Caching Support for Range Query Processing on Bitmap Indices

Sarah McClain Manya Mutschler-Aldine
Colin Monaghan David Chiu
Computer Science
University of Puget Sound
Tacoma, WA, USA

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

ABSTRACT

Bitmaps are commonly used for indexing read-mostly data sets. The range of an attribute is split into bins, where its values are placed: $b_{ij} = 1$ denotes the value of the ith tuple is in the jth bin, and $b_{ij} = 0$ otherwise. A number of query types can be decomposed into the systematic application of boolean operators over sets of bins. However, when bitmaps are high-dimensional, the overall query-processing performance can deteriorate due to the increased number of bins that participate per query.

We propose a caching framework that organizes, manages, and integrates cached partial results to accelerate query processing on high-dimensional bitmaps. We begin by showing that, to resolve general complex disjunctive and conjunctive queries, the selection of an optimal set of partial bitmap results is NP-complete. A restriction on this problem to only consider consecutive bin sequences (characteristic of common range and point queries) allows us to solve it efficiently. The evaluation our caching system over several workloads carried out on the TPC-H benchmark and a real network-intrusion data set is presented.

KEYWORDS

Bitmap index, caching, performance

1 INTRODUCTION

Praised for versatility and space-time efficiency, *bitmaps* [38] are a common indexing scheme that have found their way into modern data-management applications, including scientific analysis [23, 26, 35, 36], databases and data warehouses [19, 29, 34], information retrieval [14, 27], data streams [15, 30], among others. A bitmap index, essentially a sparse 2D array containing a binary representation of the underlying database. To generate a bitmap, an attribute is partitioned into multiple bins, with each bin representing a range of values (or an atomic value) in the attribute domain. Either a 0 or 1 is assigned to the corresponding bin depending on whether a row's attribute falls in the bin's range.

An example is shown in Table 1. The upper table presents a portion of a relation, *Solar Masses*. The relation consists of two attributes *Star*, which records a name, and $M_\odot$ corresponds to the star's solar mass. The lower table presents a possible bitmap index for *Solar Masses*. As the names recorded in *Star* are discrete, each value is simply represented by a single bin (or *bit-vector*). For example, the s_0 bit-vector encodes α *Orionis*, s_1 encodes *R136a1*, and so on. The m bins in the bitmap represent ranges of possible solar-mass values. Thus, m_0 encodes solar-mass values from $[0, 10)$, m_1 encodes $[10, 40)$, m_2 encodes $[40, 80)$, m_3 denotes values in $[80, 150)$ and m_4 represents $[150, \infty)$. During the so-called binning process, each tuple from *Solar Masses* is converted a single row in the bitmap. Consider tuple t_2, as its *Star* value is *R136a1* a 1 is placed in the s_1. The value of $t_2[M_\odot]$ is 315, so a 1 is placed in the m_4 bit-vector and the remaining m bit-vector are assigned a 0 in that row.

Solar Masses

Tuple_ID	**Star**	$M_\odot$
t_1	α Orionis	11
t_2	R136a1	315
t_3	ρ Cassiopeia	40
t_4	Proxima Centauri	.123

Bitmap of Solar Masses

	Star				$M_\odot$				
ID	s_0	s_1	s_2	s_3	m_0	m_1	m_2	m_3	m_4
t_1	1	0	0	0	0	1	0	0	0
t_2	0	1	0	0	0	0	0	0	1
t_3	0	0	1	0	0	0	1	0	0
t_4	0	0	0	1	1	0	0	0	0

Table 1: Example Relation and Corresponding Bitmap

Upon receiving a query, the bitmap-index processor first identifies the appropriate set of bit-vector participating in that query. After applying the appropriate set of boolean operators across those bit-vectors, a *result-vector* is obtained. For instance, to answer a range query, $\sigma_{M_\odot \geq 40}(SolarMasses)$, we retrieve vectors $\{m_2, m_3, m_4\}$ of the index and apply a boolean OR across them to obtain the result-vector. A value of 1 in the ith position in the vector indicates that the ith tuple should be fetched from disk. Bitmap indices can be a very selective and efficient pruning mechanism, given that the attributes have been sufficiently binned, and higher cardinality (*i.e.*, larger number of bit-vectors) assigned per attribute generally increases selectivity.

While higher cardinality can reduce the response time of *point* queries (in which a few isolated bit-vectors are selected and operated over), the cost of executing open-ended *range* queries may actually increase. To resolve a range query, it may require the sequential processing of a large number of bit-vectors before a result-vector

SSDBM 2021, July 6–7, 2021, Tampa, FL, USA

ACM ISBN 978-1-4503-8413-1/21/07…$15.00
https://doi.org/10.1145/3468791.3468800

is obtained. This represents a significant overhead despite several efforts toward improving range-query performance [28, 33, 39].

An effective method for reducing average query response time is to *intentionally* cache query results for reuse in future query resolution. Each result is stored alongside semantic metadata (such as the query constraints) that describe its content. Subsequent queries invoke the processor to examine the available metadata to determine which cached results can be reused. Only the remainder portion of the query must be dispatched for retrieval. This approach is known as *semantic caching*, and it is an inspiration for this work.

The simple structure of bitmaps and their consistent processing pattern make them particularly amenable to caching. Metadata for each cached result-vector can be encoded simply with the *start* and *end* bit-vector identifiers. Because result-vectors are themselves consistent in format with the index's bit-vectors, any result-vector can be seamlessly included as part of another query using existing bitmap operators. Additionally, bit-vectors can be maintained and processed in a compressed format, thus reducing memory overhead.

This paper explores the integration of bitmap indices and a caching framework in an existing bitmap-index processor that we developed previously [18]. We focus on the nontrivial problem of result-vector identification in the cache and present an optimal algorithm for their retrieval. This paper makes the following contributions:

- We integrate a semantic-caching inspired technique into an existing bitmap-index processor. To the best of our knowledge, this is a novel technique for accelerating bitmap supported query processing.
- We show that selecting the optimal number of result-vectors to aid in the processing of an arbitrary query is NP-complete. However, with a reasonable restriction on the type of cached results, a provably-correct greedy algorithm can derive an optimal solution in $O(n \log n)$, where n is the number of cached results.
- We present a rigorous experimental study of our caching framework applied to the well-known TPC-H and KDD-Cup'99 benchmarks. In this study, our framework realized workload depended speedups of approximately 2× to over 140× for moderate cache sizes.

2 BACKGROUND

In this section we first give background on bitmap indexing compression and query processing, followed by an overview of the semantic-caching paradigm.

Bitmap-Index Processing: As illustrated in Table 1, bitmap indexes are generally sparse. There has been a myriad of compression schemes designed to exploit the sparseness of bitmap indices. Many of these are variations of hybrid run-length compression, and one of the classical approaches is called the *Word-Aligned Hybrid* codes (WAH) [40]. We restrict our background description to WAH due to the fact that most recent compression schemes were heavily inspired by WAH.

WAH compresses each bit-vector (column) of the bitmap index independently. It first partitions a bit-vector into consecutive 63-bit sequences. Heterogeneous sequences (those that are not all 0's or all 1's) are encoded in *literal* atoms. *Literal* atoms are 64-bit words and have the form: $(flag, literal)$. The $flag$ is the most-significant-bit (MSB) of atom and it is set to 0, indicating the atom is a *literal*. The remaining 63 bits, $literal$, record the heterogeneous sequence.

WAH compresses consecutive homogeneous sequences in *fill* atoms. These atoms have the form $(flag, value, run\_length)$. The $flag$ (the MSB) is set to 1 indicating the atom is a *fill*. The $value$ bit (second-MSB) encodes the homogeneous value of the sequences being compressed (1 or 0). The remaining 62 bits are dedicated to run_length, which records the number of homogeneous sequences being compressed.

WAH-compressed bit-vectors do not need to be decompressed to query. When processing a point query of the form, $X \circ Y = Z$ where X and Y are compressed bit-vectors, $\circ$ is a bitwise logical operation and Z is a result bit-vector, the WAH query processing algorithm treats X and Y as stacks of atoms. The top atoms of X and Y are popped off the stack and processed in an iterative fashion. Each iteration of processing results in a new atom being added to Z or an existing atom being updated. After an operand atom is completely processed, which could take multiple iterations for *fill* atoms, the next atom from the bit-vector is popped.

The WAH point query algorithm can easily be extended to process *range queries*. A WAH range queries take the form $Z = X_1 \vee X_2 \vee \ldots X_n$ where X_i is a WAH compressed bitmap bin that encodes attributes within the desired range and Z is the result vector that will indicate the tuples that should be retrieved from disk. A simple iterative algorithm is often employed to solve range queries. First Z is initialized to X_1, then $Z \leftarrow Z \vee X_j$ is repeated for all j such that $2 \leq j \leq n$.

