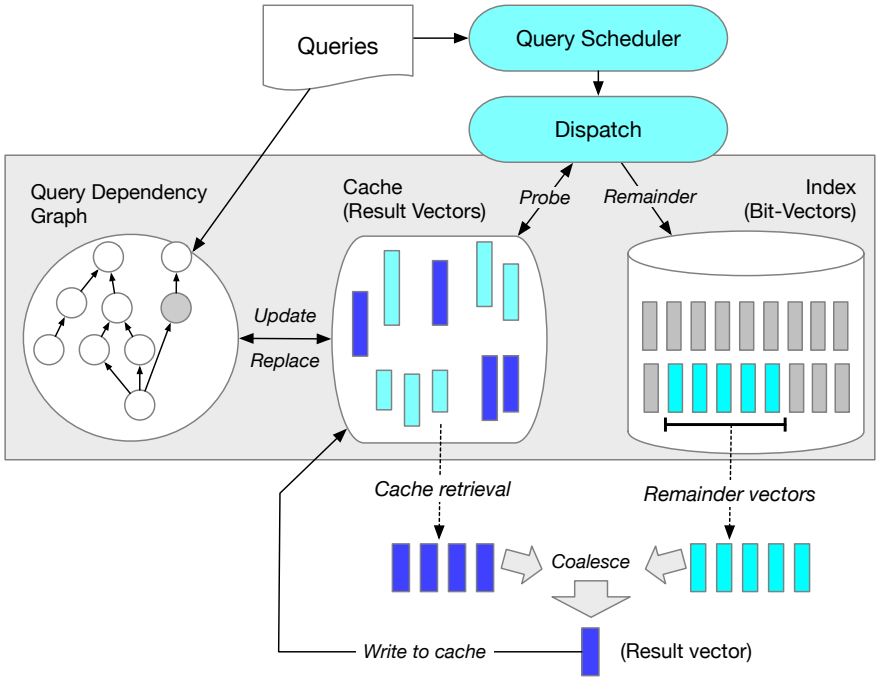


Figure 1: Overview of the Bitmap Caching Framework

Particularly, our proposed LAC replacement algorithm (described in detail below) consults the `Query Dependency Graph` to make this decision.

3.1 Shortest-First Batch Ordering

When the batch size reaches a designated threshold, our system reorders the queries for faster execution. We employ the Shortest-First query scheduling policy, which simply selects the next outstanding query with the smallest size. The rationale behind Shortest-First is straightforward: in order for a cached result to be used, it must have a smaller range than that of a subsequent query. Therefore, it follows that the queries covering smaller ranges should be executed (and cached) first.

We now show that Shortest-First produces an optimal schedule with respect to maximizing cache reuse over time. For ease of writing, we adopt the use of the following operators for expressing relationships between queries as follows:

- **Query Equality:** $q_i = q_j$. Queries q_i, q_j are equal if and only if $q_i.start = q_j.start \wedge q_i.end = q_j.end$.
- **Query Size:** $|q|$. Returns $(q.end - q.start) + 1$, which is the number of bit vectors needed to process q.
- **Coverage:** $q_i \circ\!\!- q_j$. Predicate that is true if q_i *covers* q_j and false otherwise. A query q_i is said to (partially) *cover* q_j if $q_i.start \geq q_j.start$ and $q_i.end \leq q_j.end$. Essentially, q_i is a subrange of q_j.
- **Covered Set:** $q_i \circledcirc q_j$. If $q_i \circ\!\!- q_j$, this operation returns a set of q_i's bit vector IDs $\{q_i.start, q_i.start + 1, \ldots, q_i.end\}$, otherwise it returns $\emptyset$.

We now define a few terms that will be used throughout the proof. Recall from earlier that the execution of a query produces a result vector. The cache can be modeled as a stored set of result vectors $R = \{r_1, r_2, ...\}$ that correspond to queries $q_1, q_2, ...$ We also denote R_i to be the cache's state after the execution of all queries $q_1, q_2, ..., q_{i-1}, q_i$. Specifically,

$$R_i = \begin{cases} \bigcup_{k=1}^{i} \{r_k\}, & i > 0 \\ \emptyset, & \text{Otherwise} \end{cases} \tag{1}$$

We further define the *cache coverage* $c(q, R)$ to represent the number of bit vectors that can be covered in q with respect to cache state R.

$$c(q, R) = \left| \bigcup_{r \in R} q \circledcirc q(r) \right| \tag{2}$$

where $q(r)$ refers to the query represented by result vector r, so this term is the total number of bit vectors that the current state of the cache R covers for q.

Lemma 3.1.

Given two queries q_i and q_j, if $|q_i| \leq |q_j|$ and $q_i \neq q_j$ then $\neg(q_j \circ\!\!- q_i)$.

Proof. If $q_j.end \leq q_i.start$ or if $q_j.start \geq q_i.end$, then queries are entirely disjoint and $\neg(q_j \circ\!\!- q_i)$. If the queries overlap and $|q_i| \leq |q_j|$ it is clearly impossible for both $q_j.start \geq q_i.start$ and $q_j.end \leq q_i.end$, and so by definition $\neg(q_j \circ\!\!- q_i)$. Finally, if $|q_i| = |q_j|$ and since $q_i \neq q_j$, if $q_j.start < q_i.start$ then by definition $\neg(q_j \circ\!\!- q_i)$. If $q_j.start > q_i.start$, since the queries are the same size, then $q_j.end > q_i.end$ and again by definition $\neg(q_j \circ\!\!- q_i)$. □

Theorem 3.2. *The schedule determined by Shortest-First is optimal for maximizing total cache coverage.*

Proof. The Shortest-First policy produces a schedule, $S = (q_1, ..., q_i, ..., q_{i+j}, ..., q_n)$ such that $|q_i| \leq |q_{i+j}| \; \forall j \geq 1$. Let T_S denote the total cache coverage from executing schedule S. Specifically,

$$T_S = \sum_{i=1}^{n} c(q_i, R_{i-1}) \tag{3}$$

If T_S is sub-optimal, then it would follow that there exists an alternate schedule $S' = (q_1, ..., \boldsymbol{q_{i+j}}, q_i, ..., q_n)$ in which at least one query must have been moved forward in the execution order to produce greater total coverage, *i.e.*, $T_S < T_{S'}$.

