

Dynamic Bitmap Join Execution with Integrated Range Query Support

Alex Kirner

Mathematics and Computer Science
University of Puget Sound
akirner@alumni.pugetsound.edu

Jason Sawin

Computer and Information Sciences
University of St. Thomas
jason.sawin@stthomas.edu

David Chiu

Mathematics and Computer Science
University of Puget Sound
dchiu@pugetsound.edu

Abstract—Bitmap indices support efficient filtering of large data sets through compressed bitwise operations, but dynamically joining bitmap-indexed tables complicates subsequent range queries because the join result is no longer naturally represented as a conventional bitmap index. In this paper, we propose two algorithms for evaluating range queries over dynamically computed bitmap-join results. *Early Expansion* follows the natural approach of first constructing the complete bitmap representation of the join and then applying a range filter to that result. While straightforward and reusable across subsequent queries, this approach incurs substantial join-materialization costs. We therefore further propose *Deferred Expansion*, which applies the range predicate to the source bitmap before constructing the structures required by the join, avoiding expansion of tuples that cannot appear in the final result.

We prove that *Early* and *Deferred Expansion* produce equivalent query results and evaluate both approaches using TPC-H data set. Our experiments show that *Deferred Expansion* substantially outperforms *Early Expansion* for practical workloads, despite the latter’s ability to reuse its materialized join result across repeated range queries. We further examine how bitmap granularity affects *Deferred Expansion* and show that reducing the dimensionality of the join-key bitmap can produce disproportionately large performance improvements, eventually making the approach competitive with B+Tree-based execution.

I. INTRODUCTION

Bitmap indexing is widely used in analytical, column-oriented storage systems where queries scan large volumes of data with relatively low update rates [1]–[3]. Bitmap indices have several desirable features. First, by representing the underlying data as bit-vectors, they can be directly applied to fast, hardware-supported operators (e.g., $\wedge$, $\vee$, $\neg$) for filtering. Second, bitmap indices are generally sparse, which allows them to be compressed aggressively [4]–[8]. Third, bitmap processing is highly parallelizable [9]–[12], and bitmap-query results can be cached and reused as part of future queries with relative ease [13], [14].

Despite these advantages, operations surrounding joins remain a major weakness of on-the-fly bitmap query processing. Bitmap representations are organized around a tuple’s value membership within individual attributes, rather than around tuple-to-tuple relationships across relations. This makes simple selections and range predicates extremely efficient to perform, since they can often be evaluated using bitwise operations over a small collection of bitmap columns. Joins between two or more tables represented by bitmaps, in contrast, require

matching attribute values *across* relations and propagating those matches back to the corresponding tuples.

Algorithms for performing such joins directly over bitmap indices exist, but their execution is computationally expensive and can require the construction of large intermediate structures. Consequently, existing bitmap-oriented data management systems commonly avoid performing joins dynamically, instead relying on precomputed bitmap-join indices, materialized views, or de-normalizing tables (where multiple tables are unified ahead of time). These approaches move much of the join costs offline, but introduce their own costs in added storage, maintenance, redundancy, and potential for data inconsistency [15]–[18].

The cost of joins is particularly limiting for queries that combine the joined results with subsequent filtering. Consider the common relational algebra expression of the form, $\sigma_{lo \leq r \leq hi}(T_A \bowtie T_B)$, where $T_A \bowtie T_B$ represents the joining of correlated rows between the two tables T_A and T_B . The range-filter operation $\sigma_{lo \leq r \leq hi}$ is later applied over the joined result to further select the rows that satisfy the given range. Range filtering is the type of operation for which bitmap indices are well suited, yet a straightforward execution must first incur the cost of constructing the bitmap join before the range predicate can be applied. Therefore, the expensive join operation prevents the system from taking full advantage of one of bitmap indices’ primary strengths.

In this paper, we explore how to make this join+range query pattern viable for online SQL processing, while remaining within standard bitmap representations. We propose two algorithms. *Early Expansion* follows the natural execution order: It first constructs the complete bitmap-join result and then applies the range predicate over it. *Deferred Expansion* applies the range predicate to the source bitmap prior to performing the expensive expansion structures and procedures used to construct the join. This allows tuples eliminated by the range predicate to be eliminated before they contribute to intermediate join structures.

Both algorithms operate over conventional bitmap indices and require neither precomputed join indices nor obscure modifications to the underlying bitmap index. We prove that *Early* and *Deferred Expansion* produce equivalent results, but show experimentally that *Deferred Expansion* substantially reduces execution cost. We further investigate the effects of

bitmap granularity and bucket alignment, showing additional opportunities for reducing the cost of dynamic bitmap joins. In sum, these optimizations make the dynamically executed bitmap join+range queries online and viable.

Our paper’s contributions include,

- Early Expansion and Deferred Expansion algorithms, for evaluating range queries over dynamically computed bitmap joins.
- A proof that Early and Deferred Expansion produce equivalent query results.
- An evaluation showing that Deferred Expansion substantially reduces the cost of join+range query execution and is competitive with B+Trees under certain conditions.
- An analysis of how bitmap granularity and bucket misalignment affect the performance of joins.

II. BACKGROUND

A bitmap index is a discretization of various attributes, followed by a structural re-encoding of the relation into a set of aligned bitmap columns. Given a relation R of m tuples, we first choose one or more attributes to index. For each indexed attribute, its domain is then partitioned into a collection of *bins*. These bins may correspond to singleton values (e.g., a specific category) or intervals (e.g., numeric ranges). For this work, we assume that the bins form a non-overlapping cover of the domain so that each attribute value maps to exactly one bin, though it should be noted that overlapping bins are quite useful in certain applications [19].

To create a bitmap, we scan the input relation tuple-by-tuple. For each tuple and each indexed attribute, we determine the bin into which the attribute value falls. For that tuple, the bit corresponding to the appropriate bin is set to 1, while all other bits for that attribute are set to 0. Repeating this process across all tuples yields a collection of n bitmap columns, where n . Each bitmap column has length $m = |R|$, i.e., the number of tuples in R . Equivalently, the bitmap index can be viewed as an $m \times n$ binary matrix, where rows correspond to tuples and columns correspond to bins. A 1 at position $[i, j]$ indicates tuple i contains an indexed attribute whose value falls into bin j . In practice, each row is associated with a pointer to the corresponding tuple on disk, allowing the bitmap to act as a filtering index structure rather than a full data storage system.

For example, consider a relation *SiteTemp* storing environmental sensor readings with two attributes, Location and Temperature, given in Table I (left).

<i>SiteTemp</i>			<i>SiteTemp.Location</i>					
	Location	Temperature		A	B	C	D	E
t_1	A	12	t_1	1	0	0	0	0
t_2	B	27	t_2	0	1	0	0	0
t_3	C	33	t_3	0	0	1	0	0
t_4	D	22	t_4	0	0	0	1	0
t_5	E	41	t_5	0	0	0	0	1

Table I

A BITMAP (RIGHT) THAT INDEXES OVER THE LOCATION ATTRIBUTE OF THE SITETEMP TABLE (LEFT).

To construct a bitmap index over, say, the Location attribute, we start by decomposing the location attribute values into the bins: A, B, C, D, and E. Then for each tuple in *SiteTemp*, we indicate a 1 under the bin(s) in which the value of Location falls, and 0s elsewhere. This results in the bitmap index of Location, shown in Table I (right).