Due to the overhead of stack processing, the more compressed the bit-vector operands are, the more efficient the query becomes. Thus, a chief contributor to a bitmap's efficiency in terms of query-processing performance is the number of bins assigned to a given search attribute. For example, suppose that we wish to index an `Income` attribute, and just four bins are used to correspond with the following ranges $[0, 25K)$, $[25K, 50K)$, $[50K, 80K)$, and $[80K, \infty)$. Assume that the distribution of salaries is heavily skewed toward the $[50K, 80K)$ bin, resulting in a high volume of 1s in this column. Any query within this bin does not stand to benefit due to low selectivity. The natural solution to this problem is to increase bin-cardinality, splitting high-volume regions into more bins with finer granularity. The obvious tradeoff of high-cardinality binning is the increased size of the bitmap. However, higher cardinality also impacts the efficiency of range-query processing, in which the set of bins in the selected range must be processed sequentially.

Semantic Caching: Independent of bitmaps, we give a brief exposition of *semantic caching*, the technique from which we drew inspiration for this work. On the surface, semantic caching [10, 22] is a simple query optimization scheme in which query results are cached, along with sufficient metadata for their identification, in higher levels of memory. A subsequent query initially searches the cache for results that might satisfy a subset of the desiderata. This query to the cache is called the *probe query*. Based on what can be retrieved from the cache, the original query may be realigned to retrieve a smaller subset of data on disk (*remainder query*). Using a simple example to demonstrate, suppose we have an *Employees*

table on which the following workload is executed:

$$Q_1 \leftarrow \sigma_{salary \geq 80000}(Employees)$$
$$Q_2 \leftarrow \sigma_{salary \geq 40000}(Employees)$$

Note that Q_1 contains a subset of results requested for in Q_2, and therefore, Q_2 can be decomposed into a union of two terms:

$$Q_2 \leftarrow Q_1 \cup \sigma_{salary \geq 40000 \wedge salary < 80000}(Employees)$$

where the latter σ-term is the remainder query of Q_2, which will be dispatched to the query processor for execution. Assuming the results for Q_1 can be identified quickly in a cache, the execution of Q_2 may be greatly accelerated.

In this paper, we study the performance benefits afforded through the unification of bitmap indices and a scheme based loosely on semantic caching. In the section that follows, we formalize the result-vector selection problem and present algorithms that solve or approximate the solution.

3 CACHE-ENABLED BITMAP PROCESSOR

In this section, we first describe the overall architecture of our caching framework, then we formalize the *result-vector selection problem*, and later, its restricted counterpart, which we will solve.

3.1 System Overview

For this study, bitmap processor considers *ors-of-ors* and *ands-of-ors* queries.[1] Specifically, let $B = \{1, ..., m\}$ denote a set of bit-vector identifiers, and a bit-vector b_i ($i \in B$) is stored on disk. Our query processor can solve queries of the form:

$$Q_1 \circ Q_2 \circ ... \tag{1}$$

where each Q_k is a *query segment*, defined as a set of bit-vectors identified in B and is processed iteratively using disjunction: $\bigvee b$ for all $b \in Q_k$, and $\circ$ is a conjunction ($\wedge$) or disjunction ($\vee$) operator. Consider the following query: $\{2, 3\} \wedge \{5, 8, 9, 10\}$. The processor decomposes this query into query segments $\{2, 3\}$ and $\{5, 8, 9, 10\}$. Corresponding bit-vectors are retrieved for each query segment, and processed using disjunction. For example, $(b_2 \vee b_3)$ and $(b_5 \vee b_8 \vee b_9 \vee b_{10})$ will be processed independently, and their results are combined with $\wedge$.

Our system reduces the overhead of query processing by maintaining a cache of query segment results. Figure 1 illustrates the high-level view of our cache-enabled bitmap processor. Given a query, the system dispatches each segment as a unit of execution. On the left, the cache stores a set of *result-vectors* from previous query segments. Each result-vector associates with the query segment (shown as ovals) from which it was derived. Note that some query segments may overlap. On the right, the full bitmap index is initially stored on disk: one file per bit-vector.

On receiving a probe query, the cache runs a *result-vector selection* routine to identify a set of result-vectors R'. The bit-vectors that the probe query could not cover with the fetched result-vectors comprise the remainder query. The *remainder vectors* (*i.e.*, the set of bit-vectors that satisfy the remainder query) are fetched and processed using disjunction. Finally, the result-vectors and the remainder vectors are merged into a single result-vector representing

[1]This supports the class of both point queries and range queries.

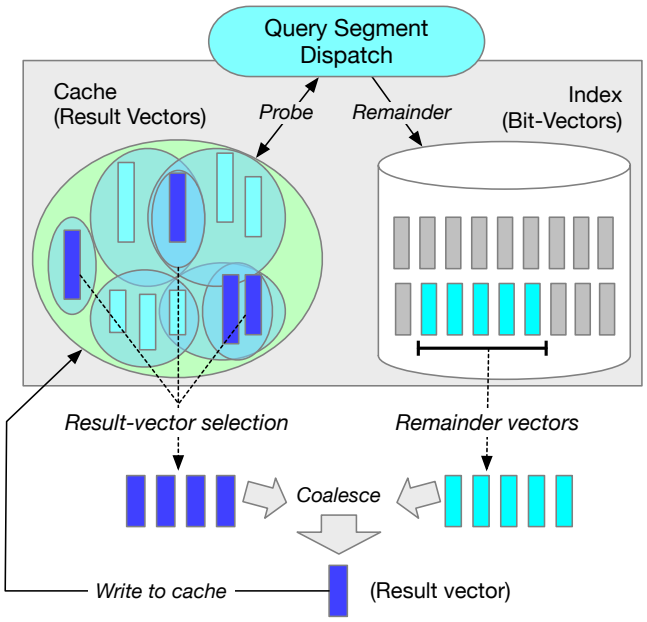


Figure 1: Overview of the Bitmap Processor

the result for that query segment, and this result-vector is cached. This process is repeated for all query segments, and their results are combined using $\circ$. Finally, the combined result is used to fetch candidate tuples on disk and is then cached.

While the overall cache framework is rather intuitive, the challenge lies in the result-vector selection routine. We are interested in finding the fewest result-vectors that cover the largest number of bit-vectors requested by a query segment. The minimal set of result-vectors is desirable because they will be iteratively coalesced in the formation of the query-segment solution.

3.2 Problem Statement

The successful execution of a query segment produces a result-vector that can be cached. Let $R = \{1, ..., n\}$ be a set of identifiers for all result-vectors cached by prior queries. For some $j \in R$, $Q_j^r \subseteq B$ denotes the set of bit-vectors used to derive result-vector j. Given a subset $R' \subseteq R$ of result-vectors, we define the *coverage* of R' with respect to query segment Q as follows,

$$cov(Q, R') = \begin{cases} -\infty, & \text{if } |(\cup_{j \in R'} Q_j^r) \setminus Q| > 0 \\ |Q \cap (\cup_{j \in R'} Q_j^r)|, & \text{otherwise} \end{cases} \tag{2}$$

Specifically, $cov(Q, R')$ is the number of Q's bit-vectors that the result-vector j ($\forall j \in R'$) represents. Note that R' provides $-\infty$ coverage of Q if any of its result-vectors were derived by bit-vectors exceeding the boundaries of Q. For instance, consider a cache that contains only two result-vectors x and w where $Q_w^r = \{11, \ldots, 50\}$ and $Q_x^r = \{1, \ldots, 40\}$. If the query segment $Q = \{5, \ldots, 100\}$ is currently being requested, then $cov(Q, R') = 50$ if $R' = \{w\}$ is returned. It follows that $cov(Q, R') = -\infty$, when $R' = \{x\}$ or $R' = \{w, x\}$ is returned.

The *result-vector selection problem* is defined as follows. Given a query segment Q and a set of result-vector identifiers R, we wish

to find an optimal subset of result-vector identifiers $R^* \subseteq R$ such that $cov(Q, R^*)$ is maximized, subject to:

$$|R^*| = min\ \{|S|\}, \forall\ S \subseteq R \tag{3}$$

In other words, we wish to determine a subset of result-vector identifiers $R^* \subseteq R$ with minimal size that produces the maximal coverage of Q.