Expressing the total coverage more explicitly, we obtain:

$$\begin{aligned} T_S = {} & c(q_1, R_0) + c(q_2, R_1) + ... + c(q_i, R_{i-1}) && (4)\\ & + c(q_{i+1}, R_i) && (5)\\ & + ... && (6)\\ & + c(q_{i+j}, R_{i+j-1}) && (7)\\ & + c(q_{i+j+1}, R_{i+j}) + ... + c(q_n, R_{n-1}) && (8) \end{aligned}$$

Conversely, we can rewrite T_S to account for the reordering of schedule S'. Specifically,

$$\begin{aligned} \mathbf{T_{S'}} = {} & \mathbf{c(q_1, R_0) + c(q_2, R_1) + ... + c(q_{i-1}, R_{i-2})} && (9)\\ & \mathbf{+c(q_{i+j}, R_{i-1})} && (10)\\ & \mathbf{+c(q_i, R_{i-1} \cup \{r_{i+j}\})} && (11)\\ & \mathbf{+...} && (12)\\ & \mathbf{+c(q_{i+j-1}, R_{i+j-2} \cup \{r_{i+j}\})} && (13)\\ & \mathbf{+c(q_{i+j+1}, R_{i+j}) + ... + c(q_n, R_{n-1})} && (14) \end{aligned}$$

(For readability, all $\mathbf{T_{S'}}$ terms will be **emboldened** for the remainder of this section).

Note that all terms up to query q_{i-1} (line 9) are unchanged in sequence (and thus coverage is unchanged) and that the only affected queries range from q_i to q_{i+j} inclusive (lines 10 - 13). Line 10 is the new order position for q_{i+j}. From that position to query q_{i+j-1} (line 13), all the queries have the same coverage as in T_S with the additional coverage provided by q_{i+j}. Thus each line contains the original S cache configuration $\cup\{r_{i+j}\}$. After query q_{i+j-1}, the coverage is again the same as the cache will contain all the same result vectors as in T_S.

We now show that the difference $T_S - \mathbf{T_{S'}} \geq 0$. Note that all terms cancel except for
$T_S - \mathbf{T_{S'}} =$

$$\begin{aligned} & c(q_{i+j}, R_{j-1}) - \mathbf{c(q_{i+j}, R_{i-1})} && (15)\\ & + c(q_i, R_{q_{i-1}}) - \mathbf{c(q_i, R_{q_{i-1}} \cup \{r_{i+j}\})} && (16)\\ & ... && (17)\\ & + c(q_{i+j-1}, R_{q_{i+j-2}}) - \mathbf{c(q_{i+j-1}, R_{q_{i+j-2}} \cup \{r_{i+j}\})} && (18) \end{aligned}$$

Because S is ordered by size, we know that $|q_b| \leq |q_{i+j}| \; \forall b \in \{i, i+1, \ldots, i+j-1\}$. First, we examine when $q_b \neq q_{i+j}$. In this case, we know by lemma 3.1 that $\neg(q_j \multimap q_i)$, and so the result bit vector r_{i+j} will not add any additional coverage to q_b. Therefore $c(q_b, R_{b-1}) = \boldsymbol{c(q_b, R_{b-1} \cup r_{i+j})} \; \forall b$, and $T_S - T_{S'}$ simplifies to

$$c(q_{i+j}, R_{i+j-1}) - \mathbf{c(q_{i+j}, R_{i-1})} \tag{19}$$

Notice that $R_{i-1} \subseteq R_{i+j-1}$ and thus $c(q_{i+j}, R_{i+j-1}) \geq \mathbf{c(q_{i+j}, R_{i-1})}$ which implies $T_S - T_{S'} \geq 0$.

The more interesting case is when the promoted query q_{i+j} is equal to a least one of the queries it was moved in front of, or more precisely, $\exists d \mid i \leq d \leq i+j-1 \; \wedge \; q_d = q_{i+j}$. It is possible that q_{i+j} is equal to multiple other queries. Assume that q_d represents the first query that q_{i+j} is equal to in S'. This means that q_{i+j} provides complete coverage for q_d, and $\mathbf{c(q_d, R_{d-1} \cup \{r_{i+j}\})} = |q_d|$. Notice that for queries following r_d in S', the coverage remains unchanged as r_d and r_{i+j} provide the exact same coverage. Similarly, since d is the first query q_{i+j} is equivalent to, by lemma 3.1, the coverage for all the queries that preceded d are unchanged. Therefore $T_S - T_{S'}$ simplifies to:

$$\begin{aligned} & c(q_{i+j}, R_{j-1}) - \mathbf{c(q_{i+j}, R_{i-1})} && (20)\\ & \quad + \mathbf{c(q_d, R_{d-1})} - \mathbf{c(q_d, R_{d-1} \cup \{r_{i+j}\})} && (21) \end{aligned}$$

As established above, we know that $\mathbf{c(q_d, R_{d-1} \cup \{r_{i+j}\})} = |q_d|$. Since $r_d = r_{i+j}$ and in the S ordering r_d is executed before r_{i+j}, we know that $c(q_{i+j}, R_{j-1}) = |r_d|$. So the two equations cancel, and we simplify again to

$$\begin{aligned} & |r_d| - \mathbf{c(q_{i+j}, R_{i-1})} && (22)\\ & \quad + c(q_d, R_{d-1}) - |r_d| && (23) \end{aligned}$$

or

$$\mathbf{-c(q_{i+j}, R_{i-1})} + c(q_d, R_{d-1}) \tag{24}$$

Since $q_{i+j} = q_d$, we can perform the following substitution:

$$\mathbf{-c(q_d, R_{i-1})} + c(q_d, R_{d-1}) \tag{25}$$

Since $d \geq i$, we know $R_{i-1} \subseteq R_{d-1}$. Thus, $\mathbf{c(q_d, R_{i-1})} \leq c(q_d, R_{d-1})$ and therefore $T_S - \mathbf{T_{S'}} \geq 0 \implies T_S \geq \mathbf{T_{S'}}$.