Say we wished to return all the rows pertaining to locations A and E, i.e., $\sigma_{(Location=A \vee Location=E)}(SiteTemp)$. To answer this query, we simply extract columns A and E from the bitmap index, and perform a logical $\vee$ operation between them. The resulting column contains 1s in positions whose corresponding row is a candidate for the selection.

A. Bitmap Join Representation

Suppose we store two relations: *SiteTemp*, which stores environmental sensor readings (repeated here for readability), and *SiteInfo*, which stores metadata about each sensor site.

<i>SiteTemp</i>		
	Location	Temperature
t_1	A	12
t_2	B	27
t_3	C	33
t_4	D	22
t_5	E	41

<i>SiteInfo</i>			
	Location	Region	Elevation
s_1	B	Coastal	35
s_2	A	Inland	220
s_3	C	Mountain	910
s_4	A	Inland	180
s_5	C	Mountain	760

The relations can be joined on the shared Location attribute, producing the result below.

<i>SiteTemp</i> $\bowtie$ <i>SiteInfo</i>				
	Location	Temp	Region	Elevation
t_1, s_2	A	12	Inland	220
t_1, s_4	A	12	Inland	180
t_2, s_1	B	27	Coastal	35
t_3, s_3	C	33	Mountain	910
t_3, s_5	C	33	Mountain	760

However, it is yet not clear how this joined result is represented using bitmaps. Consider the corresponding bitmap indices to these tables as given in Table II. Madduri and Wu proposed in [20] the representation in Table II (bottom) for expressing the join result as a bitmap. Note that the rows and columns in this join result now represent tuples in the corresponding tables: $t_i \in SiteTemp$ and $s_j \in SiteInfo$. In this representation, a 1 in coordinate $[i, j]$ indicates that tuples (t_i, s_j) are in the join. It is understood that the underlying system would further fetch the corresponding tuples from disk that participate in the joined product.

<i>SiteTemp.Location</i>						<i>SiteInfo.Location</i>			
	A	B	C	D	E		A	B	C
t_1	1	0	0	0	0	s_1	0	1	0
t_2	0	1	0	0	0	s_2	1	0	0
t_3	0	0	1	0	0	s_3	0	0	1
t_4	0	0	0	1	0	s_4	1	0	0
t_5	0	0	0	0	1	s_5	0	0	1

<i>SiteTemp</i> $\bowtie$ <i>SiteInfo</i>					
	s_1	s_2	s_3	s_4	s_5
t_1	0	1	0	1	0
t_2	1	0	0	0	0
t_3	0	0	1	0	1
t_4	0	0	0	0	0
t_5	0	0	0	0	0

Table II
BITMAP INDEX OF SITETEMP.LOCATION (TOP-LEFT); THE BITMAP FOR SITEINFO.LOCATION (TOP-RIGHT), AND THE JOINED RESULT (BOTTOM).

III. ALGORITHM OVERVIEW

In this section, we explain the mathematical structures and operations required to perform joins over bitmap indices. The structures and processes required to produce the natural join bitmap result fundamentally lie in a special case of the “Kronecker product” [21]. Below, we give an overview of this structure and its associated operations. Afterwards, we show how to decompose the natural join operation into elementary matrix operations over these Kronecker products.

A. The Kronecker Product

The Kronecker product [21], expressed using $\mathbf{A} \otimes \mathbf{B}$, is a matrix operation that observes the following properties. If $\mathbf{A}$ is an $n_1 \times m_1$ matrix, and $\mathbf{B}$ is an $n_2 \times m_2$ matrix, then $\mathbf{A} \otimes \mathbf{B}$ results in an $n_1 n_2 \times m_1 m_2$ matrix which can be represented in the block-matrix form below,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \dots \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \quad (1)$$

In essence, the Kronecker product is an operation that expands two matrices into a larger block matrix. In this paper, we can focus on the restricted case in which $\mathbf{A}$ and $\mathbf{B}$ are bit vectors of lengths n and m , respectively. Specifically, let $\mathbf{A}$ be a column vector and $\mathbf{B}$ be a row vector. Then the result of $\mathbf{A} \otimes \mathbf{B}$ is an $n \times m$ matrix where $(\mathbf{A} \otimes \mathbf{B})[i, j] = \mathbf{A}[i] \cdot \mathbf{B}[j]$. The Kronecker product ultimately performs multiplication, which in the context of bitmaps is equivalent to a logical $\wedge$ operation. Thus, for bit vectors we may equivalently write:

$$(\mathbf{A} \otimes \mathbf{B})[i, j] = \mathbf{A}[i] \wedge \mathbf{B}[j] \quad (2)$$

meaning that the Kronecker product of two bit vectors produces a matrix that effectively acts as a lookup table for determining the value of $\mathbf{A}[i] \wedge \mathbf{B}[j]$. For example,

$$\text{if } \mathbf{A} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}, \text{ then}$$

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

B. Madduri-Wu Bitmap Join Algorithm

In this subsection, we outline the existing algorithm for performing a natural join using exclusively bitmap indices built over the join-key attribute of two relations. This algorithm and related structures first appeared in [20], but we express it here in terms of Kronecker product [21].

First, we define the notation and definitions used throughout the remainder of the paper. Let T_A and T_B denote database tables sharing a common attribute K (i.e., K is the join-key appearing in both tables). The *natural join* between T_A and T_B , written as $T_A \bowtie T_B$, returns a set containing all pairs of tuples $\{(a, b) | a \in T_A \wedge b \in T_B \wedge a.K = b.K\}$. Generally speaking, there may be multiple shared attributes between T_A and T_B , but for the sake of simplicity, we assume that the shared key K consists of a single attribute.

We let $\mathbf{A}$ and $\mathbf{B}$ denote bitmap indices built over some join-key K for relations T_A and T_B , respectively. Furthermore, $\mathbf{A}[k]$ and $\mathbf{B}[k]$ denote the bit-vector columns associated with some distinct key-value $k \in K$. The bitmap representation $\mathbf{J}$ of $T_A \bowtie T_B$ can then be computed as:

$$\mathbf{J} = \bigvee_{k \in \text{dom}(T_A.K) \cap \text{dom}(T_B.K)} \mathbf{A}[k] \otimes \mathbf{B}[k]^T \quad (3)$$

where $\text{dom}(T_A.K)$ and $\text{dom}(T_B.K)$ denote the sets of values appearing for join-key K for tables T_A and T_B , respectively. Equivalently, this operation can be expressed algorithmically, in Algorithm 1.

Algorithm 1 BITMAPJOIN

```

1: Input
2:    $\mathbf{A}$    Bitmap index for the  $T_A$ 's join key
3:    $\mathbf{B}$    Bitmap index for the  $T_B$ 's join key
4: Output
5:    $\mathbf{J}$    Lookup table representing  $T_A \bowtie T_B$ 
6:  $\triangleright$  Initialize  $\mathbf{J}$  to a zero-filled matrix
7:  $\mathbf{J} \leftarrow \mathbf{0}_{\mathbf{A}.numRows \times \mathbf{B}.numRows}$ 
8:  $\triangleright$  Iterate over bitmap columns shared by both tables
9: for  $k \in \text{cols}(\mathbf{A}) \cap \text{cols}(\mathbf{B})$  do
10:   for  $j \leftarrow 1$  to  $\mathbf{A}.numRows$  do
11:      $\triangleright$  Accumulate matches from column  $k$  of  $\mathbf{B}$ 
12:      $\triangleright$  into row  $j$  of  $\mathbf{J}$  when  $\mathbf{A}[j, k] = 1$ 
13:     if  $\mathbf{A}[j, k] = 1$  then
14:        $\mathbf{J}[j, :] \leftarrow \mathbf{J}[j, :] \vee \mathbf{B}[:, k]^T$ 
15: return  $\mathbf{J}$ 

```

As a running example, let us consider two tables, T_A and T_B with a join-key attribute x , shown in Table III.

