Efficient Joins with Compressed Bitmap Indexes

Kamesh Madduri
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
KMadduri@lbl.gov

Kesheng Wu
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
KWu@lbl.gov

Abstract

We present a new class of adaptive algorithms that use compressed bitmap indexes to speed up evaluation of the range join query in relational databases. We determine the best strategy to process a join query based on a fast sub-linear time computation of the join selectivity (the ratio of the number of tuples in the result to the total number of possible tuples). In addition, we use compressed bitmaps to represent the join output compactly: the space requirement for storing the tuples representing the join of two relations is asymptotically bounded by $min(h, n \cdot c_b)$, where h is the number of tuple pairs in the result relation, n is the number of tuples in the smaller of the two relations, and c_b is the cardinality of the larger column being joined. We present a theoretical analysis of our algorithms, as well as experimental results on large-scale synthetic and real data sets. Our implementations are efficient, and consistently outperform well-known approaches for a range of join selectivity factors. For instance, our count-only algorithm is up to *three orders of magnitude* faster than the sort-merge approach, and our best bitmap index-based algorithm is 1.2×–80× faster than the sort-merge algorithm, for various query instances. We achieve these speedups by exploiting several inherent performance advantages of compressed bitmap indexes for join processing: an implicit partitioning of the attributes, space-efficiency, and tolerance of high-cardinality relations.

1 Introduction

The join is a fundamental and versatile relational database operation. Informally, it is used to combine tuples from two (or more) different relations (or tables) based on some common information [11]. There are several different kinds of joins, and a vast collection of algorithmic strategies to solve them [11, 13]. In general, the join is considered one of the most difficult relational operations to implement efficiently.

A join is performed on two input relations R and S, producing a result relation Q. The *join condition* specifies the desired relationship to combine the tuples from R and S. We denote the join attributes (or join columns) in R by $r(a)$, and the attributes in S by $s(b)$. The join relation Q is then given by $R \bowtie_{r(a)\theta s(b)} S$, where $r(a)\theta s(b)$ defines the join condition. Let n and m denote the number of tuples in R and S, respectively. The *join selectivity factor* is defined as the ratio of the number of tuples in the resulting join relation (h) to the total number of possible tuple pairs due to the cross-product of the two tables (mn).

In this paper, we present new algorithms for a specific type of join called the *band* [5] or *range join*. For this join variant, the join condition is of the form $r(a) - \delta_1 \leq s(b) \leq r(a) - \delta_2$, for some constants δ_1 and δ_2. Note that if δ_1 and δ_2 are both zero, this reduces to the commonly-used *equijoin*.

There are several known approaches for optimizing band joins, and in general the join operation. If the number of join attributes is small and known before-hand, one could construct a *join index*, a special relation tailored for answering these queries. *Bitmap join indexes* are a specialization of this data structure that use bitmaps (bit vectors), and are extensively studied [2, 9, 12, 14]. However, by definition, they are only useful in cases involving specific columns for which the index has been built for; i.e., joins on arbitrary columns cannot benefit from this specialized index. In addition, join indexes can be significantly larger than regular bitmap indexes. For instance, in a recent study on a geographical application, Siqueira et al. found that a specific join index for their data was 1000 times larger than a regular bitmap index [17]. For these reasons, it is worthwhile to investigate the possibility of using regular bitmap indexes for computing band joins and other variants.

The other class of algorithms for evaluating joins are robust and general-purpose, requiring no specialized data structures. The *nested-loop*, *sort merge*, *hash join*, and their variants are some well-known examples [11]. The sort-merge join, as its name suggests, sorts the join attributes to reduce the number of tuple comparisons. It then merges the tuples according to the join condition, and if the join selectivity is low ($\mathcal{O}(n)$), this approach is significantly faster than nested-loop, and is more efficient in I/O accesses. The execution is typically dominated by the sorting step, which is $\mathcal{O}(n \log n)$ using a generic comparison-based sorting algorithm. Hash-based algorithms have a linear time complexity, but are known to be difficult to solve non-equijoins (for instance, band joins). Partitioned band join algorithms [5] try to minimize the disk I/O time in materializing the join result by partitioning the relations using a sampling strategy, and then computing band joins on the smaller partitions. This divide and conquer strategy outperforms the standard sort-merge in several instances, but still relies on a sort as an inner computational kernel. In comparison, our approaches operate entirely with bitmap indexes and avoid the sorting kernel.

1.1 Our Contributions

We present a class of new algorithms for computing range joins using bitmaps indexes. We directly compute cross-products using bitmaps for answering a join query, whereas the bitmap join index can be thought of as precomputing such cross-products. Because we keep the involved bitmaps compressed during every step of the query evaluation process, the storage requirement is low. We additionally use compressed bitmaps to represent the results of joins, which keeps the output also compact.

Our motivating applications for this work arise from the domain of scientific data analysis with predominantly numerical data sets. Here, there is a need to perform multi-dimensional range join operations on arbitrary attributes of two or more relations. For instance, in astronomical data analysis, we have large catalogs on which queries such as cross matching [16], cone searches, and spatio-temporal matching [6] are posed, with range joins as one of the key underlying computational kernels. Our algorithms can use compressed bitmap indexes already built for range queries, and thus do not require additional storage or preprocessing. To demonstrate the applicability of our approaches to large-scale astronomy data sets, we present range join query performance results on the Sloan Digital Sky Survey Release-1 in this paper.

This paper is organized as follows. We introduce compressed bitmap indexing in Section 2, define the problem of computing the cross-product of two bit vectors, and describe a baseline approach to compute equijoins using compressed bitmap indexes. We then present the range join problem formulation in Section 3 and outline an algorithm based on cross products of bit vectors to solve this problem. In Section 3.2, we specialize this approach and present an algorithm to compute the join selectivity factor quickly, by inspecting compressed bit vectors corresponding to the two attributes being joined. Based on this value, we present two new strategies for actual expansion of the resulting join table in Section 3.3. Specifically, we consider the cases where $h = \mathcal{O}(m)$ and $h = \mathcal{O}(mn)$. In both the cases, we show how the resulting output can be represented compactly. We then extend the ideas to multiway range joins in Section 4, and also detail how these join queries can be answered in conjunction with the search capability of compressed bitmap indexes. Finally, we present a detailed experimental study in Section 5 comparing our new approaches with the traditional sort-merge algorithm. With a combination of theoretical and empirical results, we clearly identify the cases when our approach would be faster than the sort-merge algorithm, and quantify the speedups achieved. We demonstrate that our best approach is 1.2×–80× faster than the sort-merge implementation.