The above approach is intuitively generalizable to any number of promotions, and thus S is an optimal schedule for maximizing overall coverage. □

The optimality of Shortest-First rests on the assumption the cache is sufficiently large, *i.e.*, enough to store the results for all n queries. In practice, this assumption often does not hold, and a cache-replacement algorithm (such as FIFO, CLOCK, LRU, etc.) must be used to evict certain entries to manage the cache size. In our experience, however, these classical replacement algorithms are suboptimal when used in conjunction with Shortest-First scheduling. To this end, we propose a new replacement policy specifically designed to work alongside Shortest-First scheduling. The next subsection describes the the algorithm to build the graph model that will be used to inform our proposed replacement policy.

3.2 Query Dependency Graph

To support our novel cache replacement algorithm (which will be described in the following subsection), we model the query workload using a dependency graph. The dependency graph is a directed acyclic graph $G = (V, E)$, where nodes $V = \{q_1, ..., q_n\}$ is a set of queries to be executed, and a set of directed edges $E = \{(q_i, q_j) : q_i \text{ non-redundantly covers } q_j\}$. That is, an edge $(q_i, q_j) \in E$ if the following conditions are met:

(1) $q_i \multimap q_j$, and
(2) $(q_k, q_j) \notin E$ such that $q_i \multimap q_k$.

Property (1) simply ensures that q_i covers q_j, while property (2) is to ensure that q_i is non-redundant, *i.e.*, q_i provides unique and maximal coverage of q_j.

Algorithm 1 BUILDGRAPH

```
1:  Input
2:      Q    Batch of queries
3:  Output
4:      (V, E)    Query dependency graph
5:  ▷ Pre-sort the queries
6:  Q ← Q.preSort()
7:  ▷ Instantiate an empty graph
8:  V ← {}
9:  E ← {}
10: for i ← 0 to Q.size() do
11:     q_i ← Q.get(i)
12:     if q_i ∉ V then
13:         maxEnd ← 0
14:         for j ← i + 1 to Q.size() do
15:             q_j ← Q.get(j)
16:             ▷ Terminate once ranges no longer overlap
17:             if q_j.start > q_i.end then
18:                 break
19:             ▷ Skip invalid queries
20:             if q_j.end > q_i.end || q_j.end ≤ maxEnd then
21:                 continue
22:             ▷ Add an edge to the graph
23:             E ← E ∪ {(q_j, q_i)}
24:             maxEnd ← q_j.end
25:             j ← j + 1
26:         ▷ Add node q_i to the graph
27:         V ← V ∪ {q_i}
28:     i ← i + 1
29: return (V, E)
```

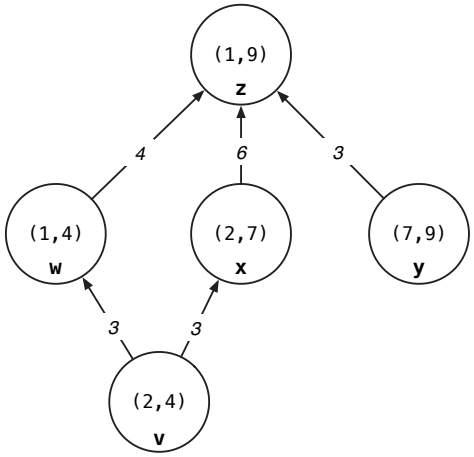


Figure 2: Example Query Dependency Graph

Figure 2 shows an example of a query dependency graph. For clarity, each node is labeled with a query expression, and each edge is labeled with the size of the query's coverage. Note in the figure that an edge from v to z is considered redundant according to the graph's definition, and is therefore removed: Although $v \multimap z$, the second edge property is not satisfied, due to the existence of x.

Algorithm 1 produces the query dependency graph based on the definition of G given above. The graph-building algorithm inputs a batch of queries $Q = \{q_1, q_2, ..., q_n\}$, and relies on a presorting step on Q (Line 6) to work efficiently. Essentially, a query q_i precedes q_j in ordering if:

$$q_i.start < q_j.start \vee (q_i.start = q_j.start \wedge q_j.end > q_i.end)$$

Figure 3 illustrates an example of this query presorting step. Each horizontal bar represents the start and end points of a query, and they are sorted in top-down order. As a first pass, the queries are grouped by their $q.start$ points. Within each group, the queries are further sorted in descending order of their $q.end$ points. This sequencing is an optimization that allows the graph-building algorithm to stop early in several cases described below.

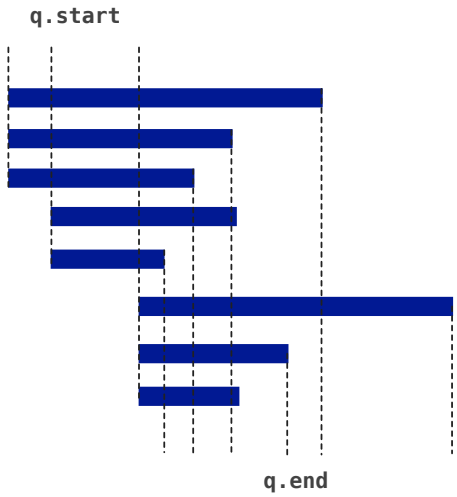


Figure 3: Query Presorting

After presorting, the algorithm examines each query iteratively. For each query q_i, an inner loop (Line 14) finds all non-redundant queries q_j that provide some cover to q_i. The conditional on Line 17 exploits the presorted order by short-circuiting out of the inner loop once q_j no longer covers q_i. The conditional on Line 18 further ensures that an edge from q_j is not added if q_j exceeds the range of q_i. To reduce the size of the graph, redundant edges are not added. This is accomplished by keeping track of the current maximum end bit vector ID of a node's edges, *maxEnd*, and only adding a new edge if it has a larger $q.end$ ID, thus providing new coverage.