	Definition
R	Set of all cached result-vector identifiers
R'	Set of cached result-vector identifiers ($R' \subseteq R$) returned by the result-vector selection routine
R^*	Optimal set of cached result-vectors identifiers ($R^* \subseteq R$) for a given query segment.
B	Set of all bit-vector identifiers
Q	Query segment ($Q \subseteq B$)
Q_j^r	Query segment associated with the jth result-vector
$Q_1 \circ Q_2 \circ \ldots$	A query, *i.e.*, a logical operations applied to a sequence of query segments.
$cov(Q, R')$	Number of bit-vectors requested by query segment covered by R'

Table 2: Notation Reference

3.3 Proof of NP-Completeness

In this subsection, we show that the *result-vector selection problem* is NP-Complete. In order to demonstrate its hardness, we must first re-cast it to a decision-based variant. In this variant, we are given a query segment Q and a set of result-vector identifiers, an integer $k \geq 1$, and an integer $v \geq 0$. We wish to decide whether there exists a subset $R^* \subseteq R$ whose coverage $cov(Q, R^*) = v$ and $|R^*| \leq k$. Formally, we define the language *RES_VEC* as follows,

$$\begin{aligned} RES\_VEC = \{&\langle Q, R, k, v\rangle : \\ &Q \text{ is a set of integer bit-vector identifiers,} \\ &R \text{ is a set of integer result-vector identifiers,} \\ &k \geq 1 \text{ and } v \geq 0 \text{ are integers,} \\ &\exists\ R^* \subseteq R : cov(Q, R^*) = v \text{ and } |R^*| \leq k\} \end{aligned} \tag{4}$$

Note that our original optimization problem is a special case of *REC_VEC*. Recalling that $|Q| \leq m$ and that $|R| = n$, one would only need to test all pairs $\{(k, v) : 1 \leq k \leq n \wedge 1 \leq v \leq m\}$ to determine whether the optimal solution R^* exists. As there are at most mn pairs to test, the problem transformation only induces an additional polynomial factor of running time, so it does not change *RES_VEC*'s complexity class.

Theorem 3.1. *RES_VEC* $\in$ *NP*.

Proof. To show that *RES_VEC* $\in$ *NP*, we present the following polynomial-time verifier. In addition to the problem inputs $\langle Q, R, k, v\rangle$, we further accept a certificate $C \subseteq R$ representing a subset of result-vector identifiers. The verification algorithm must ensure that the following properties hold: $C \subseteq R$, $C \leq |k|$, and $cov(Q, C) = v \neq -\infty$. Because the complexity of each check is at-worst linear, we conclude that the verification algorithm runs in polynomial time, and therefore *RES_VEC* $\in$ *NP*. □

Next we show that the *RES_VEC* problem is NP-hard via reduction from the *set cover* problem.

Theorem 3.2. *RES_VEC is NP-hard.*

Proof. We consider the well-known NP-complete problem *SET_COVER* [21], stated as follows. Given a set of positive integers U and a set of sets $S = \{s \mid s \subseteq U\}$, *SET_COVER* decides whether there exists a subset $S' \subseteq S$ of cardinality c, such that the union of all sets in $S' = U$.

The *SET_COVER* $\leq_p$ *RES_VEC* reduction is straightforward. We assign the query segment Q to be the set of positive integers U. The cardinality constraint k is assigned with c, and the minimal coverage constraint v is assigned with $|U|$ to ensure that the union of all identified subsets must contain $|U|$ elements. The set of result-vector identifiers R is prepared by associating each subset $s \in S$ with an identifier $j > 0$, that is, $R \leftarrow \{j \mid 1 \leq j \leq |S|\}$. Finally, the queries Q_j^r are assigned the corresponding subset s_j. This reduction can be performed in polynomial time, and therefore *RES_VEC* is NP-hard. □

Theorem 3.3. *RES_VEC is NP-complete.*

Proof. Because Theorems 3.1 and 3.2 hold on *RES_VEC*, we can conclude that *RES_VEC* is NP-complete. □

Given the hardness of the *result-vector selection problem*, we decided to focus on solving a restricted version, described next.

3.4 Problem Restriction and Solution

The *result-vector selection problem* is posed in a way as to provide support for compound database queries satisfying multiple conditionals. And though the problem is computationally prohibitive to solve, we observe that a compound query can often be decomposed into the successive resolution of simpler queries, in which the query segments are of the form, $b_c \vee b_{c+1} \vee \ldots \vee b_{c+d}$ where $b_i \in B$ and c and d are integers such that $1 < c < d$. In other words, query segments must be a sequence of *consecutive* bit-vectors. Results of a compound query can then later be reconstructed through a combination of these simplified query results. We define this restricted version of the *result-vector selection problem* as the *range-restricted result-vector selection problem*. Given a query segment Q and a set of result-vector identifiers R, we wish to find an optimal subset of result-vector identifiers $R^* \subseteq R$ such that $cov(Q, R^*)$ is maximized, subject to: $|R^*| = min\ \{|S|\}, \forall\ S \subseteq R$. In other words, we wish to determine a subset of result-vector identifiers $R^* \subseteq R$ with minimal size that produces the maximal coverage of Q.

With the slight restriction that the cache result-vectors must be derived from consecutive bit-vectors, we can define a polynomial-time solution for the selection problem. To aid in our discussion we are adding the following notation: let j be a result-vector identifier, under the above restriction that Q_j^r must be consecutive sequence of bit-vector identifiers, we denote $j.startId$ as the lowest valued identifier in this range and $j.endId$ as the highest value.

In Figure 2 we demonstrate MaximizeCoverage using an example. Suppose the result-vectors $A \ldots G$ are stored in the cache. The rectangle at the bottom of the figure represents the queried range.

Algorithm 1 MaximizeCoverage

```
1:  Input
2:      Q    the query segment, Q ⊆ B
3:      R    the set of result-vector identifiers
4:  Output
5:      R*   an optimal set of result-vector identifiers
6:  ▷ reduce search space (remove result-vectors with no coverage)
7:  for all j ∈ R do
8:      if cov(Q, {j}) = −∞ then
9:          R ← R \ {j}
10: L ← toList(R)
11: L ← sortListByStartId(L)
12: ▷ remove subranges
13: i ← 0
14: while i < length(L) − 1 do
15:     if L[i].startId = L[i + 1].startId then
16:         if L[i].endId ≥ L[i + 1].endId then
17:             L.remove(i+1)
18:         else
19:             L.remove(i)
20:     else if L[i].endId ≥ L[i + 1.]endId then
21:         L.remove(i+1)
22:     else
23:         i ← i + 1
24: ▷ remove vectors whose ranges do not change coverage
25: R* ← R
26: for i ← 1 to length(L) do
27:     if cov(Q, R* \ {L[i]}) = cov(Q, R*) then
28:         R* ← R* \ {L[i]}
29: return R*
```

MaximizeCoverage will first remove subrange G because it is subsumed by E. It scans left to right and fetches the result-vectors associated with the ranges (highlighted in blue). After fetching A and C, range B is initially retained. However, when D is encountered, its $endId$ extends beyond B, allowing us to drop it from consideration. After the result-vector D is selected, the algorithm then inspects every point $p \in [D.startId + 1, D.endId]$ for any result-vectors that have a $startId = p$ and an $endId > D.endId$. Two result-vectors, E and F, are identified as possibilities, and the algorithm selects E since its end-point extends farther. In this particular example, the fetched ranges A, C, D, E just happen to fully cover the queried range, though that may not always be true. Any missing bit-vectors not covered by partial solutions will have to be retrieved from disk.

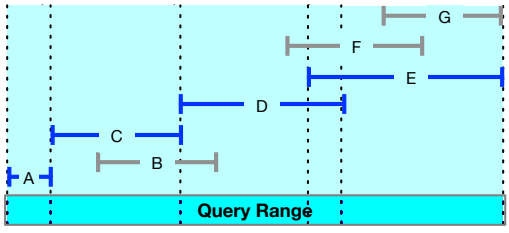


Figure 2: MaximizeCoverage Example

3.5 Optimality of MaximizeCoverage

The MaximizeCoverage (Algorithm 1) produces an optimal solution to *range-restricted result-vector selection problem*. The inputs are Q, the query segment, and R, the set of cached result-vector identifiers. The algorithm returns the optimal subset $R^* \subseteq R$, where $cov(Q, R^*)$ is the maximum possible coverage of Q given R and $|R^*|$ is the minimum number of result-vectors that can be used to achieve this coverage. First, the algorithm removes all result-vectors from R that extend past the bounds of Q (lines 7 to 9). The remaining result-vector identifiers are put into a list and ordered by the identifier of their starting column (lines 10 and 11). All subranges are removed at line 13 to 23. A result-vector x is a subrange of a result-vector y if $x.startId \geq y.startId \bigwedge x.endId \leq y.endId$.