T_A				T_B			
	x	y	...		x	z	...
t_1	2	...	...	s_1	3	...	...
t_2	3	...	...	s_2	1	...	...
t_3	3	...	...	s_3	3	...	...
t_4	2	...	...	s_4	2	...	...
t_5	4	...	...				

Table III
SAMPLE DATABASE TABLES FOR THE RUNNING EXAMPLE

Assume that attribute x has been bitmap indexed, and that values of $T_A.x$ and $T_B.x$ draw from domains $\text{dom}(T_A.x) = \{2, 3, 4\}$ and $\text{dom}(T_B.x) = \{1, 2, 3\}$, respectively. As stated earlier, $\mathbf{A}$ and $\mathbf{B}$ denote the bitmap indices generated for join-keys $T_A.x$ and $T_B.x$ respectively. Observe that each distinct value of x is assigned its own bit column in the bitmap:

$$\mathbf{A} = \begin{array}{c|c|c} \mathbf{2} & \mathbf{3} & \mathbf{4} \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array} \quad \mathbf{B} = \begin{array}{c|c|c} \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}$$

The algorithm begins by examining the first shared column, “2,” which symbolically represents $x = 2$ in the original tables. It calculates the Kronecker product between the corresponding vectors,

$$\mathbf{A}[2] \otimes \mathbf{B}[2]^T = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The same is done for the next shared column “3,” or conceptually, $x = 3$.

$$\mathbf{A}[3] \otimes \mathbf{B}[3]^T = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

To complete the join, a logical $\vee$ operation is performed across all of the intermediate Kronecker product frames. We call this process “expansion.”

$$\mathbf{J} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \vee \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The joined result $\mathbf{J}$ forms a binary “lookup table” used to identify the tuples that participate in the join (as in bottom Table II). For example, let us examine the entry $\mathbf{J}[2, 1]$, which is a 1. If we return to the original data from Table III, tuple t_2 in table T_A has value $x = 3$, as does tuple s_1 from table

T_B , meaning the pair $(t_2, s_1) \in T_A \bowtie T_B$. Conversely, the entry $\mathbf{J}[2, 2]$ is marked 0 in the join result. Tuple t_2 in T_A has value $x = 3$, but tuple s_2 in T_B has value $x = 1$. Since those values of x do not agree, $(t_2, s_2) \notin T_A \bowtie T_B$. To finalize the join result, the database system therefore simply has to fetch all pairs of tuples in which a 1 is indicated in $\mathbf{J}$.

C. Linearized View of Bitmap Matrices

In our implementation, Kronecker-product frames are linearized into 1D arrays using row-major order. That is, $\mathbf{J}[i, j]$ is actually indexed as $\mathbf{J}[i * m + j]$, where m is the number of columns in the Kronecker product $\mathbf{J}$. In our running example, this means that $\mathbf{J} = [0001 \ 1010 \ 1010 \ 0001 \ 0000]$. The rationale for the decision to linearize $\mathbf{J}$ is to make it amenable to be compressed aggressively using bitmap-compression schemes such as WAH [4] and others [5]–[8], [22]. For the remainder of this paper, the reader should assume that the joined product $\mathbf{J}$ has been linearized in all the pseudocode algorithms that we present. However, when presenting mathematical expressions and proofs, we return to its 2-dimensional matrix representation for clarity and intuition.

IV. RANGE QUERY INTEGRATION

In practical database and analytical workloads, the results of joins are often combined with subsequent filtering operations. In this paper, we consider queries of the form $\sigma_{lo \leq r \leq hi}(T_A \bowtie T_B)$, which can be equivalently expressed in SQL as:

```
select * from TA join TB on TA.k = TB.k
where r between lo and hi;
```

though it should be noted that our approach easily extends to the popular subcase of exact-match queries as well. Before we explain how to prepare our join structures for range queries, we first describe how to fundamentally perform range queries over bitmap indices.

Table IV shows an example bitmap constructed over some “Temperature” attribute. Each possible temperature value forms its own column. To perform a range query over this table for all tuples satisfying $13 \leq \text{Temperature} \leq 16$, we would just need to iteratively apply a logical OR ($\vee$) across columns 13, 14, 15, 16. The result is a single bit vector, which we call the range-filter column. A 1 at position i in this vector indicates that tuple t_i has a temperature value within the specified range. In this example, the resulting filter column is $[0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ \dots]^T$ indicating that tuples $t_2, t_6, t_7, \dots$ satisfy the range.

Mathematically, the process as described involves applying the logical $\vee$ across all columns in the given range, that is,

$$\bigvee_{lo \leq r \leq hi} \mathbf{R}[r]$$

where $\mathbf{R}$ is a set of bitmap columns for the target table. The range-filtering method (Algorithm 2) can be performed by recursively applying the logical $\vee$ operation between non-overlapping pairs of columns until the column set is reduced to a single range-filter column. This recursive approach, in

Temperature							
	...	12	13	14	15	16	...
t_1	...	1	0	0	0	0	...
t_2	...	0	1	0	0	0	...
t_3	...	0	0	0	0	0	...
t_4	...	0	0	0	0	0	...
t_5	...	0	0	0	0	0	...
t_6	...	0	0	0	1	0	...
t_7	...	0	1	0	0	0	...
...				...			

Table IV
BITMAP REPRESENTING SITE TEMPERATURES.

contrast to the iterative approach, helps preserve WAH compression longer by combining narrower and sparser subranges before producing denser intermediate results.

Algorithm 2 RANGEFILTER: Recursive Range Filtering over Bitmap Columns

```

1: Input
2:   R   Bitmap index for the range attribute  $r$ 
3:    $lo$   Lower limit of range query
4:    $hi$   Upper limit of range query
5: Output
6:   RF A bit column of tuples satisfying  $lo \leq r \leq hi$ .
7: if  $lo > hi$  then
8:   return  $\mathbf{0}_{R.numRows \times 1}$ 
9: else if  $lo = hi$  then
10:  return  $\mathbf{R}[lo]$ 
11: else
12:   $m \leftarrow \lfloor (lo + hi) / 2 \rfloor$ 
13:  return  $\text{rangeFilter}(\mathbf{R}, lo, m) \vee$ 
          $\text{rangeFilter}(\mathbf{R}, m + 1, hi)$ 

```

A. Bitmap Join + Range Query Integration

The Madduri-Wu bitmap join (Algorithm 1) produces a linearized bitmap J whose bit positions correspond to tuple pairs $(T_A[i], T_B[j])$. More precisely, J is indexed over the tuple-pair space $T_A \times T_B$, with 1-bits identifying pairs that participate in the join $T_A \bowtie T_B$. Algorithm 2, in contrast, produces a range-result vector RF over the tuples of the relation containing the range attribute, with 1-bits identifying tuples that satisfy the range predicate. Because J is defined over tuple pairs while RF is defined over tuples of a single relation, the two cannot be directly combined with a logical $\wedge$. Instead, RF must first be mapped to the $T_A \times T_B$ space represented by J .