2 Compressed Bitmap Indexes

A bitmap index is an indexing scheme that stores data as bit sequences, and answers queries using bitwise logical operations. The bit sequences are referred to as bitmaps or bit vectors. Typically, a bitmap index is generated for one attribute of a relation/table, and is comprised of a bit vector for each distinct value of the attribute. The distinct values are referred to as

the *keys*. Since the bitmap size will then grow linearly with the number of distinct values, which is also known as the *attribute cardinality*, it is efficient only for columns with low cardinalities. The same restriction also applies to the bitmap join index [12, 14]. There are a number of commercial implementations of bitmap indexes, and users are often cautioned to use them only on low cardinality columns.

WAH compression: There are however various approaches to extend the effectiveness of bitmap indexes to columns with higher cardinalities. The three common ones are binning [8], encoding [3, 4, 14], and compression [1, 7]. In this work, we use compressed bitmaps with a compression method called Word-Aligned Hybrid (WAH) [19] for range join evaluation. In prior research, Wu et al. demonstrated that WAH compression is extremely efficient for compressing bitmap indexes and optimal for range queries. Our new algorithms are implemented in a software system built for scientific data, where the majority of the records are read-only. In this software package FastBit[1], all data tables are vertically partitioned to support efficient ad-hoc queries. In many applications that use FastBit, a compressed bitmap index is already built for each column that is queried. Thus, our join algorithms require no additional preprocessing for supporting ad-hoc join queries. Our new join algorithms extensively use logical operations on WAH compressed bitmaps, which are shown to be very efficient in FastBit implementations [19].

WAH compression is based on the idea of run-length encoding. Two types of words are used to store bits: *literal* words and *fill* words. The most significant bit of the word is used to distinguish between a literal and fill word. Let w denote the number of bits in a computer word. Then, the lower $w-1$ bits of a literal word contain bit values, 0's and 1's, in an uncompressed form. The fill word, on the other hand, is used to compactly store 0'a and 1's that are multiples of $w-1$ bits. The second most significant bit in a fill word indicates the type of fill, and the rest of the $w-2$ bits are used for storing the length of the fill, or the number of integer multiples of $w-1$. If the total number of bits is not a multiple of $w-1$, then the last literal word created has just a few valid bits. This is referred to as an *active word*.

We assume that equality encoding is used for the bitmap index. Bitmaps constructed with range [3] and interval encoding are not as compressible as equality-encoded bitmaps, and the index size may become prohibitive when doing arbitrary range joins. Binary encoding [14, 18] would be significantly more expensive than equality encoding for narrow-range queries, which would be our typical query instances on scientific data sets.

2.1 The cross-product of two bit vectors

We first present a baseline algorithm for an important subroutine, a cross-product of two WAH-compressed bit vectors. We will discuss equijoin algorithms based on this cross-product subroutine (denoted as BCP, or Baseline Cross Product) in the following sections.

Consider a join condition that just operates on one join column per relation, i.e., $|r(a)| = |s(b)| = 1$. Let a and b denote these attributes in R and S, respectively. Let c_a and c_b denote

[1]More information about FastBit can be found at http://sdm.lbl.gov/fastbit.

Table 1: List of symbols.

Symbol(s)	Description
R, S	Relations/tables that are being joined.
$r(a)$, $s(b)$	Join attributes/columns in R, S respectively.
n	Number of tuples in R, and size of each bit vector in bitmap index of a.
m	Number of tuples in S, and size of each bit vector in bitmap index of b.
h	Number of tuples in the join result.
c_a, c_b	Number of bit vectors in bitmap index (or attribute cardinalities) of a, b.
a_i	Bit vector corresponding to value i in bitmap index of a.
C_i	Bit matrix corresponding to cross product of a_i and b_i.
l_a	Number of bits set to 1 in bit vector a.
l_C	Number of bits set to 1 in the bit matrix.
n_{Ceq}	Number of bit vector OR operations in CP-EquiJoin.
n_C	Number of unique b bit vectors touched in Algorithm CP-FastRangeJoin.
l_{nzcol}	Number of non-empty columns in compact bit matrix representation (Algorithm CP-FastRangeJoin).
w	Number of bits in a computer word.
m_R, m_S	Selection conditions on R, S respectively, in addition to the join condition.

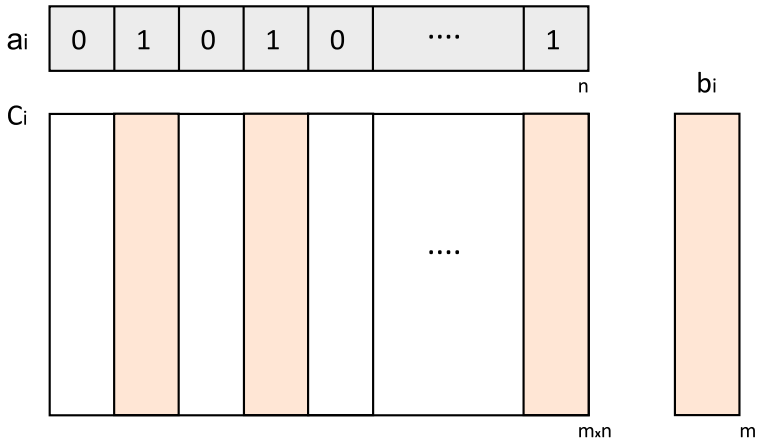

Figure 1: A simple representation of the cross-product of two uncompressed bit vectors.

the attribute cardinalities of a and b respectively. See Table 1 for a listing of other symbols.

Let a_i and b_i denote bit vectors corresponding to the value i in attributes a and b respectively. The *cross-product* of these two bit vectors is defined as a bit matrix C_i, with $C_i(p,q) = a_i(p)\&b_i(q)$, where & denotes the bitwise logical AND operator (illustrated in Figure 1). Clearly, an uncompressed representation of the bit matrix would quickly become intractable to store on current workstations, due to the quadratic space complexity. The performance of any high-level routine that may operate on this cross-product will also be adversely affected, due to the large memory footprint of the result. However, if we store both a_i and b_i in a compressed manner, then computing the cross-product would still be feasible.