Lemma 3.1. *After R is restricted to result-vectors in bounds of Q, subranges are not required to build an optimal result-vector set for the range-restricted result-vector selection problem.*

Proof. We show subranges are not needed to construct an optimal solution by contradiction. Assume that subranges are needed to construct an optimal solution. Further assume, Q is a query segment, R is the set of cached rages, and R^* is an optimal solution such that $c = cov(Q, R^*)$ and $n = |R^*|$. Let s be any subrange $\in R^*$, such that $s, t \in R$ and s is a subrange of t. Notice that both s and t cannot be in R^* since then s could be removed without loss of coverage, meaning that if they were both included, n would not be minimal. However, since $cov(Q, \{s, t\}) = cov(Q, \{t\})$, $cov(Q, ((R^* \setminus \{s\}) \cup \{t\})) = c$ and $|(R^* \setminus \{s\}) \bigcup \{t\}| = n$. This substitution can be performed for all subranges in R^*, which contradicts the assumption that subranges are needed to build an optimal set. □

At line 25, MaximizeCoverage initializes the return set to contain all remaining result-vectors. Notice that at this point, $cov(Q, R^*)$ is guaranteed to be the maximal coverage that can be achieved as R^* contains all result-vectors that need to be considered. The algorithm then iterates through all of the candidate result-vectors in the order of their $startId$ (lines 26 to 28). In each iteration, it tests to see if removing the candidate result-vector reduces the coverage of R^*. If coverage does change, the candidate vector is kept, otherwise it is removed. In this manner all result-vectors that uniquely cover a portion of Q are kept. If a portion of Q is covered by multiple result-vectors, Maximize_Coverage takes the covering vectors with the largest $startId$ (which also must have the largest $endId$ since all subranges have been removed.)

Theorem 3.4. *Maximize_Coverage produces an optimal solution to the range-restricted result-vector selection problem.*

Proof. Let $R^* = \{r_1, r_2, \ldots, r_p\}$ be a solution generated by Maximize_Coverage ordered such that:

$$r_i.startId < r_{i+1}.startId, \quad \forall i : 1 \leq i \leq p - 1$$

Let $O = \{o_1, o_2, \ldots, o_m\}$ be an optimal solution also ordered by $startId$. The goal is to show that for $R_i^* \subset R^*$ such that $R_i^* = \{r_1, r_2, \ldots, r_i\}$ and $O_i \subset O$ such that $O_i = \{o_1, o_2, \ldots, o_i\}$ that $\forall i \leq p : cov(Q, R_i^*) \geq cov(Q, O_i)$.

This can be shown through induction. Let $i = 1$ for the base case. Since all the subranges have been removed, all of the $startIds$ must be unique. Thus r_1 is the result-vector with the earliest $startId$ in

R and it must be included in the optimal set as it provides unique coverage. Therefore, $cov(Q, R^*_1) \geq cov(Q, O_1)$.

For $k > 1$, assume the statement is true for $k-1$ and we will prove it for k. There are two cases to consider for r_k: the first case is when r_k uniquely covers a portion of Q, meaning it is the *only* result-vector in R that covers some part of Q. In this case r_k must be part of the optimal solution, and by the induction hypothesis $cov(Q, R^*_{k-1}) \geq cov(Q, O_{k-1})$, thus, $cov(Q, R^*_{k-1} \bigcup r_k) \geq cov(Q, O_{k-1} \bigcup r_k)$. The other case to consider is when r_k is providing coverage for a portion of Q that is covered by multiple result-vectors in R. In these cases, MaximizeCoverage takes the range with the largest $startId$ that covers the portion. Therefore, $cov(Q, O_k)$ could only be greater than $cov(Q, R^*_k)$ if o_k had a larger $endId$, however that would imply $o_k.startId \leq r_k.startId \bigwedge o_k.endId > r_k.endId$, making r_k a subrange of o_k, but all subranges have already been removed. Therefore, $cov(Q, R^*_k) \geq cov(Q, O_k)$.

The above proves that for all $k \leq i$, $cov(Q, R^*_k) \geq cov(Q, O_k)$. If R^* is not optimal, then it must be the case that $m > p$, and $cov(Q, O_m) > cov(Q, R^*_p)$. However, this leads to a contradiction since we know through its construction that $cov(Q, R^*_p)$ is the maximum coverage that can be reached by any subset of R. Thus R^* must be optimal. □

Due to the requirement that cached query result-vectors have to be sorted, MaximizeCoverage runs on order of $O(n \log n)$. In our implementation of MaximizeCoverage, we simplify the result-vector filtering and ordering steps shown in line 7 to line 13 (algorithm 1) through the use of a hash map data structure. This map's key-value pairs are of the form $(b_i, [r_{a_1}, r_{a_2}, ..., r_{a_t}])$, where $b_i \in B$ and $r_{a_x} \in R \wedge r_{a_x}.startId = b_i$. Simply put, this structure maps bit-vector identifiers to a list of all cached result-vector identifiers whose coverage starts at that column. Further overhead can be eliminated by maintaining the result-vector lists in sorted order, descending on $endId$ (*i.e.*, the result-vector to provides the most coverage first). Though the use of a hash map makes the theoretical asymptotic upper bound $O(n^2)$, in practicality, it reduces the number of result-vectors that have to be investigated to only those with $startId$s in the bounds of the query range while providing near constant-time access to the result-vectors lists.

4 SYSTEM EVALUATION

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 and 64 GB of RAM. Initially, we present a detailed study into the TPC Benchmark H (TPC-H) [1]. This is followed by a case study into a real intrusion-detection data set that was the subject of KDDCup'99 [13].

4.1 TPC-H Data Characteristics

We used the TPC-H 2.18.0 toolkit to generate our data set and workloads. The TPC-H database models an e-commerce organization spanning eight tables. We focused our experiments on the `lineitem` table, which has 6M entries across 16 attributes, and it plays a significant role in many of the benchmark's queries. Within the `lineitem` table, we indexed its `SHIPDATE` attribute because of its high dimensionality and visibility in common queries. According to TPC-H documentation, the `lineitem.SHIPDATE` attribute ranges from 1992-01-01 to 1998-12-31, spanning 2526 days. Its values are uniformly distributed. To produce our bitmap, we created a bit-vector (column) for each day in the range, dropping a '1' in the column for a `lineitem` row that falls on a given date, and a '0' in all other columns. This process generates a sparse 1.89 GB bitmap index containing 2526 columns and 6M rows.

Each bit-vector is independently compressed using the Word-Aligned Hybrid Code (WAH) method [40], a commonly-used standard. Several properties of WAH are worth noting. It is well documented to excel when compressing highly sparse bitmaps. Interestingly, WAH's query-processing time decreases as function of its compression ratio, so the more aggressively that bit-vectors can be compressed, the faster that a boolean operation can be executed across them. However, because SQL range queries decompose into sequences of boolean OR operators across bit-vectors, as explained earlier in Section 2, the 1-bits will accumulate in the (intermediate) result, frequently disrupting the highly compressible long runs of 0s. Therefore, long-sequences of OR queries may lead to a *"ballooning effect"* on the size of result vectors: worsening query performance and induces pressure on the storage of intermediate results.

4.2 Description of Workloads

We generated multiple workloads based on TPC-H query specifications, which describes a range query to be carried out over the `lineitem` table.

- **Long Confined Ranges (`WL-tpch-q1`)** has 25000 queries that select rows with a `SHIPDATE` as of a specified date: $\sigma_{SHIPDATE \leq (1998/12/01 - \Delta\ days)}(lineitem)$, where Δ is randomly chosen from [60, 120]. This query follows query "Q1" exactly as described in TPC-H benchmark specifications. It has very low selectivity; most bit-vectors participate in processing each query, so cache-less execution is extremely slow. The number of unique queries is very low due to the start and end date restrictions, so results for all queries can easily be cached without replacement.
- **Random Long Ranges (`WL-long-ranges`)** has 25000 queries that select rows with a `SHIPDATE` between 1992/01/01 and 1998/12/31 inclusive. The interval between the start and end dates is at least 1,768 days, or 70% of available dates. The miss penalty can be quite high, due to the average number of participating bit-vectors. Due to the ballooning effect, we anticipate that a low number of result-vectors can be cached, triggering frequent replacement. The high range cardinality also implies that the MaximizeCoverage search algorithm would induce high overhead.
- **Random Short Ranges (`WL-short-ranges`)** executes 25000 queries that select rows with a `SHIPDATE` between 1992/01/01 and 1998/12/31 inclusive. To increase selectivity, the start and end dates are randomly chosen, but are restricted to within 30 days of each other. Very few bit-vectors participate in each query, so the miss penalty is low. The number of uniquely queried ranges is high, triggering cache replacement early and often, which will in turn increase miss rate.