Suppose the range attribute belongs to T_A , and let RF denote the range-result vector produced by Algorithm 2. Each bit $RF[i]$ applies to every tuple pair $(T_A[i], T_B[j])$, for all rows j of T_B . Therefore, to represent the same range predicate over the $T_A \times T_B$ space, each bit of RF must be repeated once for every row of T_B . Let $\vec{1}$ denote an all-ones vector with one entry for each row of T_B . The mapped range result can then be written as $RF \otimes \vec{1}$. If the range attribute instead belongs

to T_B , the analogous mapping is $\vec{1}_{|T_A|} \otimes RF$. In either case, the mapped range-result vector contains one bit for each tuple pair in $T_A \times T_B$, aligning it with the linearized join bitmap J so that the two can be logically ANDed to identify joined tuple pairs that also satisfy the range predicate.

The remaining question is when this filtering should occur relative to Kronecker-product expansion. One approach is to first construct the complete bitmap join result and then apply the mapped range filter. Alternatively, the range predicate can be applied to the source bitmap before expansion, thereby preventing tuples eliminated by the predicate from contributing to the join structure. We refer to these approaches as Early Expansion and Deferred Expansion, respectively.

Early Expansion: This approach first constructs the complete Madduri-Wu bitmap join result J . The range predicate is then evaluated over the source relation containing the range attribute, producing the range-result vector RF . Early Expansion then combines the complete J bitmap with the mapped range result to retain only joined tuple pairs that satisfy the range predicate. We assume that the range attribute r belongs to exactly one of the two relations, T_A or T_B .

Algorithm 3 EARLY EXPANSION RANGE QUERY

```

1: Input
2:   A   Bitmap of table  $T_A$ 
3:   B   Bitmap of table  $T_B$ 
4:   R   Bitmap of the range attribute  $r$ 
5:    $lo$   Lower limit of range query
6:    $hi$   Upper limit of range query
7: Output
8:   X   Bitmap representing  $\sigma_{lo \leq r \leq hi}(T_A \bowtie T_B)$  (linearized)
9:   J  $\leftarrow \text{BitmapJoin}(\mathbf{A}, \mathbf{B})$ 
10:  RF  $\leftarrow \text{RangeFilter}(\mathbf{R}, lo, hi)$ 
11:  X  $\leftarrow ()$ 
12:  for  $i \leftarrow 0$  to  $\mathbf{A.rows} \times \mathbf{B.rows}$  do
13:    if range attribute  $r$  is in table  $T_A$  then
14:       $\mathbf{X.appendBit}(J[i] \wedge \mathbf{RF}[\lfloor \frac{i}{\mathbf{B.rows}} \rfloor])$ 
15:    else
16:       $\mathbf{X.appendBit}(J[i] \wedge \mathbf{RF}[i \bmod \mathbf{B.rows}])$ 
17:  return X

```

The mapped range result does not need to be materialized explicitly. Instead, Algorithm 3 derives the appropriate source-range bit directly from each position in the linearized join bitmap J . Suppose the range attribute r belongs to T_B . For a linearized position p , the corresponding bit of RF is $RF[p \bmod |T_B|]$. Conversely, if r belongs to T_A , the corresponding bit is $RF[\lfloor p/|T_B| \rfloor]$.

Deferred Expansion: In SQL databases, a common optimization is to perform the join+range query out of order, i.e., perform the range filtering first and then apply the join query on the reduced set of tuples. For Deferred Expansion, we assume that T_A contains the range attribute; if necessary, the input relations may be exchanged to satisfy this convention.

In our implementation, the range predicate is evaluated over T_A prior to the Kronecker-product expansion. First, a range-filter column is computed over T_A via Algorithm 2. This filter column is then propagated into the bitmap columns of $\mathbf{A}$ that are about to participate in Kronecker products. Each participating bitmap column is logically ANDed with the range-filter result before expansion occurs. The filtered bitmap columns are then expanded into Kronecker products.

Algorithm 4 DEFERRED EXPANSION RANGE QUERY

```

1: Input
2:    $\mathbf{A}$    Bitmap of table  $T_A$ 
3:    $\mathbf{B}$    Bitmap of table  $T_B$ 
4:    $\mathbf{R}$    Bitmap of the range attribute  $r$ 
5:    $lo$    Lower limit of range query
6:    $hi$    Upper limit of range query
7: Output
8:    $\mathbf{X}$    Bitmap representing  $\sigma_{lo \leq r \leq hi}(T_A \bowtie T_B)$ 
9:  $\mathbf{RF} \leftarrow \text{RangeFilter}(\mathbf{R}, lo, hi)$ 
10:  $\mathbf{A}' \leftarrow \mathbf{A}.\text{clone}()$ 
11: for  $i \leftarrow 0 \dots \mathbf{A}'.\text{numCols}$  do
12:    $\mathbf{A}'[i] \leftarrow \mathbf{A}'[i] \wedge \mathbf{RF}$ 
13:  $\mathbf{X} \leftarrow \text{BitmapJoin}(\mathbf{A}', \mathbf{B})$ 
14: return  $\mathbf{X}$ 

```

B. Equivalence of Early and Deferred Expansion

We now prove that the Early and Deferred Expansion algorithms are equivalent. To establish their equality, we first prove a lemma describing how logical conjunction distributes across Kronecker products of bit vectors.

Lemma 1. *Let $\mathbf{x}, \mathbf{u}$ be bit vectors of equal length, and let $\mathbf{y}, \mathbf{v}$ be bit vectors of equal length. Then*

$$(\mathbf{x} \wedge \mathbf{u}) \otimes (\mathbf{y} \wedge \mathbf{v}) = (\mathbf{x} \otimes \mathbf{y}) \wedge (\mathbf{u} \otimes \mathbf{v}).$$

Proof. Consider arbitrary indices i and j . By the definition of the Kronecker product over bit vectors,

$$((\mathbf{x} \wedge \mathbf{u}) \otimes (\mathbf{y} \wedge \mathbf{v}))[i, j] = (\mathbf{x}[i] \wedge \mathbf{u}[i]) \wedge (\mathbf{y}[j] \wedge \mathbf{v}[j]).$$

By associativity and commutativity of $\wedge$,

$$(\mathbf{x}[i] \wedge \mathbf{u}[i]) \wedge (\mathbf{y}[j] \wedge \mathbf{v}[j]) = (\mathbf{x}[i] \wedge \mathbf{y}[j]) \wedge (\mathbf{u}[i] \wedge \mathbf{v}[j]).$$

Using the definition of the Kronecker product again,

$$(\mathbf{x}[i] \wedge \mathbf{y}[j]) \wedge (\mathbf{u}[i] \wedge \mathbf{v}[j]) = (\mathbf{x} \otimes \mathbf{y})[i, j] \wedge (\mathbf{u} \otimes \mathbf{v})[i, j].$$

Therefore,

$$((\mathbf{x} \wedge \mathbf{u}) \otimes (\mathbf{y} \wedge \mathbf{v}))[i, j] = ((\mathbf{x} \otimes \mathbf{y}) \wedge (\mathbf{u} \otimes \mathbf{v}))[i, j].$$

Since this holds for every i and j ,

$$(\mathbf{x} \wedge \mathbf{u}) \otimes (\mathbf{y} \wedge \mathbf{v}) = (\mathbf{x} \otimes \mathbf{y}) \wedge (\mathbf{u} \otimes \mathbf{v}).$$

□

We can now prove the equivalence of Early and Deferred Expansion.