Algorithm 1 describes a simple approach to construct the cross-product. At a high level, the algorithm for computing the cross product is to simply append the bit vector b to the appropriate location in the result bit matrix. For internal representation of the result matrix, we linearize it in column-major ordering. This allows us to treat the bit matrix as a long bit vector and apply the WAH-compression technique.

Algorithm 1: BCP: A simple approach for computing the cross-product of two compressed bit vectors.

Input: Two compressed bit vectors a and b.
Output: The cross-product of a and b, stored in C.

```
1 C ← φ; (initialize an empty compressed bit vector)
2 zv ← b & 0; (initialize a compressed bit vector representing m zero bits.)
3 for i = 1 to n do
4     if a[i] = 1 then
5         C ← Append(b);
6     else
7         C ← Append(zv);
```

Analysis: Let l_a and l_b denote the number of bits that are set to 1 in each vector. Since

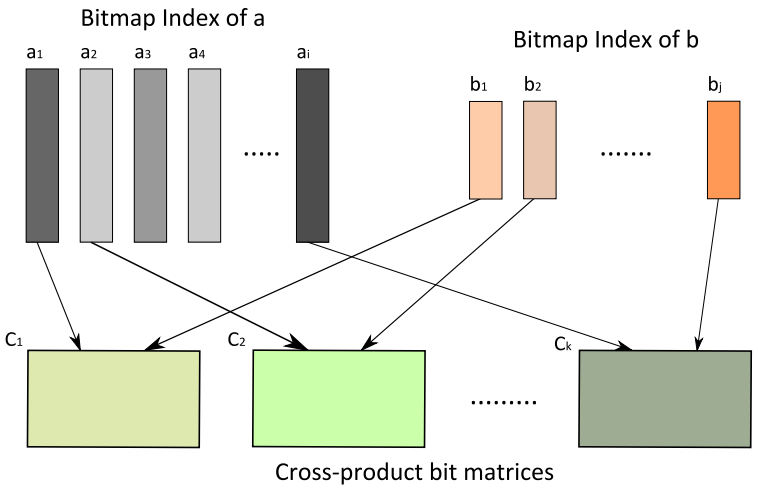

Figure 2: Computing an equijoin through a series of logical OR operations on bit vector cross-products.

Algorithm 1 simply replicates b l_a times, we can easily show that the size of the WAH-compressed cross-product is $\mathcal{O}(l_a l_b)$. The running time of Algorithm 1 is dominated by step 5, which is executed l_a times. Thus we have the following theorem.

Theorem 1 *The space and time requirements to compute a cross product C of two bit vectors a and b are both $\mathcal{O}(l_C)$, where l_C is the number of bits that are 1.*

2.2 Equijoin from cross-product bit matrices

We generate explicit cross-products of bit vectors as described in Algorithm 1, since they provide a simple interface to implement joins (and other queries) using compressed bit vectors. Consider the illustration in Figure 2. The set bits in the cross-product C_i of bit vectors a_i and b_i correspond to tuples in the join result for which the values match. It is evident that the resulting relation from an equijoin can be obtained by performing a series of logical operations of bit vectors corresponding to values that match in both the join attributes. Using compressed bit vectors to represent the cross-products helps us store the result in a space-efficient manner, and also lets us leverage previous work on efficient bitwise logical operations.

Thus, we are obtaining the result of a join operation by extracting a subset of the *Cartesian product* of the two join attributes $r(a)$ and $s(b)$. The output bit matrix gives the pairs of tuple identifiers from the two columns which satisfy the join condition. The resulting join table can then be materialized by accessing the data, which may reside in main memory or secondary storage. Algorithm 2 formally lays out this bit vector cross-product based approach.

Analysis: We assume that all the distinct values in these columns (the *index keys*) are stored in a sorted array and can be accessed in constant time. This information is typically

Algorithm 2: BCP-EqJoin: An equijoin algorithm using explicit cross-products of bit vectors.

Input: Bitmap indexes of join attributes a and b from relations R and S respectively.
Output: Pairs of tuple identifiers from R and S that satisfy the join condition. The result tuples are represented as a compressed bit matrix C, denoting a subset of the the Cartesian product.

1 $C \leftarrow \phi$; (initialize an empty compressed bit vector for storing the join tuple pair identifiers)
2 $i \leftarrow 1; j \leftarrow 1;$
3 **while** $(i \leq c_a)$ *&* $(j \leq c_b)$ **do**
4 $\quad k_a \leftarrow Fetch\_Next\_Key(a, i);$
5 $\quad k_b \leftarrow Fetch\_Next\_Key(b, j);$
6 $\quad$ **if** $k_a < k_b$ **then**
7 $\quad\quad i \leftarrow i + 1;$
8 $\quad$ **if** $k_b < k_a$ **then**
9 $\quad\quad j \leftarrow j + 1;$
10 $\quad$ **if** $k_a = k_b$ **then**
11 $\quad\quad C \leftarrow C \mid a_i \otimes b_j;$

available to us as an auxiliary data structure from the bitmap index corresponding to the join attribute. We inspect keys in both a and b by stepping though the arrays in sorted order (we assume *Fetch_Next_Key* in Algorithm 2 provides us with these values in constant time). In step 11, we perform a cross-product on bit vectors corresponding to the matches, followed by a logical OR of this matrix with a running copy of the result C. From Theorem 1, we know that the size and running time of each intermediate cross-product computation is bounded by $\mathcal{O}(l_{Ci})$, where l_{Ci} is the number of set bits in an intermediate result bit matrix. The logical OR operation can be efficiently computed on the compressed bit vectors in optimal $\mathcal{O}(l_{Ci} + l_{Cr})$ time [19], where l_{Cr} is the number of set bits in the running OR result. Let n_{Ceq} denote the total number of OR operations that are performed. By definition, we know that $\Sigma_{i=1}^{n_{Ceq}} l_{Ci} = h$. In the worst case, $n_{Ceq} = \min(c_a, c_b)$. Also, note that the l_{Cr} term grows during the execution of the algorithm. Since this cost is incurred in every OR computation, it would be wise to defer the evaluation of cross-products leading to a significant number of set bits to latter stages of the algorithm. This leads us to an upper bound on the execution time, as well as an optimal ordering for computing the cross-products.