- **Random Mixed Ranges (`WL-mixed-ranges`)** executes 25000 queries that select rows with a `SHIPDATE` between 1992/01/01 and 1998/12/31 inclusive. However, there are no range restrictions for medium selectivity. A mix of short-to-long ranges coexist in this workload. The number of uniquely queried ranges can also be quite high, triggering replacement. Cache entries also vary wildly in size due to WAH's ballooning effect, potentially reducing the number of cacheable entries.

4.3 CLOCK Replacement Policy

After processing each query, we encapsulate the result with its start and end date markers and cache it indiscriminately. That is, all results are placed in the cache, regardless of its corresponding query coverage, size, or any other characteristic. To enforce fixed cache sizes, we implemented the CLOCK replacement algorithm [8], because it famously incurs very little overhead, while approximating the Least Recently Used (LRU) policy. CLOCK maintains a circular list of cache entries that are either marked "hot" or "cold." Upon invocation, a pointer traverses through the entries in clockwise order, downgrading a hot entry to cold or evicting a cold entry (and terminating) when encountered. An entry is marked "hot" again if it is used at any point in time for servicing a query.

The CLOCK algorithm may be invoked several times to create necessary space, evicting a number of cold entries. It is worth noting that we also implemented several other well-known replacement schemes, including Random, LRU, and least-frequently used (LFU), but observed that CLOCK introduced the least amount of space-time overhead while producing consistent results. In the interest of space, we decided to limit all results to using CLOCK.

4.4 Experimental Results

In this subsection, we explore the speedup and limitations from using the proposed caching scheme.

4.4.1 Performance Speedup. Table 3 shows the baseline total times to process each of our workloads over the TPC-H data set *without* caching any result-vectors for reuse. First, we describe the work involved in cache-less workload processing. For a given query, its range of bit-vectors are first retrieved from memory (or disk, upon first encounter), and a boolean OR operation is iteratively executed across these bit-vectors to form a result-vector. Once a bit-vector has been read into memory, it will remain there for the remainder of the workload.

The `WL-tpch-q1` workload requires the longest time to complete, as it requires nearly all bit-vectors to be processed per query. This workload took roughly 2.5 days of continuous execution, averaging 7.96s per query. The `WL-long-ranges` workload required nearly 2 days of processing, or 6.73s per query. Next, because its ranges were restricted to no more than 30 days, the `WL-short-ranges` workload only required 0.017s on average per query, and it completed in just over 7 minutes. Finally, the `WL-mixed-ranges` workload required 2.75s per query, and completed in 19.13 hours.

In the caching-supported runs, we fixed the cache capacity to 256mb, 512mb, 1gb, and 2gb. Larger caches were also examined, but not shown due to minimal/diminishing gains in performance. The CLOCK replacement scheme is run to ensure that these capacities are not breached. Results for total workload processing time and speedup are displayed in Tables 4 and Table 5, respectively.

Workload	Total Time
`WL-tpch-q1`	199,022.7 s
`WL-long-ranges`	168,191.6 s
`WL-short-ranges`	429.2 s
`WL-mixed-ranges`	68,855.0 s

Table 3: Workload Processing Time: Cache-Less

	Cache Capacity			
Workload	256mb	512mb	1gb	2gb
`WL-tpch-q1`	158.0 s	158.0 s	157.7 s	158.0 s
`WL-long-ranges`	10521.6 s	7910.4 s	5754.8 s	4478.6 s
`WL-short-ranges`	208.0 s	185.6 s	171.3 s	167.84 s
`WL-mixed-ranges`	1253.8 s	781.4 s	582.5 s	481.9 s

Table 4: Workload Processing Time: With Caching

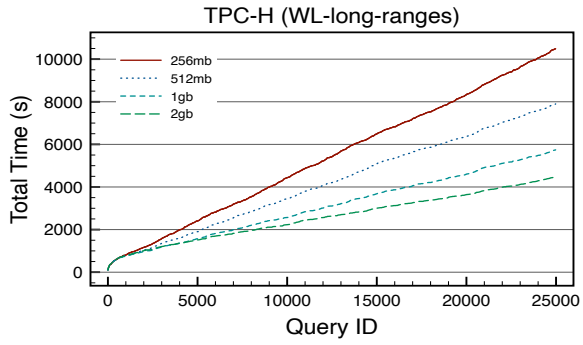
	Cache Capacity			
Workload	256mb	512mb	1gb	2gb
`WL-tpch-q1`	1514.2 ×	1514.2 ×	1517.5 ×	1514.2 ×
`WL-long-ranges`	16 ×	21.3 ×	29.2 ×	37.6 ×
`WL-short-ranges`	2.06 ×	2.31 ×	2.50 ×	2.56 ×
`WL-mixed-ranges`	54.91 ×	88.12 ×	118.22 ×	142.88 ×

Table 5: Speedup over Cache-Less Execution

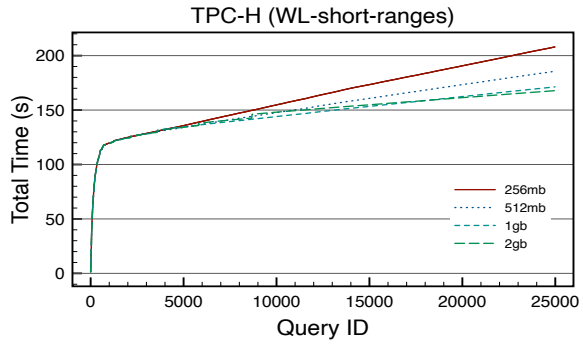
Unsurprisingly, the `WL-tpch-q1` (the specified TPC-H Q1 workload) delivered hyperbolic gains, achieving over 1500× speedup, as shown in Table 5. Increases in capacity have little to no impact — due to the considerably low numbers of unique ranges queried, all possible results are easily stored without replacement. In fact we found that less than 1% of queries engage in any misses, and once the cache has been filled with the 60 unique entries, all remaining queries are fully covered. Given the predictable nature of `WL-tpch-q1`, and the unlikelihood that this *best case scenario* would be commonly observed in practice, we will focus on the remaining three workloads for the remainder of this paper.

The `WL-long-ranges` workload saw more realistic speedups. The large gap injected between the start and end dates in this workload means that the potential for range overlap and reuse would be limited, and a cache-miss in this workload would add more severe penalties due to the average range size that require processing. The `WL-short-ranges` workload was indeed representative of the worst-case usage of our cache. Due to the shortness and randomness of its ranges, there were more cache misses compared to the other workloads. Moreover, because this workload's cache-less query processing times were already fast, it was harder to extract significant performance improvements. We also observe that, even though the caches fill to capacity, increasing the cache capacity only results in incremental gains. We explain this in the next subsection.

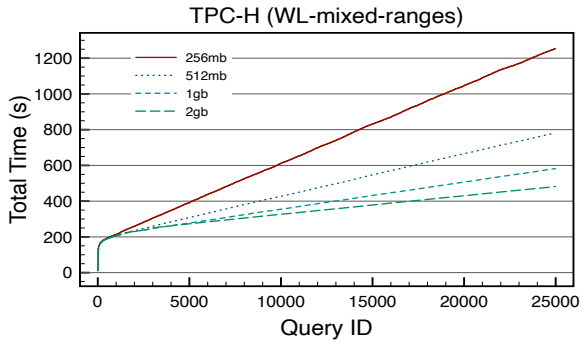
The `WL-mixed-ranges` workload observes excellent speedup and saw the most benefits from increasing the cache capacities. This is due to the relaxing of date-range restrictions, as compared to the previous workload. Result-vectors from resolving these queries vary considerably in size, which causes the cache to reach capacity



(a) Long Ranges Workload over TPC-H



(b) Short Ranges Workload over TPC-H



(c) Mixed Ranges Workload over TPC-H

Figure 3: Workload Processing Time (TPC-H)

quite early. This has the added effect of cache replacement also being invoked early and often through the lifetime of the workload. However, due to the diversity of the result-vectors, we observe a significant amount of reuse.