Theorem 1. *Given database tables T_A and T_B with a common join key K , and bounds lo and hi limiting a range key r in table T_A , Early Expansion (Algorithm 3) and Deferred Expansion (Algorithm 4) return the same value.*

Proof. As before, let $\mathbf{A}$ be a bitmap built over the join key in table T_A , $\mathbf{B}$ denote the bitmap index over the join-key of T_B . We let $K = \text{dom}(T_A.k) \cap \text{dom}(T_B.k)$ denote the common join-key values between the two tables. Let $\mathbf{R}_A$ denote the bitmap index over the range attribute r of T_A . We define $\mathbf{RF} = \text{RangeFilter}(\mathbf{R}_A, lo, hi)$ as the range-result vector obtained by filtering attribute r in T_A .

Early Expansion first constructs the complete join bitmap $\mathbf{J}$. It then evaluates the range predicate over the source relation containing the range attribute, producing the range-result vector $\mathbf{RF}$, and maps $\mathbf{RF}$ into the tuple-pair space of $\mathbf{J}$. The final result is obtained by logically ANDing $\mathbf{J}$ with this mapped range result. Let $\vec{\mathbf{1}}$ denote the all-ones vector with one entry for each row of T_B . Then the range-result vector mapped to the tuple-pair space of $\mathbf{J}$ is $\mathbf{RF} \otimes \vec{\mathbf{1}}$. The join bitmap $\mathbf{J}$ is the result of repeated $\vee$'s between Kronecker products of matching columns from $\mathbf{A}$ and $\mathbf{B}$. If $\mathbf{A}[i]$ is the bitmap column for the join-key value i , then we can express the join result as $(\mathbf{A}[1] \otimes \mathbf{B}[1]) \vee \dots \vee (\mathbf{A}[n] \otimes \mathbf{B}[n])$. Thus Early Expansion is defined by the expression,

$$((\mathbf{A}[1] \otimes \mathbf{B}[1]) \vee \dots \vee (\mathbf{A}[n] \otimes \mathbf{B}[n])) \wedge (\mathbf{RF} \otimes \vec{\mathbf{1}}) \quad (4)$$

Applying boolean distributivity, we can express this instead as $((\mathbf{A}[1] \otimes \mathbf{B}[1]) \wedge (\mathbf{RF} \otimes \vec{\mathbf{1}})) \vee \dots \vee ((\mathbf{A}[n] \otimes \mathbf{B}[n]) \wedge (\mathbf{RF} \otimes \vec{\mathbf{1}}))$, which can be written as:

$$\bigvee_{k \in K} (\mathbf{A}[k] \otimes \mathbf{B}[k]) \wedge (\mathbf{RF} \otimes \vec{\mathbf{1}}) \quad (5)$$

where K is the set of all key values present in both $\mathbf{A}$ and $\mathbf{B}$.

Deferred Expansion performs a logical $\wedge$ between the RANGEFILTER taken over $\mathbf{A}$ and each individual column, meaning that it can be expressed as $((\mathbf{A}[1] \wedge \mathbf{RF}) \otimes (\mathbf{B}[1] \vee \dots \vee ((\mathbf{A}[n] \wedge \mathbf{RF}) \otimes \mathbf{B}[n])))$, or:

$$\bigvee_{k \in K} (\mathbf{A}[k] \wedge \mathbf{RF}) \otimes \mathbf{B}[k] \quad (6)$$

Because $\vec{\mathbf{1}}$ is an all-ones vector, $\mathbf{B}[k] \wedge \vec{\mathbf{1}} = \mathbf{B}[k]$. Therefore, $(\mathbf{A}[k] \wedge \mathbf{RF}) \otimes \mathbf{B}[k] = (\mathbf{A}[k] \wedge \mathbf{RF}) \otimes (\mathbf{B}[k] \wedge \vec{\mathbf{1}})$. By Lemma 1, $(\mathbf{A}[k] \wedge \mathbf{RF}) \otimes (\mathbf{B}[k] \wedge \vec{\mathbf{1}}) = (\mathbf{A}[k] \otimes \mathbf{B}[k]) \wedge (\mathbf{RF} \otimes \vec{\mathbf{1}})$. Therefore,

$$\bigvee_{k \in K} (\mathbf{A}[k] \otimes \mathbf{B}[k]) \wedge (\mathbf{RF} \otimes \vec{\mathbf{1}}) = \bigvee_{k \in K} (\mathbf{A}[k] \wedge \mathbf{RF}) \otimes \mathbf{B}[k] \quad (7)$$

Hence, Early and Deferred Expansion return the same value. □

C. Complexity Analysis

Let $N_A = |T_A|$ and $N_B = |T_B|$ denote the number of rows in T_A and T_B , and let N_K denote the number of join-key bitmap columns shared by $\mathbf{A}$ and $\mathbf{B}$. Algorithm 1 initializes the $N_A \times N_B$ join bitmap $\mathbf{J}$, requiring $O(N_A N_B)$ time, and

scans all $N_K N_A$ entries of the shared join-key columns in $\mathbf{A}$. When $\mathbf{A}[j, k] = 1$, a length- N_B bitmap column from $\mathbf{B}$ is ORed into row j of $\mathbf{J}$. Under the non-overlapping bitmap representation assumed here, each row can trigger at most one such OR across the shared join-key columns. Thus, the worst-case cost of constructing $\mathbf{J}$ is $O(N_A N_B + N_K N_A)$.

Both Early and Deferred Expansion additionally evaluate the range predicate using Algorithm 2. If the requested range spans N_Q bitmap columns over a relation containing N_R rows, constructing $\mathbf{RF}$ requires $N_Q - 1$ logical OR operations over length- N_R vectors, giving $O(N_Q N_R)$ sequential work. Because this cost is common to both approaches, the primary difference lies in how $\mathbf{RF}$ is applied to the join. Early Expansion constructs the complete join bitmap and then scans all $N_A N_B$ positions of the linearized result to apply the mapped range filter. This additional scan does not change the asymptotic bound.

For Deferred Expansion, we assume T_A contains the range attribute, and thus $\mathbf{A}$ is first cloned, requiring $N_K N_A$ cost. Rather than applying the range result across the expanded $N_A N_B$ tuple-pair space, $\mathbf{RF}$ is ANDed directly with each of the N_K participating join-key columns of $\mathbf{A}$, requiring $O(N_K N_A)$ work before the join. This avoids the $O(N_A N_B)$ post-join range-application scan required by Early Expansion. In addition, filtering reduces the number of 1-bits in the resulting bitmap $\mathbf{A}'$. Consequently, the condition $\mathbf{A}'[j, k] = 1$ in the result of $\text{BitmapJoin}(\mathbf{A}', \mathbf{B})$ is satisfied less often, causing fewer length- N_B OR operations to be performed during join construction. The increased sparsity of $\mathbf{A}'$ may also improve bitmap compression, further reducing the amount of data processed.

D. Handling Misalignment of Bucketed Bitmaps

In this subsection we address the problem where the granularity of the bitmap columns differs between two tables. When there are too many discrete values for an attribute, systems commonly quantize multiple values into a singular “bucket.” This has the downside of producing false positives, but it significantly reduces storage costs. If a column in bitmap index $\mathbf{A}$ represents all values in the range $[0, 2]$, then that column must participate in Kronecker products with every bitmap column in index $\mathbf{B}$ whose bucket overlaps this interval.

This requires: $(\mathbf{A}_1 \otimes \mathbf{B}_1) \vee (\mathbf{A}_1 \otimes \mathbf{B}_2)$. However, we can reduce the number of Kronecker products by exploiting distributivity via Lemma 2, below.