Theorem 2 *The execution time of BCP-EqJoin is given by* $\mathcal{O}(n_{Ceq} \cdot h)$*, where* $n_{Ceq} \leq min(c_a, c_b)$ *and* h *is the number of set bits in the result relation. The space requirements of the algorithm are* $\mathcal{O}(h)$*.*

We assume a worst-case ordering of the OR's to compute this bound. In practice, we can scan through the lists in $\mathcal{O}(c_a + c_b)$ time to determine the number of matches and the intermediate result sizes. We could then order the OR computations according to the

Table 2: Possible compressed result words created from a bit-level combination of uncompressed a_1 and a_2 bit vectors when evaluating $a_1 \otimes b_1 | a_2 \otimes b_2$.

Bit value in a_1	Bit value in a_2	Result word(s)
0	0	0-fill word, value u_b
0	1	b_2
1	0	b_1
1	1	$b_1 \vert b_2$

estimated result matrix sizes, starting with the smallest one first. Note that the average-case running time here is dependent on characteristics of the join attributes.

2.3 Improvements to the cross-product based equijoin algorithm

Algorithm 2 is useful as a baseline approach for comparison to the new join algorithms we present in this paper. From the worst case theoretical bounds, we expect this cross-product based algorithm to either outperform or be competitive with the traditional approaches such as sort-merge and hash-join in the following scenarios:

- The number of OR operations performed is a constant value, or much smaller than $\log n$ (n is the number of tuples in the input relation).
- Low join selectivity factors.
- A skewed distribution of cross-product output sizes, i.e., the presence of a very few large cross-product computations that may be deferred to the end.

The multiplicative factor of n_{Ceq}, the number of OR operations that have to be performed, is the primary performance concern for this equijoin approach. We now investigate techniques to speed up this equijoin algorithm by first exploring approaches to reduce the computational complexity.

Step 11 of Algorithm 2 involves a cross-product computation followed by a logical OR operation on potentially large vectors. To answer most join queries, we need to perform bitwise ORs of several different bit matrices. Consider an OR operation $c_1 | c_2$ where $c_1 = a_1 \otimes b_1$ and $c_2 = a_2 \otimes b_2$. Table 2 lists all possible outcomes of the output based on a bitwise comparison of the uncompressed bits in a_1 and a_2.

We see from the table that it is possible to compute the result of the OR operation by just comparing bits of the uncompressed a_1 and a_2 vectors. Further, the result of the OR operation need not be explicitly stored, as the output words are possibly one of the following four combinations: a 0-fill word, b_1, b_2, or $b_1 | b_2$. We can compute the join result size by just determining the number of b_1, b_2, and $b_1 | b_2$ occurrences in the result, and the number of 1's in each of these vectors.

Table 3: Possible cases to consider in the compressed-word implicit cross-product and OR computation $a_1 \otimes b_1|a_2 \otimes b_2$.

Word type in a_1	Word type in a_2	Result words	# of result words
0-fill, value p	0-fill, value q	0-fill, value $min(p,q) \cdot u_b$.	1
0-fill, value p	1-fill, value q	b_2 replicated $min(p,q) \cdot (w-1)$ times.	$(w-1) \cdot min(p,q) \cdot u_b$
0-fill, value p	literal word (LW)	0-fill words (value u_b) and b_2 replicates (equal to # of 1's in LW).	runtime
1-fill, value p	0-fill, value q	b_1 replicated $min(p,q) \cdot (w-1)$ times.	$(w-1) \cdot min(p,q) \cdot u_b$
1-fill, value p	1-fill, value q	$b_1\|b_2$ replicated $min(p,q) \cdot (w-1)$ times.	$(w-1) \cdot min(p,q) \cdot u_b$
1-fill, value p	LW	b_1 or $b_1\|b_2$ replicates (equal to # of 1's in LW).	runtime
LW	0-fill, value q	0-fill words (value w_b) and b_1 replicates (equal to # of 1's in LW).	runtime
LW	1-fill, value q	b_2 or $b_1\|b_2$ replicates (equal to # of 1's in LW).	runtime
LW	LW	0-fill words, b_1, b_2, and $b_1\|b_2$ replicates.	runtime

Operating on individual bits is expensive. Consider working with WAH-compressed a_1 and a_2 vectors, and performing the OR operation without expanding the cross-product. Table 3 lists all possible combinations of word-level OR operations, assuming WAH compression. To simplify the analysis, we assume that there are no active words in the a or b bit vectors. The number of new words created in the result, and the composition of the result for each combination, is listed in the table. As noted earlier, any contiguous section of compressed words in the result array can only be one of four different possibilities: a 0-fill word, b_1, b_2, or $b_1|b_2$. This leads to a further optimization. As illustrated in Figure 3, if we just store the type of the pattern (one of these four possibilities) and a running sum of the position offset associated with its location, we can reconstruct the result array.

Analysis: The implicit cross-product approach we describe above is a replacement to line 11 of the BCP-EqJoin algorithm. We first evaluate pairwise combinations of the n_{Ceq} result vectors in this manner, and then perform OR operations on the resulting compressed bit vectors. In terms of storage, we need n pointers to compressed bit vectors of m bits each. The n pointers require $\mathcal{O}(n)$ words, and the size of each m-bit vector is proportional to the number of set bits in it.

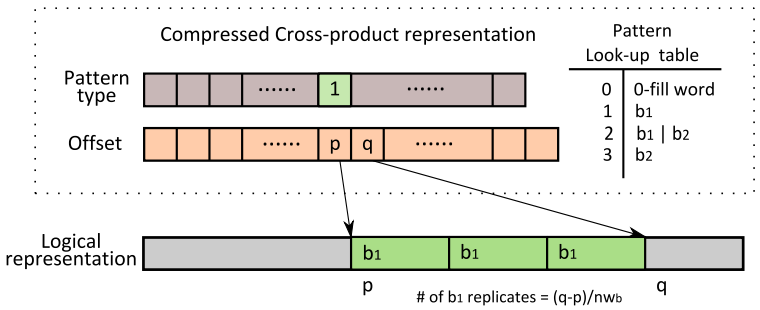

Figure 3: A scheme to further compress the cross-product bit matrix.