To show an example of Algorithm 1, suppose batch Q is input as $Q = (v, w, x, y, w, z)$, which are the query nodes depicted in Figure 2. The presorting step would impose the sequence: (z, w, x, v, y). The duplicate pairs of $w = (1, 4)$ are placed after $z = (1, 9)$, since w's end point is smaller, which ensures that w covers query z. Furthermore, the duplicate copy of w is collapsed, since it does not add new

information to the graph. After presorting, the algorithm goes on to examine nodes in the presorted sequence, beginning with node z. Edges from w and x are added, because they non-redundantly cover z. The node v is also considered as a candidate for linking z, but because v's end point (4) does not exceed the current maximum end point of 7 (via the earlier addition of x), v is deemed redundant and the edge (v, z) is precluded. Finally, y is added as a non-redundant predecessor to z. This process repeats for all other queries in the batch (outer loop), producing the remaining edges (v, w) and (v, x). The result is the dependency graph shown in Figure 2.

The graph-building algorithm runs in $O(|V| + |E|)$, since in the worst case, every node needs to test for an edge with every other node with a greater or equal starting bit vector ID. Although this complexity is not ideal, it bears pointing out that G is generally sparse in practice. Moreover, the query dependency graph pays dividends when it is used to determine the order in which cached results are evicted/replaced, and the savings in query workload execution time easily offsets the graph-building overhead.

3.3 Least Aggregated Coverage Replacement

Least-Aggregated-Coverage (LAC) is our novel cache replacement policy designed to work efficiently with the Shortest-First scheduling algorithm. LAC uses a priority queue to order cached results based on the total amount of coverage they provide within the query dependency graph, and results with the least aggregated coverage are evicted when the cache is full.

Algorithm 2 addEntryLAC

```
1:  Input
2:      q    Query to cache
3:      G    Query dependency graph
4:      PQ   Ordered multimap of priorities and cache entries
5:  Output
6:      None
7:  ▷ Calculate the aggregated coverage of q. Serves as priority.
8:  cov_q ← |G.successors(q)| × |q|
9:  PQ.add(cov_q, q)
10: ▷ Update priorities of existing cache entries
11: for all p ∈ G.predecessors(q) do
12:     if PQ.contains(p) then
13:         cov_p ← PQ.getPriority(p) − |p|
14:         PQ.remove(p)
15:         PQ.add(cov_p, p)
```

The *aggregated coverage* of a query node is calculated by examining the count of its direct successors. If a query q node directly feeds into n succeeding nodes, then its aggregated coverage is $|q| \times n$. We assume that query results that have higher aggregated coverage implies that it is more valuable to the cache and should be kept longer in the priority queue. We adhere to the standard definition of a priority queue, in which entries with lower values imply higher priority, and in our case, queries with higher aggregated coverage will be farther from the head.

The procedure to add a result to the cache is given in Algorithm 2. The algorithm starts by obtaining all successor nodes of the given entry, q to calculate q's aggregated coverage cov_q. The new entry, along with its coverage, are then added to the priority queue. Since the cached predecessors of q were used to aid in the execution of q, they now hold less value in the cache. For each of q's cached predecessor p, the algorithm reduces p's priority by $|p|$ (Line 13). The predecessors are reinserted in the priority queue to adjust their positions (lines 14-15).

For clarification, suppose in Figure 2 that query node x is being added to the cache. It would have an aggregated coverage value of 6, since only one other query (z) is dependent on x. Furthermore, x's predecessor node v, which had an aggregated coverage of $|v| \times 2 = 6$, must reduce its aggregated coverage to 3. The reason for this reduction is because, after x's execution, the edge (v, x) is no longer useful in the current query batch, so v contributes to one fewer outstanding query.

In order to support all the required operations for LAC, a multimap with both key and value searching is used to implement a priority queue. The data structure consists of a red-black tree that uses priority values for keys (lower values imply higher priority) and stores hash sets of query nodes, as well as a hash map that uses the queries as keys and stores priority values. The red-black tree provides the priority queue functionality, while the hash map allows entries to be accessed and updated with negligible cost. With this implementation, determining whether an entry exists can be done in $O(1)$ time, and adding, accessing, or removing an entry to the queue can be done in amortized $O(\log n)$. Thus, adding an entry to the cache can be performed in $O(|E| \log |V|)$, when accounting for updating the predecessors' priorities.

4 EVALUATION AND RESULTS

This section presents a detailed experimental evaluation of our caching framework. The caching structures and bitmap processor were implemented in Java, and experiments were executed on a machine running Windows 10 Pro, equipped with an 8-core Intel Core i7-9700K at 3.60 GHz, 64 GB of RAM, and a 2 TB HDD.

4.1 Data Set and Query Workload

We used the TPC-H 2.18.0 toolkit [1] to create the data set and query workload used in our experimental studies. The TPC-H database models an e-commerce enterprise. Our experiments are carried over TPC-H's `lineitem` table. It contains 6 million rows and 16 columns. We chose the `lineitem` table due to its central role in the TPC-H benchmark. We created a large bitmap index over the table's `SHIPDATE` column due to this column's high dimensionality. The bitmap index contains 2526 bins and 6 million rows.

The TPC-H toolkit is used to generate queries over the `lineitem` table. Each query can be generated to select rows in `lineitem` given a pair of *start* and *end* `SHIPDATE` values, providing us with a set of 3, 191, 601 possible unique queries. To create each of our query batches, we randomly draw from this set of possible queries according to the desired batch size.

4.2 Experimental Setup and Metrics

Our system processes a batch of multiple queries at a time. A batch of queries are submitted to the scheduler all at once for processing. If no scheduling policy is used (labeled `sched-NONE` in our results),

then the queries are dispatched for execution in the order they arrived. When running Shortest-First scheduling (labeled `sched-SF`), the batch of queries is reordered in ascending order of size before dispatching to the query processor.