Lemma 2. *For bit vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, where $|\mathbf{y}| = |\mathbf{z}|$, and a boolean operation $\circ$ that obeys the distributive property in combination with the $\wedge$ operator,*

$$(\mathbf{x} \otimes \mathbf{y}) \circ (\mathbf{x} \otimes \mathbf{z}) = \mathbf{x} \otimes (\mathbf{y} \circ \mathbf{z}) \quad (8)$$

Proof. Consider an arbitrary coordinate $[i, j]$. By definition of boolean $\circ$,

$$((\mathbf{x} \otimes \mathbf{y}) \circ (\mathbf{x} \otimes \mathbf{z}))[i, j] = (\mathbf{x} \otimes \mathbf{y})[i, j] \circ (\mathbf{x} \otimes \mathbf{z})[i, j] \quad (9)$$

Using the definition of the Kronecker product (Equation 2), we have both:

$$\begin{aligned} (\mathbf{x} \otimes \mathbf{y})[i, j] &= \mathbf{x}[i] \wedge \mathbf{y}[j] \\ (\mathbf{x} \otimes \mathbf{z})[i, j] &= \mathbf{x}[i] \wedge \mathbf{z}[j] \end{aligned} \quad (10)$$

Rewriting after substituting for each,

$$((\mathbf{x} \otimes \mathbf{y}) \circ (\mathbf{x} \otimes \mathbf{z}))[i, j] = (\mathbf{x}[i] \wedge \mathbf{y}[j]) \circ (\mathbf{x}[i] \wedge \mathbf{z}[j]) \quad (11)$$

By distributivity of boolean algebra,

$$(\mathbf{x}[i] \wedge \mathbf{y}[j]) \circ (\mathbf{x}[i] \wedge \mathbf{z}[j]) = \mathbf{x}[i] \wedge (\mathbf{y}[j] \circ \mathbf{z}[j]) \quad (12)$$

Additionally,

$$\mathbf{y}[j] \circ \mathbf{z}[j] = (\mathbf{y} \circ \mathbf{z})[j] \quad (13)$$

Thus,

$$((\mathbf{x} \otimes \mathbf{y}) \circ (\mathbf{x} \otimes \mathbf{z}))[i, j] = \mathbf{x}[i] \wedge (\mathbf{y} \circ \mathbf{z})[j] \quad (14)$$

Finally, by definition of the Kronecker product,

$$\mathbf{x}[i] \wedge (\mathbf{y} \circ \mathbf{z})[j] = (\mathbf{x} \otimes (\mathbf{y} \circ \mathbf{z}))[i, j] \quad (15)$$

Therefore, for every coordinate $[i, j]$,

$$((\mathbf{x} \otimes \mathbf{y}) \circ (\mathbf{x} \otimes \mathbf{z}))[i, j] = (\mathbf{x} \otimes (\mathbf{y} \circ \mathbf{z}))[i, j] \quad (16)$$

$\therefore (\mathbf{x} \otimes \mathbf{y}) \circ (\mathbf{x} \otimes \mathbf{z}) = \mathbf{x} \otimes (\mathbf{y} \circ \mathbf{z}) \quad \square$

This lemma result means that we can safely reduce the number of Kronecker multiplications required during bucket alignment, *i.e.*, $(\mathbf{A}_1 \otimes \mathbf{B}_1) \vee (\mathbf{A}_1 \otimes \mathbf{B}_2) = \mathbf{A}_1 \otimes (\mathbf{B}_1 \vee \mathbf{B}_2)$. Furthermore, the logical $\vee$ is applied on smaller bitmap columns prior to Kronecker expansion, substantially reducing computation costs.

V. PERFORMANCE EVALUATION

We conducted several experiments to evaluate our structures and algorithms. In this section we initially describe the benchmark for testing and then we give detailed explanations of our experimental results.

A. Experimental Setup

All tests were performed on an `m3.large` instance in Jetstream2 IU [23]. The large instance is equipped with 16 vCPUs, 60 GB RAM. All system implementations were written in Java, and run using OpenJDK 16 on an Ubuntu 22.04 headless server image. Our experiments use the TPC-H Benchmark [24] to generate the database and query workload. TPC-H models a wholesale parts supplier: customers place orders, orders may contain multiple line items, suppliers provide parts. We focus our experiments on two tables: *partsupp*, which suppliers can supply which parts and at what cost, and *lineitem*, which records lines in a customer order. The *lineitem* table contains 600,000 rows and a foreign key to *partsupp*, which has 20,000 rows. A join must be taken between the two tables to reference each line item’s supplier.

We then created the bitmap indices over the necessary attributes to perform joins and range queries. For the *partsupp* table, we indexed its primary key, resulting in a $20,000 \times$

20,000 bitmap. For the *lineitem* table, two bitmaps were created. The first bitmap is on the foreign key *lineitem.partkey*, in which 20,000 distinct *partkey* values are each represented with a bitvector (column). This results in a 600,000 row $\times$ 20,000 column bitmap. Another bitmap is constructed on the *lineitem.shipdate* attribute, in which there are roughly 2,026 distinct dates represented, *i.e.*, a 600,000 row $\times$ 2,026 column bitmap. For comparison, we also constructed B+Trees over these same attributes.

Because our system is specifically designed to optimize a series of range queries after a join is taken, we created a query workload of numerous join + range queries of the form:

```
select * from partsupp join lineitem
on partsupp.partkey = lineitem.partkey
where startdate < lineitem.shipdate < enddate;
```

where legal values of *startdate* and *enddate* date are randomly selected for each query.

B. Experimental Results

In this subsection, we present an evaluation of our system design. The first set of experiments compare Early Expansion against Deferred Expansion. We performed a join between the *lineitem* and *partsupp* tables and executed increasing numbers of range queries over the joined result. The results of this comparison are shown in Figure 1. As expected, Early Expansion carries a high initial cost due to the construction of the bitmap-join result. After incurring this cost, the join result *J* is cached, and the subsequent range queries are applied directly over *J*, incurring substantially lower costs per query thereafter.

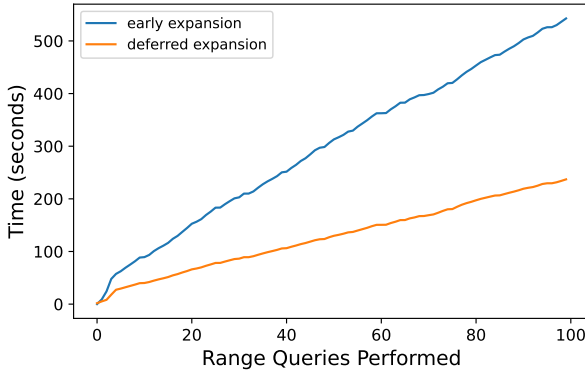


Figure 1. Early Expansion vs. Deferred Expansion

Deferred Expansion, in contrast, delays the bitmap join. The range filter is applied first, which has trivial cost compared to the join, then depending on the tuples indicated in this range-filter result, we expand those into the joined result. This produces low initial costs, but linear growth per query. But as can be seen in the figure, the rate of growth is slow enough that it would require several hundred subsequent range queries for Early Expansion to begin outstripping the performance of Deferred Expansion. This scenario would be impractical in transactional database environments. We therefore believe

that Early Expansion is nonviable and focus the remaining experiments on Deferred Expansion.