The percentage of literal and fill words in each of the a_i vectors, and the actual mix of operations performed in Table 3, would give us a better estimate of the execution time. We can apply an OR ordering similar to the BCP-EqJoin algorithm, deferring expensive OR's to latter stages of the algorithm. By working on compressed bit vectors instead of bitwise comparisons (i.e., operations in Table 3 instead of Table 2), we achieve a multiplicative factor execution time saving proportional to the word size. The worst case occurs when both the a_i vectors are composed of only literal words. Then we need to look at each bit to expand the result, and so the time for one cross-product would be $\mathcal{O}(n)$.

The compact output representation sketched out in Figure 3 leads to significant space savings. Consider a single cross-product OR computation. By not expanding the b's explicitly, we can cut down on the result space requirements by a multiplicative factor of $\mathcal{O}(l_b)$. We refer to this improved equijoin algorithm as CP-EqJoin.

3 Range Join Query Processing

The most general form of the range join takes an arbitrary distance function and a threshold, and finds all pairs of tuples chosen from the input relations whose values are within the specified threshold. In this paper, we focus on a more restrictive form of the range join that can be specified as a set of pairwise joins. This is equivalent to decomposing the distance function into individual dimensions. We denote each pairwise join condition between attributes $r(a)$ and $s(b)$ as $a \approx_\delta b$, which indicates a result relation that has merged tuples with attributes a and b within δ of each other. In SQL, the same join may be expressed as "a between $b-\delta$ and $b+\delta$." The equijoin discussed in the previous section is a special case. If the join involves more than two attributes, we call it a multiway range join. Our focus is on ad-hoc joins that will work on arbitrary pairs of columns.

In the equijoin examples, we did not explicitly specify the keys corresponding to the bit vectors in a bitmap. However, for range joins, we need to examine the values of keys to determine the bit vectors to use in the cross-product.

3.1 Basic Algorithm

Algorithm 3 gives the pseudo-code for range join evaluation using cross-products of bit vectors. Depending on the size of the band (the value of δ), we may need to evaluate a series of cross-products with the same left-hand-side $a_i \otimes b_p|a_i \otimes b_{p+1}| \ldots$. Instead, performing the logical OR operations before the cross-products (as depicted on line 4 of the algorithm) is much more efficient.

Typically, there is an additional set of *selection conditions* on the relations R and S. We evaluate these conditions using bitmap indexes and produce two bit vectors (denoted by m_R and m_S in the algorithm) to represent them. Note that these masks resemble the Positionally Encoded Record Filters (PERF) used in the PERF join algorithm [10]. However, the two major differences in our approach are that these masks can be efficiently computed using logical operations on bitmaps, and we can store these masks compactly with WAH compression. To reduce the cost of the cross-product function, we usually apply the masks before each invocation of the cross-product function.

Algorithm 3: CP-RangeJoin: A generic two-way range join algorithm based on cross-products of bit vectors.

Input: Bitmap indexes of join attributes a and b from relations R and S respectively. Additional masks on the relations, m_R and m_S.

Output: Relation Q corresponding to a range join on attributes a and b. The join condition specifies a threshold parameter δ. The result is stored in a compressed bit matrix C.

1 $C \leftarrow \phi$; (initialize an empty compressed bit vector for storing the join output)
2 **for** $i = 1$ *to* c_a **do**
3 Check whether k_{a_i} satisfies the join condition;
4 Examine keys of b to determine the largest interval (p, q) such that $[k_{b_p} \ldots k_{b_q}] \subseteq [k_{a_i} - \delta \ldots k_{a_i} + \delta]$;
5 $v_b \leftarrow m_S \,\&\, (b_p|b_{p+1}| \ldots |b_q)$;
6 $v_a \leftarrow m_R \,\&\, a_i$;
7 $C \leftarrow C \mid v_a \otimes v_b$;

Analysis: The algorithm iterates over each bit vector (key) of a and identifies the keys of b for which the join condition is satisfied. Thus, the outer for loop executes c_a times. However, if there is a range condition on a, for instance $\alpha \leq a \leq \beta$, then the number of keys inspected, and consequently the amount of work done in the cross-product, is reduced. Similarly, step 4 prunes some of the keys in b.

We assume that the result bit matrix is generated using the cross-product algorithms described in the previous section. In comparison to the equijoin algorithm, this approach performs more work in evaluating the range join condition in step 5. Having the key information of b readily available through the bitmap index is certainly advantageous. In the worst case, step 5 will have to sift through all the keys of b, leading to an asymptotic time complexity of $\mathcal{O}(h \cdot c_m \cdot c_n)$. Clearly, the straight-forward approach of creating the cross-product

matrix will only be useful for attributes of low cardinalities.

3.2 Determining the Join Selectivity Factor

For all the algorithms proposed so far, cross-product evaluation becomes expensive for large ($h >> \mathcal{O}(n)$) join selectivity factors. From Theorem 1, we have an estimate of the size of the join result. Further, if we assume equality encoding, then there is no overlap between the bit vectors. It leads us to a significant reduction in the complexity of range join evaluation. Note the following result for WAH-compressed data with equality encoding:

Lemma 1 *Let $b_1, b_2, \ldots, b_k$ denote the bit vectors corresponding to the bitmap index of an attribute b. Then, $b_i \,\&\, b_j$ is a null vector for all $1 \leq i \neq j \leq k$.*

This result implies that the the number of set bits in any OR computation of two a_i or b_i bit vectors is the sum of the set bits in the individual vectors, as these vectors do not overlap in their uncompressed form. This lets us very efficiently compute the tuple count in the join result for a range join (see Algorithm 4).

Algorithm 4: CP-JoinCount: A fast algorithm to determine the join selectivity factor.

Input: Bitmap indexes of join attributes a and b from relations R and S respectively.
Output: The number of tuples in the relation Q, corresponding to a range join on attributes a and b. The join condition specifies a threshold parameter δ.