We also define the metrics used for evaluation. A *cache miss* is counted for each bitmap bit vector that is fetched for processing: A query that selects k bit vectors could result in k misses in the worst case, and as such, the *miss rate* is defined to be the total number of misses over the total number of bins requested across all queries in a batch. We also define *total time* to be the overall time elapsed to process the lifecycle of a batch of queries. That is, in addition to the query execution and cache-hit times, the *total time* includes any overhead to pre-analyze the batch as well as any cache-replacement overhead.

4.3 Evaluation of Shortest-First Scheduling

In the first set of experiments, we are interested in evaluating the effectiveness of the Shortest-First scheduling policy. We observed the execution time and total number of misses over batches of size 1000, 2500, 5000, and 10,000 queries that were generated by TPC-H. To run this experiment, we assumed an infinite-capacity cache so that cache-replacement would not be invoked, and therefore all query results are unconditionally placed in the cache for reuse. This allows us to evaluate Shortest-First in isolation.

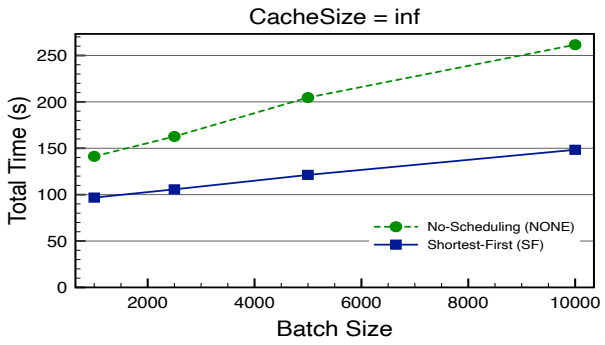


Figure 4: Performance of Shortest-First Scheduling

The results presented in Figure 4 show the total execution times per each batch size. Across the increasing batch sizes, `sched-SF` scheduling yields significant speedups over the runs in which no scheduling is applied. It is worth noting that the total times here include the scheduling overheads of `sched-SF`. The reported speedups over unscheduled batches range from 1.5× to 1.76×. Because the cache capacity is unlimited, and due to `sched-SF` scheduling having shown to be optimal, this is likely the best performance that can be achieved from a purely scheduling-only approach.

Batch Size	Scheduling Overhead
1000	0 ms
10,000	3 ms
100,000	24 ms
1,000,000	201 ms

Table 1: Overhead of Shortest-First

Next, we isolated the overhead of the `sched-SF` algorithm on varying sized query batches. We showed earlier that the `sched-SF` algorithm is trivial and can be implemented in $O(n \log n)$ time, where n is the number of queries in the batch. The results in Table 1 support this claim, as the negligible overhead of `sched-SF` seems to scale gracefully over exponentially increasing batch sizes. This is a salient result: the `sched-SF` scheduling algorithm can be safely recommended irrespective of batch sizes and (as we shall show) replacement policies.

While running the above experiments, we saw that the caches grew to substantial sizes, *i.e.*, requiring over 30 GB of data in the case of 10,000 queries. This cache size is clearly unreasonable in most practical systems settings. Next, we evaluate how our system performs under various cache replacement schemes used to maintain a fixed capacity.

4.4 Performance of LAC Cache Replacement

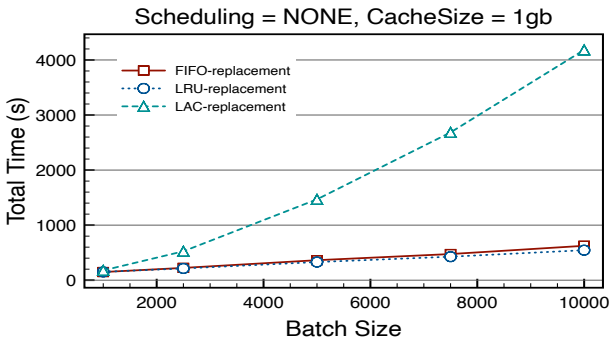
This set of experiments stress-test our cache replacement scheme. The cache capacity for this set of experiments is fixed at 1 GB, a fraction of the total cache-storage requirements for the workload. We evaluate the *Least Aggregated Coverage* replacement policy (`LAC`) and compare it against two classic replacement policies: First-In-First-Out (`FIFO`) and Least Recently Used (`LRU`).

The implementation of `FIFO` uses a linked list, resulting in $O(1)$-time operations to evict and add entries, and the implementation of `LRU` uses a min-heap priority queue. When a cached result is reused, we must update the LRU entry with a new timestamp and then re-enqueue it, which is $O(\log n)$. Likewise, when an entry is to be evicted, the queue must be heapified, also requiring $O(\log n)$ time. In contrast, `LAC` requires $O(n \log n)$ time when adding and removing entries to/from the cache, as was explained in Section 3.3.

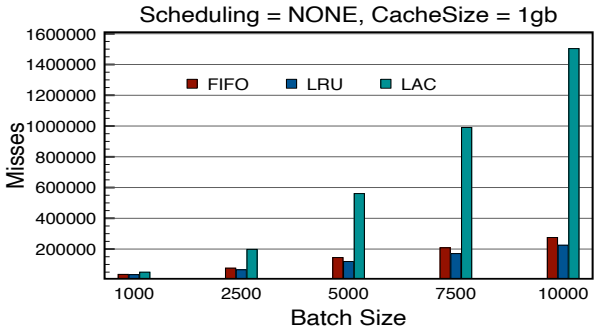
We observed that replacement is invoked early in the batch processing: On a 1 GB cache, the first eviction occurs after the ~ 350th query in the batch, which means that replacement will be invoked frequently in all batch settings. This allows us to evaluate the scheduling policies and the replacement policies in tandem. We stress-test the scalability of our algorithms under even smaller, and larger, cache sizes in a later section. We ran experiments using query batch sizes of 1000, 2500, 5000, 7500, and 10000 and measured the total time taken to execute each batch.