Workload execution times are also tracked in more detail in the Figure 3 plots. The horizontal axis denotes the queries in the order in which they were dispatched and executed, and the vertical axis tracks the accumulated execution time. The effectiveness of a cache would be indicated by a steep climb (to fill the cache initially), followed by a plateauing effect. Figure 3a plots the `WL-long-ranges` running time, which steadily reduces (albeit at a diminishing rate) given larger cache capacities. Speedups occur only a few hundred queries into the workload, as the cache fills to capacity relatively early. Next, Figure 3b shows the running time for `WL-short-ranges`. After the initial climb to the plateau, the cache reaches a sustained level of reuse at approximately the 250th query. The remaining workload beyond that point continues to climb, but only at a sublinear rate. The average cache entry is quite small in this workload due to the smaller allowable range of the queries. The separation in the four cache-capacity settings diminishes as the capacities increase, due to the miss penalty being less significant in this workload.

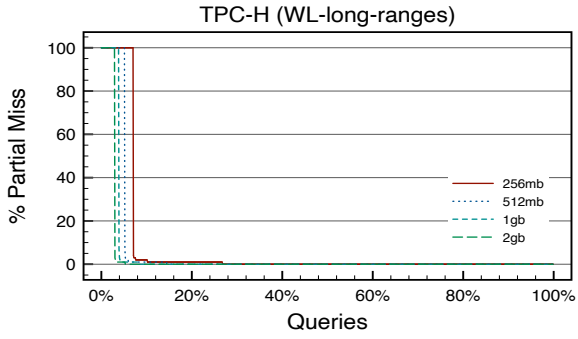
Finally, the total time for `WL-mixed-ranges` is shown in Figure 3c. Here, we can more clearly see the benefits from increasing cache capacities, resulting in far more separation after the cache is initially filled. This is because the common query in this workload requires a medium to high number of bit-vectors to resolve. This also implies that the average cache entry is likely quite large in this workload due to the ballooning effect. Therefore, the miss penalty in this workload is likely to be quite high on average.

4.4.2 Cache Misses and Penalties. In this subsection, we further explore the caching behavior under each workload. The plots in Figure 4 show the distribution of cache misses for all queries.

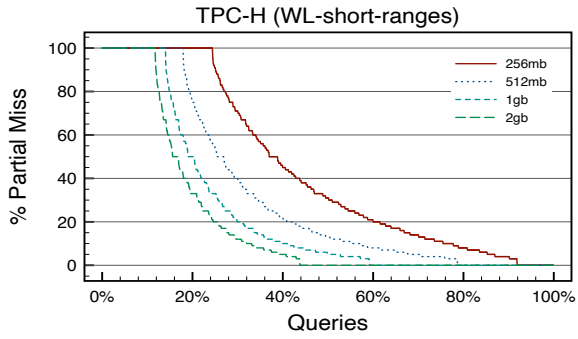
We define a *partial miss* in our cache to be the size of a remainder query, *i.e.*, the number of bit-vectors left uncovered after fetching the cached entries. The partial miss rate for a query is defined as $\frac{|Q|-|Q^r|}{|Q|}$, where Q is the queried set of bit-vector IDs, and Q^r is the set of bit-vector IDs covered by the fetched vectors. For instance, suppose we were answering a query $\{4, 5, 6, 7, 8\}$ and cached result vectors were found to form $\{5, 6, 7\}$, then this example would constitute a 40% partial miss. Lower partial miss rates generally correspond to lower miss penalty since fewer remainder bit-vectors must be processed.

Figure 4 plots the cache miss distribution for the three workloads. To avoid confusion, note that the horizontal axis in these plots is *not* ordered on QueryID as before. For clarity, we sorted all queries in descending order of its partial miss rate so that we can have a sense of its distribution across the workload. A sharp and early descent for this metric would therefore be highly desirable. The results in these figures should be viewed concurrently with Figure 5, which shows the query execution times over the number of columns missed in the smallest cache capacity, 256mb.

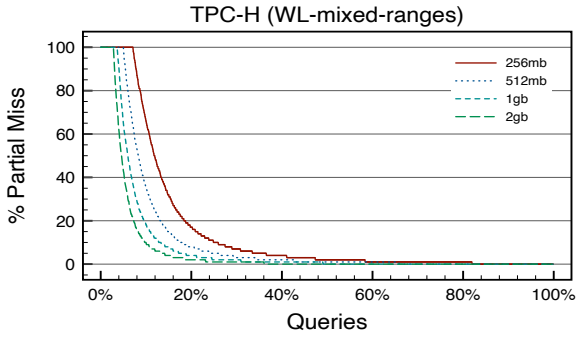
The partial-miss distribution for `WL-long-ranges` is shown in Figure 4a. The waterfall appearance in this figure is due to the long query structure. This workload requires that the queries cover at least 1,768 consecutive days, which means that as partial results are cached, most of the cache misses are limited to a few columns at the ends of the query sequences. This overlap leads to a high miss rate for a limited number of queries, primarily when the cache is cold, and a low miss rate for the majority of the others. Given the remarkably low total miss rate, it is surprising that speedups in the previous section were not more pronounced. Figure 5a gives us a clue. We can see that the misses are concentrated in two regions. The lower region, accounting for 93% of the queries involves few remainder vectors, and thus, less miss penalty and execution time. The performance limitation can be observed in the higher region, which only accounts for 7% of misses. However, noting that the vertical-axis is plotted in log-scale, we can clearly see the exponential increase in query processing time as a function of columns



(a) Large Ranges Workload over TPC-H

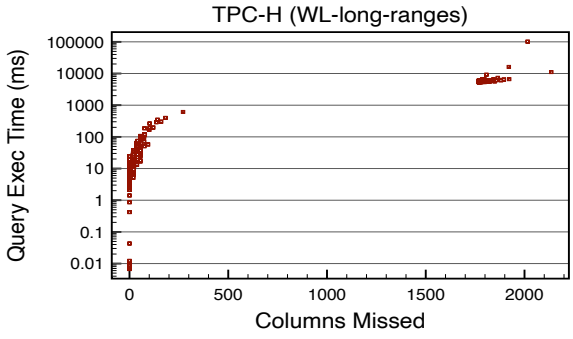


(b) Short Ranges Workload over TPC-H

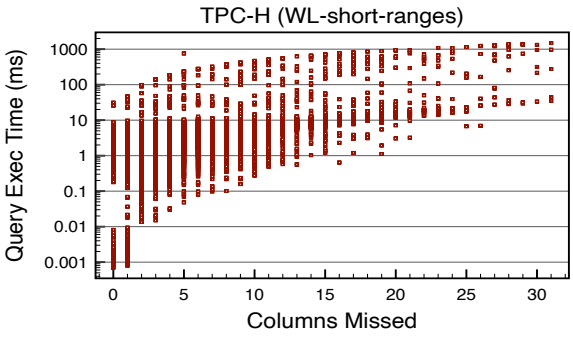


(c) Mixed Ranges Workload over TPC-H

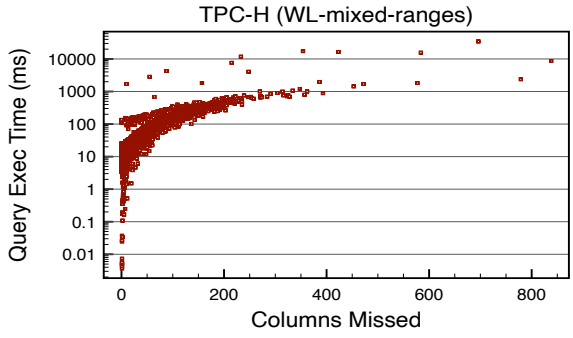
Figure 4: Distribution of Partial Misses (TPC-H)



(a) Long Ranges Workload over TPC-H (256mb capacity)



(b) Short Ranges Workload over TPC-H (256mb capacity)



(c) Mixed Ranges Workload over TPC-H (256mb capacity)

Figure 5: Query Time vs. Columns Missed (TPC-H)

missed. Because the ranges are so long, the few queries with high miss rates dominate total execution time.

The partial-miss distribution is flatter for `WL-short-ranges` (Figure 4b), where we observe a gentle decline. The abundance of short ranges in this workload induce a significant number of full and partial misses. Full misses constitute 12% of all queries in the 2gb case, and up to 24.5% of queries in the 256mb cache. It only stabilizes at 0% miss rate after a significant portion of the workload is already processed (over 90% of the workload in the 256mb case). It is interesting to see that increasing cache capacities *does* significantly diminish the miss rates, but as we saw earlier in Figure 3, this did not translate to appreciable performance speedup. Figure 5b shows that, because the ranges are limited to intervals of 30 days, difference in miss penalty is overall insignificant. This explains why `WL-short-ranges` is unable to achieve over 2× speedup, but we are careful to point out that a 2× speedup is still a significant result.