1 $count \leftarrow 0$;
2 **for** $i = 1$ *to* c_a **do**
3 Examine keys of b to determine the largest interval (p, q) such that $[k_{b_p} \ldots k_{b_q}] \subseteq [k_{a_i} - \delta \ldots k_{a_i} + \delta]$;
4 $count_b \leftarrow count(b_p) + count(b_{p+1}) + \ldots + count(b_q)$;
5 $count_a \leftarrow count(a_i)$;
6 $count \leftarrow count + count_a \cdot count_b$;
7 $jsf \leftarrow \dfrac{count}{m \cdot n}$;

The count algorithm requires inspection of the keys of b on every iteration of the outer loop to evaluate the join condition. However, we can assume that the set bit counts in the bit vectors of b are readily available. The inner loop computation is thus $\mathcal{O}(c_m)$, leading to an overall asymptotic complexity of $\mathcal{O}(c_n \cdot c_m)$. Note that if we have additional masks to apply to the bit vectors, this nice relation will not hold. In that case, we need to explicitly compute the bit vector OR's. If we use WAH compression, we know that the cost of a bitwise logical operation is in the worst case a linear function of the total size (bytes or words) of the two input bit vectors. This leads us to an asymptotic time complexity of $\mathcal{O}(h)$. The memory footprint of this algorithm is smaller than a full cross-product evaluation, as we only need to operate on a few bit vectors in every iteration. We assume that m_R and m_S are available in main memory, and the b bit vectors can be loaded as required.

3.3 Faster Algorithms for Range Join

We can design faster algorithms for the full evaluation of range join if we assume that we are given an equality-encoded compressed bitmap index for the join attributes. Algorithm 4 provides us with the exact join selectivity factor and requires linear space to compute in most cases. If we know that the result count is significantly large (quadratic space requirements), we can try out alternate strategies for representing the cross-product output. The non-overlapping nature of the bit vectors again proves helpful. In Algorithm 1 (and Figure 2), we detail the approach of constructing a bit matrix cross-product. Observe that each column of the bit matrix in its uncompressed form is a result of an OR computation on some bit vectors of b. Further, since the bit vectors are non-overlapping, we can defer the OR computations until the latter stages of the algorithm. Instead we use c_b bits per column to store the bit vector identifiers of attribute b that we need to perform OR operations on. Algorithm 5 details a two-phase algorithm for the range join that first estimates or exactly computes the count, and then performs a full range join evaluation with the OR computation.

Algorithm 5: CP-FastRangeJoin: A memory-efficient algorithm for computing the range join.

Input: Equality-encoded bitmap indexes of join attributes a and b from relations R and S respectively; Threshold parameter δ for the range join.

Output: The output relation Q, corresponding to a range join on attributes a and b.

```
1  count ← 0;
2  Qc ← φ; (a list of compressed column vectors)
3  nzcol ← φ; (an uncompressed bit vector to keep a track of columns that are non-zero)
4  for i = 1 to c_a do
5      Examine keys of b to determine the largest interval (p, q) such that
       [k_{b_p} ... k_{b_q}] ⊆ [k_{a_i} − δ ... k_{a_i} + δ];
6      if q − p > 0 then
7          count_b ← count(b_p) + ... + count(b_q);
8          count_a ← count(a_i);
9          count ← count_a · count_b;
10         nzcol ← a_i | nzcol;
11         for each set bit j in a_i do
12             Qc[j] ← Append(b_p, b_{p+1}, .., b_q);
13 for each set bit i in nzcol do
14     for each set bit j in Qc do
15         Q[j] ← Q[j] | b_j;
```

Analysis: The first phase of this range join algorithm computes the result tuple count. In addition, it is also helpful to maintain a list of the b bit vectors that would appear at least once in a column of the result matrix. Let us denote this count by n_C. In case of an equijoin, the number of cross-product OR's we need to compute is given by n_{Ceq}. It is easy to see that for a range join, $n_C, n_{Ceq} \leq c_b$ and $n_C \geq n_{Ceq}$.

We initialize a list of pointers to the column vectors in uncompressed form to store the different non-zero columns of the bit matrix. *nzcol* is a bit vector in the algorithm that indicates whether a column is present in the result bit matrix. If it is present, then we require n_C bits per column to keep a track of the different b bit vectors that appear in this column. Note that this is a complete representation of the result which can be expanded into an uncompressed bit vector or matrix if required. Deferring the OR computation lets us cut down the space requirement of the result from $\mathcal{O}(h)$ (for a linearized, compressed bit vector representation of the bit matrix) to $\mathcal{O}(n \cdot n_C)$ (for this new two-dimensional data structure that is uncompressed in one dimension). Clearly, this representation is superior to the $\mathcal{O}(h)$-space structure for cases with large join selectivity factors ($h >> \mathcal{O}(n)$). Thus, based on the value of this count, we can choose an appropriate compressed bit vector representation for the result.

Let l_{nzcol} denote the number of set bits in *nzcol*. The total number of possibilities of bit vector OR's grows exponentially with n_C, precisely as 2^{n_C-1}. Contingent on the size of the resulting columns, one possibility is to precompute all possible bit vector OR's and step through *nzcol* assigning the appropriate pointers. This approach is certainly feasible in the case of very small values of n_C (< 10), and advantageous when $l_{nzcol} = \mathcal{O}(n)$. The other extreme would be to compute bit vectors using OR's as we go along identifying non-zero *nzcol* values. We expand l_{nzcol} columns in total, and the work done is proportional to the number of set bits, since the operation being performed is a compressed bit vector OR. Thus, the total amount of work done is again $\mathcal{O}(h)$ in this case. However, in comparison to the CP-RangeJoin algorithm, the memory footprint of the individual OR computations is significantly lower, as we defer all expansion to the end and we process one column at a time.