First, we ran the tests without scheduling (`sched-NONE`). Figure 5a shows the total times over various cache-replacement policies for unscheduled workloads. The results for `FIFO` and `LRU` replacement perform quite similarly, but as expected, our `LAC` replacement performs poorly due to the fact that it was designed specifically to reward cached entries that have greater potential for reuse. This potential, however, is exploited only through the use of a complementary scheduling policy.

Figure 5b shows the number of misses across batch sizes for the three replacement policies. Total time is strongly correlated to the number of misses. And whereas the classical replacement policies tend to keep miss rates in check for unscheduled workloads, `LAC` suffers greatly, resulting in nearly 6× the number of misses compared to either `FIFO` or `LRU`. Therefore, our `LAC` replacement scheme evicts entries on an almost random nature without a conspiring scheduling algorithm and should not be used without Shortest-First.

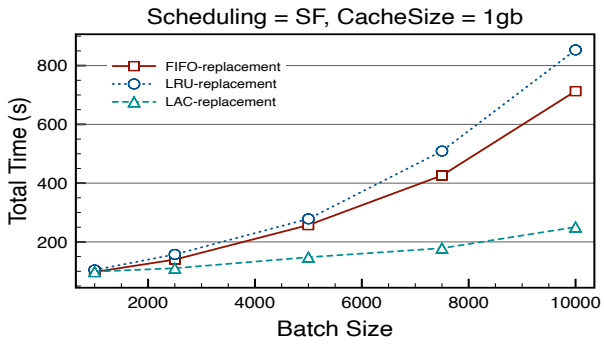


(a) No Scheduling (sched-NONE)

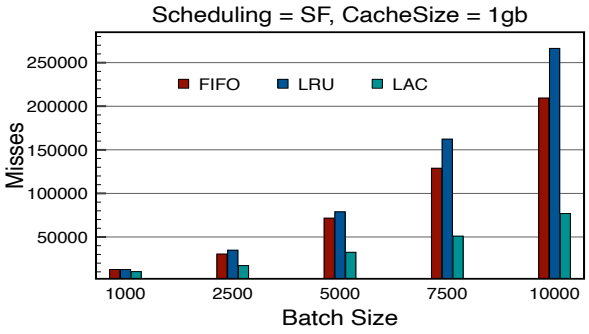


(b) No Scheduling (sched-NONE)

Figure 5: Total Times and Cache Misses (No Scheduling)



(a) Shortest-First Scheduling (sched-SF)



(b) Shortest-First Scheduling (sched-SF)

Figure 6: Total Times and Cache Misses (SF Scheduling)

We ran the same experiment using Shortest-First scheduling. The two plots in Figure 6 again show the total times and miss count. It is at first unexpected to find that FIFO outperforms LRU, which is usually not the case in practice. However, recall that when sched-SF scheduling is used, the short and singleton queries, which offer less overall reuse value, are executed and cached first. FIFO evicts these short and singleton results first, leaving the results belonging to longer queries stored for more potency when reused. In contrast, LRU rewards the results that are used more frequently, which means that the short and singleton results will tend to remain in the cache, consuming valuable space while not offering much savings in the execution of future queries.

Batch Size	% Change in Cache Misses	
	LAC vs. FIFO	LAC vs. LRU
1000	−19.90%	−18.96%
2500	−43.67%	−50.52%
5000	−54.31%	−58.95%
7500	−61.77%	−68.70%
10000	−63.75%	−71.42%

Table 2: Comparison of Miss Rates (SF Scheduling)

The strength of the LAC replacement scheme is realized when it is applied in tandem with sched-SF scheduling. Compared to sched-NONE, sched-SF scheduling does not significantly impact the execution times when using FIFO and LRU. However, our LAC policy was able to reduce the number of misses down to a fraction of both FIFO and LRU replacement schemes. Drilling deeper into these results, Table 2 shows that LAC was able to reduce the miss count by roughly 19% for smaller batches, and up to 71.42% for larger batches. This reduction corresponds to a rather significant speedup of 2.85× over FIFO replacement and 3.41× over LRU replacement.

Results in this section were limited to a 1 GB cache. We are also interested in our algorithm's performance given smaller cache sizes.

4.5 Impact of Cache Size

In the next set of experiments, we scale across different cache sizes. In addition to a 1 GB cache, we chose to experiment with two extreme size settings: a small 256 MB cache and an infinite sized cache. The total time to execute a batch of 10,000 queries are plotted in Figure 7. The LAC replacement scheme performs better than both FIFO and LRU in the 256 MB and 1 GB cache size configurations, demonstrating the consistency of LAC, when pressed with even more frequent calls for eviction. On the small 256 MB cache, LAC speeds up execution over FIFO and LRU by factors of 2.66× and 2.32× respectively. These speedups are slightly lower than respective speedups of 2.85× and 3.41× over the 1 GB cache. This is due to the fact that replacement is called more often to

maintain the smaller cache size, and that `LACAddEntry()` carries a larger overhead than either `FIFO` or `LRU`.

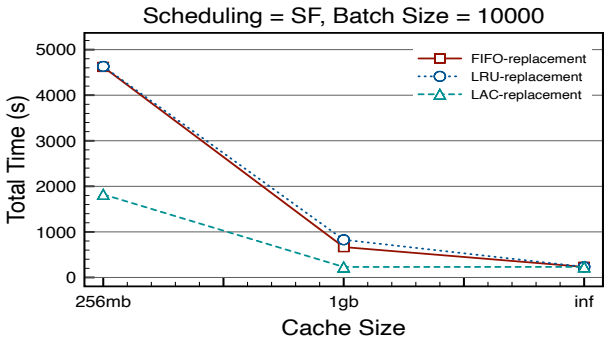


Figure 7: Total Times across Cache Sizes (Batch Size = 10000)

On the other end, an infinite sized cache does not ever invoke a replacement algorithm, so the lines converge in the plot. Interestingly, the total time to execute the 10,000 batch using infinite cache is 148, 334 ms, while the 1 GB cache using `LAC` replacement required 233, 240 ms — just 36% slower, while requiring around 1/30 of actual cache storage. In contrast, the 1 GB cache using `FIFO` is 79% slower, and `LRU` is 83% slower than cache with infinite capacity.