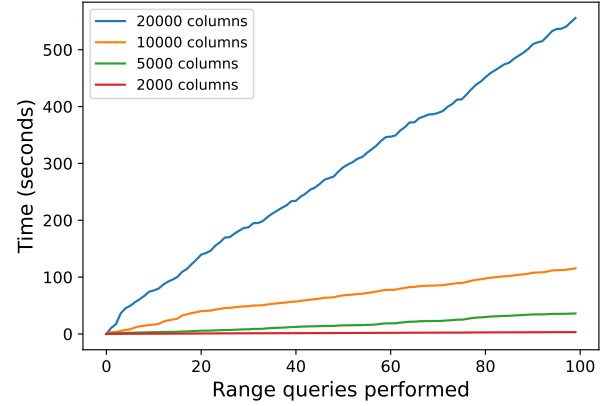


Figure 2. Effects of Coarsening Bit-Vectors on Execution Time

The number of bit-vectors used to represent the join-key values was a significant factor in execution time. As noted earlier, we constructed a 20,000-column bitmap over the *lineitem.partkey* attribute, with each column corresponding to a distinct value in the *partkey* domain. To isolate the effect of bitmap dimensionality on execution time, we coarsened the join-key bitmap by allowing each bitmap column to represent multiple key values. For example, using buckets of size two reduces the number of bitmap columns from 20,000 down to 10,000. These results are intended to measure the effect of bitmap column count on execution cost rather than the end-to-end performance of an exact query-processing strategy.

The results of coarsening each bit-vector to represent buckets of 2, 4, 8 *partkey* values meant that we have reduced the sizes of the *partkey* bitmap down to 10,000, 5,000, and 2,000 columns, respectively. Figure 2 shows the results of this experiment for Deferred Expansion. Reducing the bitmap's column count produced a disproportionately large performance gain, indicating nonlinear scaling with respect to the number of columns. There is an adverse effect of this dimensionality reduction, however. The false-positive rate would expectedly increase for a reduced column count, which may be tolerable depending on the distribution of the underlying data. This would require more candidate tuples to be fetched from disk and checked, but we did not conduct a formal evaluation into the false positive rate.

Next, we compare Deferred Expansion with the well-known indexing scheme, B+Tree. The purpose of this experiment is to demonstrate the competitiveness of Deferred Expansion. For this set of experiments, we again varied the granularity of the bitmap used to index the *lineitem.partkey* join key. Accordingly, we also reduced dimensionality of the B+Tree in lockstep with the dimensionality reduction of the bitmapped join key. We did this by calculating each key's modulo-target-value prior to inserting it into the key. This method produces the same false-positive rate as the bitmap at each step for a fair comparison.

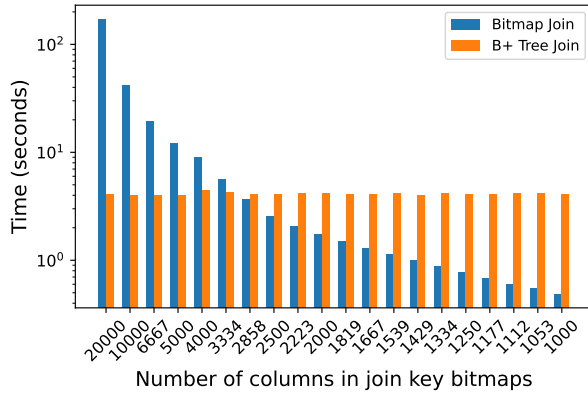


Figure 3. Deferred Expansion vs. B+Tree

We plotted the total execution times to run all 100 join+range queries in our workload, comparing their runtime over decreasing column counts. The results are shown in Figure 3, plotted on the log-scale for readability. As expected, the B+Tree’s performance is quite consistent since the dimensionality reduction can remove up to a single level in the tree. At full dimensionality, Deferred Expansion is over an order of magnitude slower than the B+Tree, but becomes competitive and eventually outperforming the B+Tree as column counts are reduced.

We emphasize that, although Deferred Expansion underperforms B+Trees at high precision, the results are nonetheless promising. The goal of this work is not to replace B+Trees, but rather to provide a viable approach for executing online join+range queries over an already-bitmapped database. That our approach remains competitive with B+Trees is therefore an encouraging result.

The final test involves the runtime comparison of how bucket alignment affects the runtime of the join (including expansion). We bucketed the results so that there were two values per column. In the *Column Aligned Join* test, we ensured that that the buckets in the two tables were identically aligned. In the *Column Misaligned Join* test, the columns were offset by one value. So, if a column in table A represented the values 11 – 13, then the columns in B overlapping with that range would be one from 10 – 12, and 12 – 14.

In all the sizes, we see approximately a 20% – 25% cost when the columns are misaligned. This is not ideal, though performing joins on misaligned buckets is generally not ideal due to the high false positive rate of the join itself.

VI. RELATED WORK

Much of the related work in the area of supporting joins over bitmaps are for OLAP workloads. In this setting, indexed data is in general assumed to be read-mostly and relatively static, supplanting the need to compute expensive joins on-the-fly as would be required in transactional systems. In this light, prior approaches have focused instead on precomputed join structures or offline techniques.

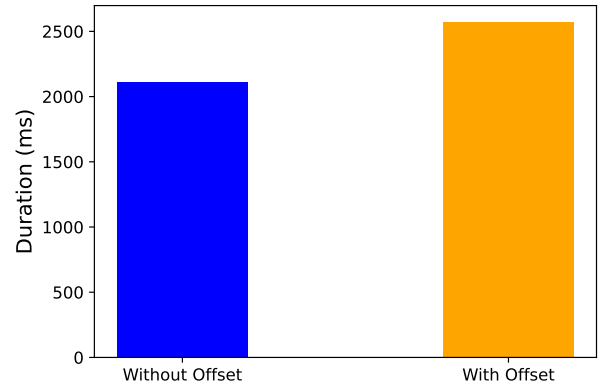


Figure 4. Joins on Aligned vs. Misaligned Buckets

O’Neil and Graefe first proposed precomputed bitmap-join indices as an approach as a method to accelerate multi-table, star join, operations in database systems, particularly for decision-support and data warehousing (OLAP) workloads [25]. The authors propose creating compressed bitmap structures that encode join relationships across tables, allowing many join queries to be answered through fast bitwise operations rather than runtime join execution. Aouiche, et al., proposed a method for automatically choosing which bitmap join indices to materialize in a data warehouse based on workload characteristics and storage constraints [16]. The authors model the tradeoff between query performance gains and index maintenance/storage overhead, then identify the most beneficial bitmap-join indices for a given workload. In contrast to these approaches, our work does not use materialized bitmap-join indices or offline index-selection strategies. Instead, we focus on dynamically executing joins through bitmap algebra, while preserving sufficient bitmap structure to support downstream range queries.

Brito, et al. proposed dBIJ (distributed Bitmap Join Index), an extended the bitmap-join index to further consider star-schema queries in distributed file systems. They partitioned the precomputed join-index into fragments and distributed them across cloud nodes to support efficient random-access retrieval while reducing communication and full-scan costs. In contrast, our work avoids precomputed join-index construction entirely, instead dynamically computing joins through bitmap algebra and Kronecker-product composition. Our work does not consider the distributed environment.

An inspiration to our work, Madduri and Wu propose algorithms that use compressed bitmap indices to accelerate “range join queries,” e.g., joins that of the form $R \bowtie_{R.a < S.a} S$ [20]. Experimental results show substantial speedup over traditional sort-merge join algorithms. In contrast, our work focuses on executing equality (natural) joins over bitmap indices while integrating support for subsequent range-query processing on the joined result itself, e.g., queries of the form $\sigma_{R.b < S.b}(R \bowtie_{R.a = S.a} S)$. Rather than accelerating inequality-based range joins, we optimize the structure and reuse of join products for downstream filtering.