Finally, observe that we can possibly achieve a multiplicative speedup factor of $\mathcal{O}(l_{nzcol})$ for the expansion (this case arises when the same bit vector OR pattern occurs in every column, and we just compute it once and reuse it for the remaining $l_{nzcol} - 1$ cases). Intuitively, a range join would introduce bands of contiguous (in the key values) b vectors on which we need to perform OR operations on. Any greedy heuristic that tries to minimize the computation must be cognizant of the fact that the total amount of work to be performed in the straight-forward approach is still $\mathcal{O}(h)$. Depending on the relative values of l_{nzcol} and n_C, and the total amount of main memory available on the computing system, we may possibly precompute and store a few recurring bit vector patterns. From the various counts, we can estimate the running time and memory overhead for any heuristic. One greedy approach we use that appears to work well in practice, is to order the OR's by the size of the resulting vector, and store a selected number of the larger pairwise OR's that are reused.

4 Extensions

4.1 Filtering and Projections

Since most queries involving joins also contain additional conditions on the participating relations, an obvious way to take advantage of bitmap indexes is to use them to evaluate the range conditions. We will refer to the conditions on each relation participating in the joins as the *selection conditions on the relation*, or simply the selection conditions. The tuples satisfying the selection conditions are called the *selected rows*. After evaluating these selection conditions, we can apply one of the above cross-product based join algorithms.

A second possible approach to evaluate range joins is to *project out* the selected tuples of the join attributes. For most modest size joins, these projections often can be stored in memory and we may subsequently apply a traditional in-memory join algorithm. Assuming we can compute the in-memory join on the projection efficiently, this approach should be efficient overall. We use the nested-loop and sort-merge join algorithms to evaluate range joins on the projections. In both cases, if we only want to count the result tuple sizes, we do not need to access any identifiers of the selected rows. However, if we need the create a new relation satisfying the join conditions, we will need to maintain two sets of row identifiers, one for relation R and the other for relation S. By definition, the projection from R has n_R tuples and the projection from S has m_S tuples. Thus, we expect the nested-loop join algorithm to take $\mathcal{O}(n_R \cdot m_S)$ time if we do not need to generate the output bit matrix C. Similarly, the sort-merge join algorithm on the projections would take $\mathcal{O}(n_R \log(n_R) + m_S \log(m_S))$ time.

4.2 Multiway Joins

In many range joins, the distance functions are defined on multiple attributes. For example, in a query on network traffic data, one might perform a join on three out of four octets of source and destination IP addresses, such as, "R.sourceIP_A $\bowtie_0$ S.destinationIP_A and R.sourceIP_B $\bowtie_0$ S.destinationIP_B and R.sourceIP_C $\bowtie_0$ S.destinationIP_C." While the join operator is still applied on two tables, multiple range join operators $\bowtie_\delta$ are specified. A common strategy to process these queries is to evaluate one of the join operators using indexes and then scan the base data to resolve the remaining join conditions. We can use one of the cross-product based approaches to construct a compressed bit matrix of the result for one pairwise join. An alternative strategy is to process each join operator separately and then combine the results with an AND operation on the compressed bit vectors. In both cases, we can apply Algorithm 4 to estimate the counts for each pairwise join operation.

5 Experimental Study

We implement the new join algorithms discussed in this paper as subroutines of the FastBit bitmap indexing package. In this section, we provide a detailed performance analysis of the bit vector cross-product computation and the various join algorithms on both synthetic and

Table 4: A summary of the asymptotic complexity of new algorithms proposed in this paper.

Name	**Description**	**Complexity** ($\mathcal{O}$)	
		Execution time	Space
BCP	Cross-product construction using compressed bit vectors.	h	h
BCP-EqJoin	Equijoin using explicit OR operations on cross-product bit matrices.	$min(c_a, c_b) \cdot h$	h
CP-RangeJoin	Range join using cross-product bit matrices.	$c_a \cdot c_b \cdot h$	h
CP-JoinCount	Algorithm for determining the number of tuples in result.	$c_a \cdot c_b$	n
CP-FastRangeJoin	Faster algorithm for range join using implicit OR and cross-product computation.	$\min(h \cdot c_a \cdot c_b, n \cdot c_b)$	$\min(h, n \cdot c_b)$

real data. Table 4 summarizes the asymptotic worst-case complexities of the new compressed bit vector based algorithms presented in this paper. In addition, we also have implementations of the sort-merge and nested loop algorithms for performing equijoins and range joins. We observe that the sort-merge algorithm outperforms nested loop algorithm variants for all the queries studied in this paper, and hence we compare our new join algorithm only with the in-memory sort-merge algorithm. In future work, we will compare our algorithms to specialized join algorithms (such as indexed nested-loop join [13] and radix cluster join) that are tuned to answer particular classes of join queries efficiently.

We will mostly focus on joins involving two attributes in this section. The following are the key data parameters that determine the performance of our approaches: a) the number of tuples in the input relations, n and m; b) The attribute cardinalities c_a and c_b; c) The range threshold parameter δ; d) The join selectivity factor and the size of the result relation.

5.1 Test Setup

We use three sets of test data for our timing measurements: a synthetic data set with random integer values, a data set from a network traffic monitoring application, and a data set from the Sloan Digital Sky Survey. The synthetic data has random integers following the Zipf distributions. We experiment with several different table sizes and attribute cardinalities using three Zipf exponents (0, 1, 2). The network traffic data is collected using an intrusion detection system called BRO[2] [15]. The network data used here is a small fraction of the data logs collected at a mid-sized research organization. It contains IP address information,

[2]Information about BRO is also available at http://bro-ids.org.

port numbers, and timestamp information. The sky observations data was obtained from the Sloan Digital Sky Survey Data Release-1[3] and corresponds to three columns roughly of size 50 million tuples each.

All the tests were conducted on an Intel Core2 quad-core workstation running at 2.4 GHz. This system has 8 GB main memory and 4 MB of L2 cache. We built our codes with the GNU C compiler (version 4.2.4) and aggressive optimization enabled. We report wall-clock running time computed to a six-digit accuracy. Note that we perform in-memory joins on all the data sets, and do not time the I/O accesses for loading the bitmap indexes. The running time is dominated by in-memory computation for the problem instances we tested on. We compare our approaches with the compute-time for an in-memory sort-merge join algorithm, and in both cases, an identical procedure can be followed to materialize the final join table after the result tuple list is determined. Our compressed bitmap-based algorithms are a replacement to existing range join approaches such as the sort-merge algorithm, and can be combined with other techniques to alleviate the I/O time of materializing the join table.