However, `LAC` does incur an initial graph-building overhead (which is incorporated in all above results). Recall from Section 3 that the time complexity of the graph-building algorithm is quadratic on batch size. Table 3 lists the graph-building overhead incurred for various sized batches. Graphs for batch sizes of 100K and over are prohibitively slow to build (over 30 minutes to prepare a batch of one million queries). This result signifies that `LAC` should only be used when batch sizes are not large, and that `FIFO` and `LRU`, which incur no pre-processing overhead, would still be appropriate for very large batches.

Batch Size	Graph Building Time
1000	50 ms
10,000	162 ms
100,000	10,208 ms
1,000,000	1,893,998 ms

Table 3: Overhead of Shortest-First

5 RELATED WORK

Ideas in this paper trace back to seminal work in multi-query optimization [32, 33]. This area of research is concerned with extracting performance via the observation that query processors may find opportunities to reuse intermediate results in other queries that are concurrently running. As detailed in [34], there are generally two classes of approaches toward multi-query optimization: online and offline. Online optimization, as done in QPipe [18], DataPath [2], and others [31], are executed on a per-query basis. Work in this area leverages both the commutativity of commonly-used relational operators, and the fact that the order in which data blocks are fetched is generally inconsequential in the relational data model. As such, the sequencing of relational operators can be dynamically reordered upon execution to optimize query plans. However, the potential for reuse that an online optimizer can extract is somewhat limited because each query is considered in isolation [13].

In contrast, *offline* multi-query optimizers are used to analyze batches of queries prior to dispatching them for execution [13, 26]. Despite the higher overhead cost for analysis compared to online models, a more in-depth analysis of a batch of queries can be performed — uncovering how their operators and result sets relate to each other. A finalized schedule of queries is produced to minimize batch execution time. Our proposed techniques also subscribe to the offline approach, and therefore, we shall focus this section on offline query optimizers.

Manegold, *et al.* describe an offline query optimizer for the columnar database, MonetDB [26]. They build a dependency graph to inform the query processor on which cached results can be reused rather than running redundant query sub-expressions. The work differs from ours in that it did not consider cache replacement, or attempt to execute queries out-of-order to exploit reusing partial results. Luo, *et al.*, described a workload-reordering scheme to increase parallel access to buffered data blocks for scan-based queries [24]. SharedDB [13] groups together similar query types, and creates a *global query plan*, which is a directed acyclic graph containing all queries and their shared operators. Expensive operators (such as joins, sorting, and group-by) are processed and results are kept *in situ*. Any batched queries requiring a subset of those results can have those results routed as part of its query plan.

The area of research within multi-query optimization from which we draw the most inspiration is *semantic caching* [9, 21]. This approach groups together query results (records/tuples) into *semantic regions*, which are constraint expressions that describe the records that fall into the region. A query first probes the available semantic regions to find subsets of results that can be unioned and reused, followed by a "remainder" query that requests the processor to retrieve the difference. Cache replacement is done on the unit of semantic regions, using temporal or distance metrics.

The current work builds on our previously implemented semantic cache of bitmap indices [28]. The query intervals assigned to each cached result are tantamount to the semantic regions described above. Work in semantic caching are varied, mainly finding applications in client-server paradigms to reduce data transfers [5–7, 11, 14, 15, 20, 22]. Our work differs in that we employ query reordering in addition to a novel cache-replacement policy to minimize cache misses.

Finally, our cache shares some similarities with *materialized views* [3, 16]. A materialized view is a precomputed query result (table) that is stored back into the database. Views commonly store the derived results of an expensive operation, such as a join or subquery, so that future queries might avoid recomputing the costly expression. Our system is fundamentally different from materialized views in that it does not systematically generate precomputed results in hopes of being useful at a later time. Also unlike materialized views, our system cache is not persisted in the database, so we are not as concerned with updating the view when the underlying tables change.

6 FUTURE WORK AND CONCLUSION

This paper investigates workload scheduling and cache-replacement schemes in a data-storage framework. Our system accepts concurrent queries to an underlying data store, and seeks to execute batches of queries with high throughput. Queries are scheduled in a shortest-first manner, which is optimal when cache capacity is unconstrained. Under most environments, in which cache capacity is limited, we found that classical replacement schemes do little to exploit the existing inter-query dependencies in a workload batch. To this end, we further presented a graph model that captures query dependencies and their (potential) contributions toward accelerating the overall execution of the workload. Our replacement policy, which is informed by this dependency graph, achieved significant speedups over classical replacement algorithms over the TPC-H benchmark.

In the next stages of our research, we plan to explore more sophisticated scheduling algorithms that work better with the LAC cache. For instance, by grouping queries that reuse the same result vectors, those results could be fully utilized more quickly, allowing that cache space to be freed up with less penalty. Additionally, we intend to explore the usage of search algorithms for query scheduling, with a focus on approximating, interruptible algorithms that can provide flexibility to fit performance constraints.

ACKNOWLEDGMENTS

This undergraduate research was funded in part by a McCormick Research Grant to Julia Kaeppel from the University of Puget Sound. We would also like to thank Dr. Patrick Jarvis for his valuable insight.