VII. CONCLUSION AND FUTURE WORK

We presented a number of optimizations to better the ability for join queries performed with bitmaps to be utilized. We showed how the join itself can be used to calculate range queries as if it were a bitmap, and how it can be even further optimized if that range query happens over the correct key. We showed how the intentional delaying of portions of the join algorithm can lead to speedups caused by range queries lowering either the number of columns or the number of dirty bits in a column. Finally, we showed optimizations to help speed up joins over bucketed bitmaps when the buckets do not perfectly align.

There are, however, more considerations that may have promises. At multiple points, we mentioned how performing an operation would either benefit from caching, benefit the cache, or both. A proper implementation and tests with caches of intermediate Kronecker products, for instance, would provide significant speedup in multiple areas. Another speedup opportunity is to parallelize the calculation of Kronecker products, which should be embarrassingly parallel. Additionally, the divide and conquer method of performing large numbers of $\vecs$ (Algorithm 2) is also highly parallelizable.

Beyond traditional OLAP workloads, vector-based filtering and structured retrieval are increasingly used in LLM retrieval systems, including RAG pipelines that combine vector similarity search with metadata filtering and relational constraints. We plan to seek ways to integrate bitmap technology into the LLM/RAG workflow.

ACKNOWLEDGMENTS

This research was funded in part by JetStream2 Indiana University through allocation CIS210089 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. This work was also funded in part by a summer research award to Alex Kirner and David Chiu by the University of Puget Sound.

AI acknowledgment: The OpenAI model (gpt-5.x) was used for fast-tracking $\LaTeX$ formatting, including the creation of the tables used in Section II and math-style suggestions. We also credit the model for suggesting the names of our two algorithms, Early Expansion and Deferred Expansion.

REFERENCES

- [1] H. K. T. Wong, H. fen Liu, F. Olken, D. Rotem, and L. Wong, "Bit transposed files," in *Proceedings of VLDB*, 1985, pp. 448–457.
- [2] P. E. O'Neil, "Model 204 architecture and performance," in *International Workshop on High Performance Transaction Systems*, 1989, pp. 40–59.
- [3] K. Stockinger and K. Wu, "Bitmap indices for data warehouses," in *Data Warehouses and OLAP*. Press, 2006.
- [4] K. Wu, E. J. Otoo, and A. Shoshani, "Compressing bitmap indexes for faster search operations," in *International Conference on Scientific and Statistical Database Management*. IEEE, 2002, pp. 99–108.
- [5] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg, "Notes on design and implementation of compressed bit vectors," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL/PUB-3161, 2001.
- [6] F. Corrales, D. Chiu, and J. Sawin, "Variable length compression for bitmap indices," in *Database and Expert Systems Applications*, 2011, pp. 381–395.
- [7] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin, "A tunable compression framework for bitmap indices," in *International Conference on Data Engineering (ICDE'14)*, 2014, pp. 484–495.
- [8] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai, "Roaring bitmaps: Implementation of an optimized software library," *Software: Practice and Experience*, vol. 48, no. 4, pp. 867–895, jan 2018.
- [9] Pilosa, "White paper: Pilosa: A technical overview," Pilosa, Tech. Rep., December 2017. [Online]. Available: <https://www.pilosa.com/pdf/PILOSA%20-%20Technical%20White%20Paper.pdf>
- [10] S. Burdick, J. Risner, D. Chiu, and J. Sawin, "Fault-tolerant query execution over distributed bitmap indices," in *International Conference on Big Data Computing Applications and Technologies*, 2018, pp. 21–30.
- [11] M. Nelson, Z. Sorenson, J. M. Myre, J. Sawin, and D. Chiu, "Parallel acceleration of CPU and GPU range queries over large data sets," *J. Cloud Comput.*, vol. 9, p. 44, 2020. [Online]. Available: <https://doi.org/10.1186/s13677-020-00191-w>
- [12] B. Tran, B. Schaffner, J. M. Myre, J. Sawin, and D. Chiu, "Exploring means to enhance the efficiency of GPU bitmap index query processing," *Data Sci. Eng.*, vol. 6, no. 2, pp. 209–228, 2021. [Online]. Available: <https://doi.org/10.1007/s41019-020-00148-8>
- [13] S. McClain, M. Mutschler-Aldine, C. Monaghan, D. Chiu, J. Sawin, and P. Jarvis, "Caching support for range query processing on bitmap indices," in *International Conference on Scientific and Statistical Database Management (SSDBM'21)*, 2021, pp. 49–60.
- [14] J. Kaeppl, J. Sawin, and D. Chiu, "Workload-aware cache management of bitmap indices," in *Proceedings of the IEEE/ACM 10th International Conference on Big Data Computing, Applications and Technologies, BDCAT 2023, Taormina (Messina), Italy, December 4-7, 2023*. ACM, 2023, pp. 05:1–05:10.
- [15] P. O'Neil and D. Quass, "Improved query performance with variant indexes," *SIGMOD Rec.*, vol. 26, no. 2, p. 38–49, Jun. 1997. [Online]. Available: <https://doi.org/10.1145/253262.253268>
- [16] K. Aouiche, J. Darmont, O. Boussaïd, and F. Bentayeb, "Automatic selection of bitmap join indexes in data warehouses," in *Data Warehousing and Knowledge Discovery, 7th International Conference, DaWaK 2005*, ser. Lecture Notes in Computer Science, vol. 3589. Copenhagen, Denmark: Springer, 2005, pp. 64–73.
- [17] L. Bellatreche, R. Missaoui, H. Necir, and H. Drias, "A data mining approach for selecting bitmap join indices," *Journal of Computing Science and Engineering*, vol. 1, no. 2, pp. 177–194, 2007.
- [18] J. J. Brito, T. Mosqueiro, R. R. Ciferri, and C. D. A. Ciferri, "Random access with a distributed bitmap join index for star joins," *Heliyon*, vol. 6, no. 2, p. e03342, Feb. 2020.
- [19] C.-Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 355–366. [Online]. Available: <https://doi.org/10.1145/276304.276336>
- [20] K. Madduri and K. Wu, "Efficient joins with compressed bitmap indices," in *ACM Conference on Information and Knowledge Management*, 2009, p. 1017–1026.
- [21] C. F. Loan, "The ubiquitous kronecker product," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1, pp. 85–100, 2000, numerical Analysis 2000. Vol. III: Linear Algebra. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0377042700003939>
- [22] F. Deligè and T. B. Pedersen, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps," in *International Conference on Extending Database Technology*, ser. EDBT '10, 2010, pp. 228–239.
- [23] D. Y. Hancock, J. Fischer, J. M. Lowe, W. Snapp-Childs, M. Pierce, S. Marru, J. E. Coulter, M. Vaughn, B. Beck, N. Merchant, E. Skidmore, and G. Jacobs, "Jetstream2: Accelerating cloud computing via jetstream," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org>
- [24] Transaction Processing Performance Council, "Tpc-h decision support benchmark," 2021. [Online]. Available: <http://www.tpc.org/tpch>
- [25] P. O'Neil and G. Graefe, "Multi-table joins through bitmapped join indices," *SIGMOD Rec.*, vol. 24, no. 3, p. 8–11, Sep. 1995.