Figure 4c shows the partial-miss distribution for `WL-mixed-ranges`, in which we again observe steeper declines. This is due in part to the blend of both shorter and longer ranges introducing more opportunities for reuse. The partial miss rate drops to near-zero levels for the vast majority (70% to 80%) of queries in the workload. As we see in the juxtaposition of Figure 5c, the number of missed columns is concentrated toward the lower end, where we only see miss penalties on the order of 100-1000ms, achieving more impressive overhead speedup. The number of queries that required higher execution time is very low compared to `WL-long-ranges`, which

also saw fast drops in miss rates, but had far more limited speedup in comparison.

4.4.3 Space-Time Overhead. The space overhead is relatively straightforward, as we varied the cache capacities in all experiments. One observation is that, especially in the `WL-short-ranges` workload, the doubling of cache capacity leads to diminishing returns in performance. Therefore, even a small cache could achieve impressive benefits. "Right-sizing" the cache would be workload-dependent, and could be a direction for future work.

Two dominant time overheads are search time (*i.e.*, by running the MaximizeCoverage algorithm) and replacement time, shown in Figure 6. Figure 6a and Figure 6c show similar profiles for long and mixed-sized range queries. The search time clearly dominates replacement time in all cases. The slowdown in search time can be traced back to the size of the search space under these workloads. Given the larger search overhead, we emphasize that it is still negligible (0.07% to 0.5% of the total execution time).

The `WL-short-ranges` (Figure 6b) differs quite drastically from the previous two workloads. Here the replacement time constitutes a significant overhead compared to the search time. Compared to the previous workloads, the search time is also far less significant. As the capacities increase, we can also observe an expected amortization in total replacement time, as well as the search time increasing corresponding to increases in cache capacity (MaximizeCoverage search space). However, the search space is generally quite small given the short ranges in this workload. Combined overhead constitute only 0.02% – 0.06% of total execution time.

4.5 Case Study: kddcup99 Data Set

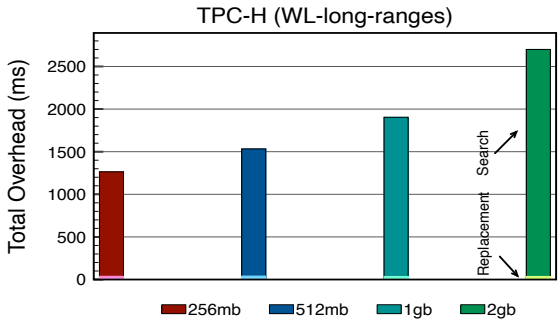
For our case study involving a real data set, we examine network intrusion data, which was subject of the KDDCup'99 competition, and has been made publicly available via the UCI ML Repository [13]. The `kddcup99` data contains 4,898,431 rows across 42 attributes. We indexed 19 continuous attributes using bitmaps, and for each of them, we discretized their values into 25 equal-depth bins, resulting in a 475-column bitmap index.

Cache Capacity				
32mb	64mb	128mb	256mb	512mb
2.09 ×	2.48 ×	2.63 ×	2.86 ×	2.96 ×

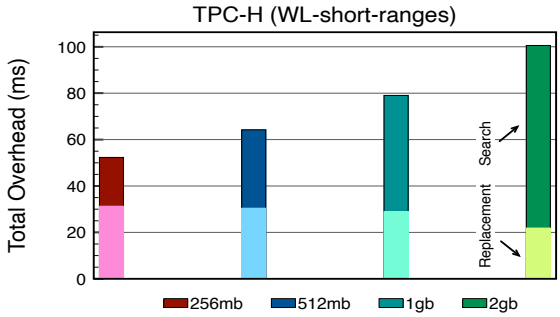
Table 6: Speedup over Cache-Less Execution (kddcup99)

For this case study, we were interested in exploring the worst case, so a workload that is similar to `WL-short-ranges` was generated as follows: each query first randomly selects one of the attributes, and within that attribute, we randomly choose a start and an end column ID. Therefore the max size per range query is 25. For consistency with previous experiments, we also affixed the size of this workload to 25,000 queries, as before.

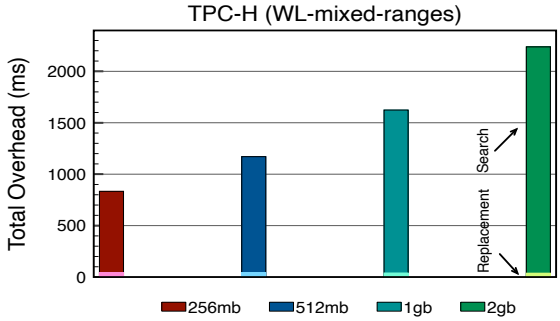
First, we examine the workload speedup over different cache capacities. The cache-less execution of the `kddcup99` workload required 129.34s, and the speedups over the 25,000 query workload are listed in Table 6. The accumulated time results are shown in Figure 7. Our experiments showed negligible gains after the 512mb capacity, so we limit our results to the 32mb and 512mb range. As can be seen, the speedup does tend to track `WL-short-ranges` in



(a) Long Ranges Workload over TPC-H



(b) Short Ranges Workload over TPC-H



(c) Mixed Ranges Workload over TPC-H

Figure 6: Caching Overhead (TPC-H)

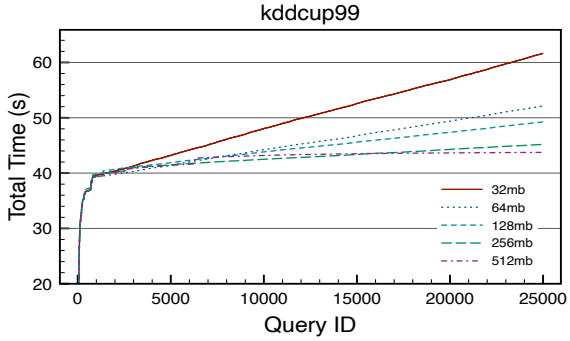


Figure 7: Workload Processing Time (kddcup99)

the TPC-H experiments, as we see gains in the $2-3\times$ range and observe diminishing returns with doubling capacity.

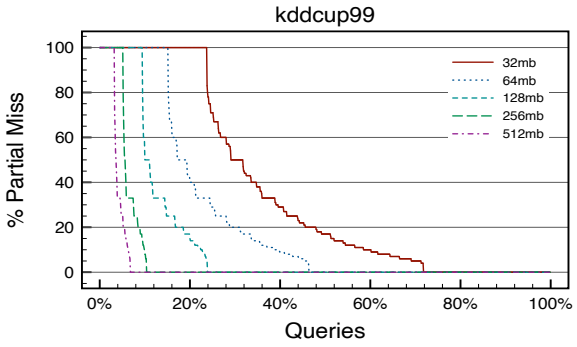


Figure 8: Distribution of Partial Misses (kddcup99)

Partial-miss distribution of this data set and workload is shown in Figure 8. Except for the smallest capacity (32mb), the miss rates for all other settings fall rather precipitously, indicating a high rate of reuse. This is probably due to the fact that *firm* boundaries exist between `kddcup99` attributes from which the query generator must draw random start and end IDs. These boundaries were nonexistent in TPC-H, because we had only indexed one attribute, giving the query generator much higher degree of freedom in choosing the start and end IDs.

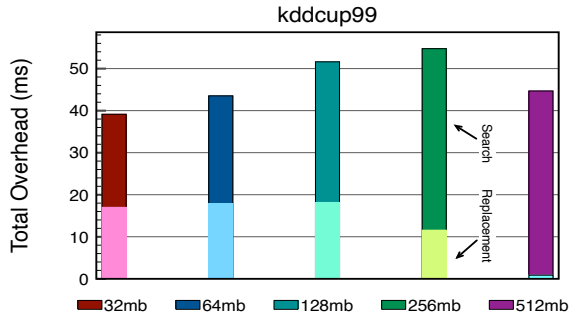


Figure 9: Caching Overhead (kddcup99)

Finally, we show the overheads in Figure 9. The overheads are insignificant as before, accounting for less than a fraction of a percent of the cache-enabled execution times. Similar to what we saw in the TPC-H `WL-short-ranges` result, search times tend to grow only linearly as the cache size doubles, while the replacement time diminishes due to fewer invocations given the larger capacities.

4.6 Summary of Results

Having rigorously stress-tested our caching framework using the TPC-H benchmark and a real data set, we make several overarching conclusions. First, and least surprisingly, intentionally caching results for systematic reuse is highly effective for accelerating range queries over bitmaps. In the worst case, we observe a 2-3 time speedup, while the average and best case workloads observe speedup in the 100-1000 time factor over only a 25,000 query workload. Given that the function scales, the speedup would increase given larger workloads.