REFERENCES

[1] 2021. TPC-H Decision Support Benchmark. http://www.tpc.org/tpch
[2] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 519–530.
[3] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. *ACM SIGMOD Record* 15, 2 (1986), 61–71.
[4] Sam Burdick, Jahrme Risner, David Chiu, and Jason Sawin. 2018. Fault-Tolerant Query Execution over Distributed Bitmap Indices. In *International Conference on Big Data Computing Applications and Technologies*. 21–30.
[5] Chungmin Melvin Chen and Nicholas Roussopoulos. 1994. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *EDBT'94*. 323–336.
[6] Li Chen, Elke A. Rundensteiner, and Song Wang. 2002. XCache: A Semantic Caching System for XML Queries. In *ACM SIGMOD International Conference on Management of Data*. 618–618.
[7] Boris Chidlovskii and Uwe M. Borghoff. 2000. Semantic Caching of Web Queries. *The VLDB Journal* 9, 1 (March 2000), 2–17.
[8] Fabian Corrales, David Chiu, and Jason Sawin. 2011. Variable Length Compression for Bitmap Indices. In *Database and Expert Systems Applications*. 381–395.
[9] Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *International Conference on Very Large Data Bases (VLDB '96)*. 330–341.
[10] François Deliège and Torben Bach Pedersen. 2010. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *International Conference on Extending Database Technology (EDBT '10)*. 228–239.
[11] Laurent d'Orazio, Fabrice Jouanot, Yves Denneulin, Cyril Labbé, Claudia Roncancio, and Olivier Valentin. 2007. Distributed Semantic Caching in Grid Middleware. In *Database and Expert Systems Applications*. 162–171.
[12] Katerina Fotiadou and Evaggelia Pitoura. 2008. BITPEER: continuous subspace skyline computation with distributed bitmap indexes. In *International Workshop on Data Management in Peer-to-Peer Systems*. 35–42.
[13] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. *Proceedings of the VLDB Endowment* 5, 6 (2012), 526–537.
[14] Parke Godfrey and Jarek Gryz. 1997. Semantic Query Caching for Hetereogeneous Databases. In *Intelligent Access to Heterogeneous Information*. 6.1–6.6.
[15] Parke Godfrey and Jarek Gryz. 1998. Answering Queries by Semantic Caches. In *Database and Expert Systems Applications*. 485–498.
[16] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995).
[17] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *International Conference on Data Engineering (ICDE'14)*. 484–495.
[18] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 383–394.
[19] hive [n.d.]. Apache Hive Project, http://hive.apache.org.
[20] Vagelis Hristidis and Michalis Petropoulos. 2002. Semantic Caching of XML Databases. In *5th International Workshop on the Web and Databases*. 25–30.
[21] Arthur M. Keller and Julie Basu. 1996. A Predicate-based Caching Scheme for Client-server Database Architectures. *The VLDB Journal* 5, 1 (Jan. 1996), 035–047.
[22] Dongwon Lee and Wesley W. Chu. 1999. Semantic Caching via Query Matching for Web Sources. In *Proceedings of the Eighth International Conference on Information and Knowledge Management (CIKM '99)*. 77–85.
[23] Daniel Lemire, Ownen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data and Knowledge Engineering* 69 (2010). Issue 1.
[24] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael Watzke. 2008. Transaction reordering with application to synchronized scans. In *International Workshop on Data Warehousing and OLAP,*. 17–24.
[25] Kamesh Madduri and Kesheng Wu. 2009. Efficient Joins with Compressed Bitmap Indexes. In *ACM Conference on Information and Knowledge Management*. 1017–1026.
[26] Stefan Manegold, Arjan Pellenkoft, and Martin L. Kersten. 2000. A Multi-query Optimizer for Monet. In *British National Conference on Databases (Lecture Notes in Computer Science)*. 36–50.
[27] Ben McCamish, Rich Meier, Jordan Landford, Robert B. Bass, David Chiu, and Eduardo Cotilla-Sanchez. 2016. A backend framework for the efficient management of power system measurements. *Electric Power Systems Research* 140 (2016).
[28] Sarah McClain, Manya Mutschler-Aldine, Colin Monaghan, David Chiu, Jason Sawin, and Patrick Jarvis. 2021. Caching Support for Range Query Processing on Bitmap Indices. In *International Conference on Scientific and Statistical Database Management (SSDBM'21)*. 49–60.
[29] Mitchell Nelson, Zachary Sorenson, Joseph M. Myre, Jason Sawin, and David Chiu. 2019. GPU Acceleration of Range Queries over Large Data Sets. In *International Conference on Big Data Computing, Applications and Technologies*. 11–20.
[30] Patrick E. O'Neil. 1989. Model 204 Architecture and Performance. In *International Workshop on High Performance Transaction Systems*. 40–59.
[31] Sunita Sarawagi and Michael Stonebraker. 1996. Reordering Query Execution in Tertiary Memory Databases. In *International Conference on Very Large Data Bases*. 156–167.
[32] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (1988), 23–52.
[33] Kyuseok Shim, Timos K. Sellis, and Dana S. Nau. 1994. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data Knowl. Eng.* 12, 2 (1994), 197–222.
[34] Panagiotis Sioulas and Anastasia Ailamaki. 2021. Scalable Multi-Query Execution using Reinforcement Learning. In *International Conference on Management of Data*. 1651–1663.
[35] Kurt Stockinger and Kesheng Wu. 2006. Bitmap Indices for Data Warehouses. In *In Data Warehouses and OLAP*. Press.
[36] Y. Su, G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, and J. Ahrens. 2013. Taming massive distributed datasets: data sampling using bitmap indices. In *Symposium on High-Performance Parallel and Distributed Computing*. 13–24.
[37] Yu Su, Yi Wang, and Gagan Agrawal. 2015. In-Situ Bitmaps Generation and Efficient Data Analysis based on Bitmaps. In *International Symposium on High-Performance Parallel and Distributed Computing*. 61–72.
[38] Miguel Velez, Jason Sawin, Alexia Ingerson, and David Chiu. 2016. Improving Bitmap Execution Performance Using Column-Based Metadata. In *International Conference on Future Internet of Things and Cloud*. 371–378.
[39] Harry K. T. Wong, Hsiu fen Liu, Frank Olken, Doron Rotem, and Linda Wong. 1985. Bit Transposed Files. In *Proceedings of VLDB*. 448–457.
[40] Kesheng Wu, Ekow Otoo, and Arie Shoshani. 2004. On the Performance of Bitmap Indices for High Cardinality Attributes. In *VLDB*. 24–35.
[41] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. 2002. Compressing bitmap indexes for faster search operations. In *International Conference on Scientific and Statistical Database Management*. IEEE, 99–108.