Second, despite the result-vectors' size "ballooning effect" over large range queries, even modest cache capacities can achieve respectable performance gains. We observed this perhaps most abundantly in TPC-H's `WL-mixed-ranges` experiment. Only a few 100 to 1000 entries could be stored because the average result-vector size was high, invoking frequent replacement. Yet, we still observed $50\times$ to $142\times$ speedup even under heightened replacement pressure.

Finally, replacement and search overheads are negligible, even as capacities and the number of entries undergo exponential growth. Finally, an area where we might expect our cache to struggle more is with short, random ranges with high miss penalties. In our experiments such a workload observes higher (partial) miss rates. However, in our data sets, the short ranges has also meant lower miss penalty due to the sparseness of our index. We might expect only extracting nominal speedup if the common bit-vector was not sparse, increasing miss penalties, but such a phenomenon would occur quite rarely in proper bitmap index setups.

5 RELATED WORK

The use of cached intermediate results is a well-documented area in query optimization. Chen and Roussopoulos presented the ADMS optimizer, which used data and pointer caching of intermediate results to aid in query processing [4]. Roy, *et al.* present several heuristics which use cached results to increase the efficiency of multi-query optimization [32]. Semantic caches were popularized in distributed databases as a method to increase performance and security in client-server systems [10, 22]. Dar, *et al.* proposed the semantic-caching architecture, in which clients cache results locally [10]. The probe and remainder queries were also described as those that would be posed to the client's local cache, and if exists, one to be the processed at the server. The authors also defined *semantic regions*, which are used to inform on a replacement policy.

Godfrey and Gryz formalize the semantic relationships between cache and query expressions, and give ways to express the semantic existential, independence, overlapping, and remainder properties [16, 17]. These seminal works were followed by applications of semantic caching in other distributed computing domains, including data integration of web sources [6, 24] and mobile-computing environments [25, 31]. Beyond the relational model, work on semantic caching has also extended into other domains. For instance, D'Orazio, *et al.* describe a semantic-cache enabled grid/cloud middleware for object retrieval [12]. *XCache* and *XCacher* both extend semantic caching to XML databases and *XQuery* [5, 20].

The bitmap compression scheme considered in this paper is WAH [40]. However, there are many other run-length compression schemes explicitly designed for bitmap indices. One of the earliest approaches was the Byte-aligned Bitmap Compression (BBC) [2] which uses byte-alignment. There are many others are variations of the word-aligned hybrid codes similar to WAH (e.g., [7, 11, 15, 41]). Other schemes, like [9], [18], and [37] use variable word-alignment. The approaches listed above have a more computationally complex query algorithm than WAH. This increased overhead suggests that

if these schemes were coupled with our caching approach, they would experience similar to better speedups as WAH.

Several other works have investigated techniques to increase the efficiency of range query processing using bitmap indices. Wu, *et al.* [39] used a size ordered priority queue to sequence the column processing of WAH and BBC compressed bitmap range queries. Their empirical study showed that this approach requires less core memory and often performed better than a random sequence of column processing. Additionally, they explored an in-place query algorithm, in which the largest column was decompressed and used to start all intermediate results. This approach was shown to be faster than the priority queue algorithm but required more memory. Slechta, *et al.* [33] explored several similar column ordering techniques for the range query processing of Variable-Aligned Length (VAL) [18] compressed bitmaps. Their most efficient ordering approach was able to achieve 2× speedup over the randomly ordered baseline. Nelson, *et al.* [28] presented several GPU based algorithms for the processing of WAH range queries. They were able to achieve an average of 30× speedup over a parallel CPU algorithm. To the best of our knowledge, we are the first to use a result-vector caching system to increase the efficiency of bitmap range queries.

6 FUTURE WORK AND CONCLUSION

In this paper we describe techniques for integrating a caching scheme in a bitmap-processing system for accelerating range queries. The system organizes, manages, and integrates partial results for query processing. We considered the optimization problem of identifying the minimal number of result-vectors that offer the widest range-query coverage, and presented an optimal algorithm to solve this problem in a restricted form. An extensive system evaluation was performed on various workloads and cache sizes, which demonstrated its effectiveness in query performance.

In the future, we plan to work on static and dynamic right-sizing of the cache to meet certain Quality of Service (QoS) constraints. We would also like to support more general and complex query types instead only ranges and point queries. We also plan to integrate this caching framework with the distributed version of our bitmap query processor, reported in [3], wherein we would expect even greater speedup due to the potential for reducing data transfer.

REFERENCES

[1] 2021. TPC-H Decision Support Benchmark. http://www.tpc.org/tpch

[2] Gennady Antoshenkov. 1995. Byte-aligned bitmap compression. In *Data Compression Conference*. IEEE, 476.

[3] 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.

[4] Chungmin Melvin Chen and Nicholas Roussopoulos. 1994. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *International Conference on Extending Database Technology*. 323–336.

[5] 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.

[6] Boris Chidlovskii and Uwe M. Borghoff. 2000. Semantic Caching of Web Queries. *The VLDB Journal* 9, 1 (March 2000), 2–17.

[7] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (2010), 644–650.

[8] F. J. Corbato. 1968. *A Paging Experiment with the Multics System*. Technical Report. Massachusetts Institute of Technology.

[9] Fabian Corrales, David Chiu, and Jason Sawin. 2011. Variable Length Compression for Bitmap Indices. In *Database and Expert Systems Applications*. 381–395.

[10] 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.

[11] 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.

[12] 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.

[13] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[14] Kengo Fujioka, Yukio Uematsu, and Makoto Onizuka. 2008. Application of bitmap index to information retrieval. In *WWW*. ACM, 1109–1110.

[15] Francesco Fusco, Marc Stoecklin, and Michail Vlachos. 2010. Net-Fli: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *VLDB* 3, 2 (2010), 1382–1393.

[16] Parke Godfrey and Jarek Gryz. 1997. Semantic Query Caching for Hetereogeneous Databases. In *Intelligent Access to Heterogeneous Information*. 6.1–6.6.

[17] Parke Godfrey and Jarek Gryz. 1998. Answering Queries by Semantic Caches. In *Database and Expert Systems Applications*. 485–498.

[18] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *International Conference on Data Engineering*. IEEE, 484–495.

[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] R. Karp. 1972. In *Complexity of Computer Computations*. 85–103.

[22] 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.

[23] Kesheng Wu, W. Koegler, J. Chen, and A. Shoshani. 2003. Using bitmap index for interactive exploration of large datasets. In *SSDBM*. 65–74.

[24] 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.

[25] Ken. C. K. Lee, H. V. Leong, and Antonio Si. 1999. Semantic Query Caching in a Mobile Environment. *SIGMOBILE Mob. Comput. Commun. Rev.* 3, 2 (1999), 28–36.

[26] 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).

[27] Alistair Moffat and Justin Zobel. 1996. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems* 14 (1996), 349–379.

[28] 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.

[29] Patrick E. O'Neil. 1989. Model 204 Architecture and Performance. In *International Workshop on High Performance Transaction Systems*. 40–59.

[30] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. 2007. Enabling Real-Time Querying of Live and Historical Stream Data. In *SSDBM*.

[31] Qun Ren and Margaret H. Dunham. 2000. Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. In *International Conference on Mobile Computing and Networking*. 210–221.

[32] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *ACM SIGMOD International Conference on Management of Data*. 249–260.

[33] Ryan Slechta, Jason Sawin, Ben McCamish, David Chiu, and Guadalupe Canahuate. 2014. Optimizing Query Execution for Variable-Aligned Length Compression of Bitmap Indices. In *IDEAS*. 217–226.

[34] Kurt Stockinger and Kesheng Wu. 2006. Bitmap Indices for Data Warehouses. In *In Data Warehouses and OLAP. 2007. IRM*. Press.

[35] 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.

[36] 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.

[37] Sebastiaan J. van Schaik and Oege de Moor. 2011. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *International Conference on Management of Data*. 913–924.

[38] Harry K. T. Wong, Hsiu fen Liu, Frank Olken, Doron Rotem, and Linda Wong. 1985. Bit Transposed Files. In *Proceedings of VLDB*. 448–457.

[39] Kesheng Wu, Ekow Otoo, and Arie Shoshani. 2004. On the Performance of Bitmap Indices for High Cardinality Attributes. In *VLDB*. 24–35.

[40] 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.

[41] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. 2001. *Notes on design and implementation of compressed bit vectors*. Technical Report LBNL/PUB-3161. Lawrence Berkeley National Laboratory.