5.2 Analysis

Since the running times of the algorithms are highly query-dependent, we first correlate them with run-time performance counts that accurately model the performance. Consider first the cost of evaluating just the join count using Algorithm 4. Figure 4 plots the execution time for various queries. On the X-axis, we indicate the expected number of bit vector OR operations that have to be performed to expand the entire join result. We observe that the running times are linear correlated with this count, and these results are consistent with the worst-case bounds given in Table 4.

Next, we study performance of the cross-product based algorithms for different types of join queries and input datasets. The queries on the real data sets are self-joins on different attributes. We observe that the two parameters introduced in Section 3.3, l_{nzcol} and n_C, capture variations in running time very effectively. As indicated in Table 4, the asymptotic upper bound is achieved when these two parameters reach their highest values.

In Figure 5, we plot the average execution time of four different range queries on synthetic data sets of different sizes. The values of l_{nzcol} and n_C for each query are indicated as the X-axis label tuple. We report performance of two variants of the CP-FastJoinCount algorithm: Defer OR and On-the-fly. CP On-the-fly implicitly evaluates cross-product OR's as they occur, while CP Defer OR applies ordering strategies and heuristics to defer OR computation to latter stages of the algorithm. We observe that in all four cases, CP On-the-fly outperforms the basic sort-merge join implementation. The speedup varies from 60× (in the case of the query with the least l_{nzcol} value) to 1.23×, with the selectivity factor being the third parameter influencing the performance. Also note that we do not explicitly produce a new result tuple list as output in the sort-merge algorithm, whereas the cross-product based relations compactly store the result in a bit matrix. Even with this work

[3]SDSS SkyServer: http://skyserver.sdss.org/dr1/en/sdss/release/.

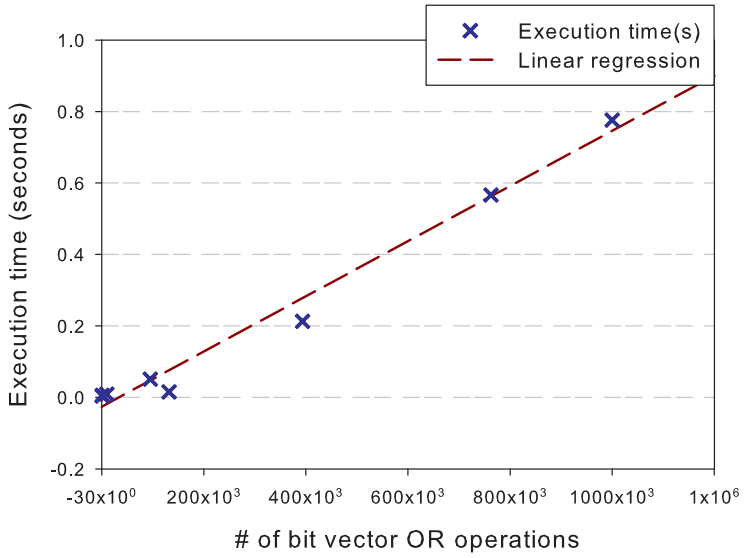

Figure 4: Performance of the join count algorithm for queries on both real and synthetic data, correlated with the number of bit vector OR operations.

imbalance, it is remarkable that the performance of the cross-product based approaches is significantly better. As expected, when there are fewer OR computations (lower l_{nzcol} values), the speedup is higher. We also observe that the running time of the Defer OR variant gets comparable to On-the-fly for larger values of l_{nzcol} and n_C. Lower values indicates that there is comparatively lesser room for reordering and precomputing heuristics. Finally, for both the implementations, as the product of l_{nzcol} and n_C increases, the running time goes up as well.

Figure 6 considers problem instances that are three orders of magnitude larger. In this case, we observe that the CP Defer OR is consistently faster than CP On-the-fly, and the reordering strategies pay off. We also observe that Defer OR outperforms Sort-merge for all the queries, with the speedup ranging from 1.37× to 44.7×. The queries are ordered in the increasing order of the join selectivity factor, starting with one that is $\mathcal{O}(n)$. As expected from the theoretical analysis, the bitmap index based approaches are faster than Sort-merge for high join-selectivity queries.

Since CP-JoinCount is a lower bound on the execution time, we indicate the ratio of these two running times in the performance charts. This number gives us an estimate of the potential speedup that we can achieve in processing this query.

We next evaluate join performance on the network data (Figure 7), where the attributes have nearly 30 million tuples and varying column cardinalities. We report the average performance of ten different queries falling in three categories: low join selectivity values, high n_C, and high join selectivity values. We plot the average running time of CP Defer OR and Sort-merge for these three cases, and observe that Defer OR is faster than Sort-merge for high n_C (similar to the results observed in Figure 5) and high join selectivity values (in

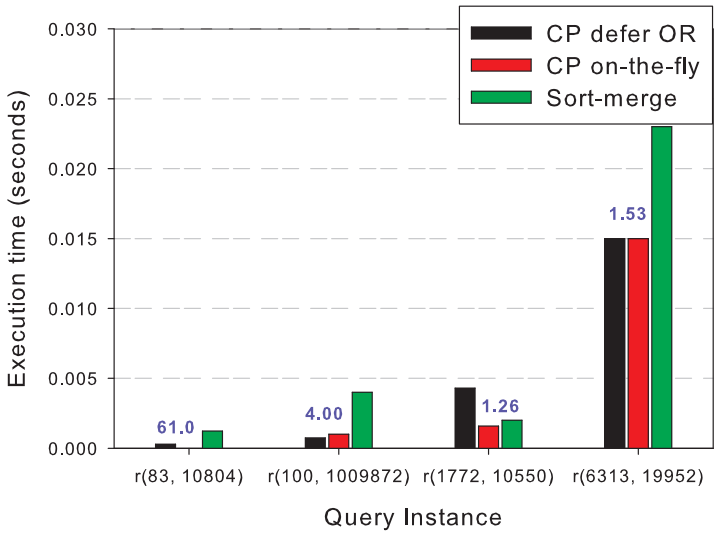

Figure 5: Performance of range join queries on synthetic Zipf data. The numbers above the bars indicate the speedup achieved by CP On-the-fly over Sort-merge, and the X-axis labels indicate values of n_C and l_{nzcol} respectively.

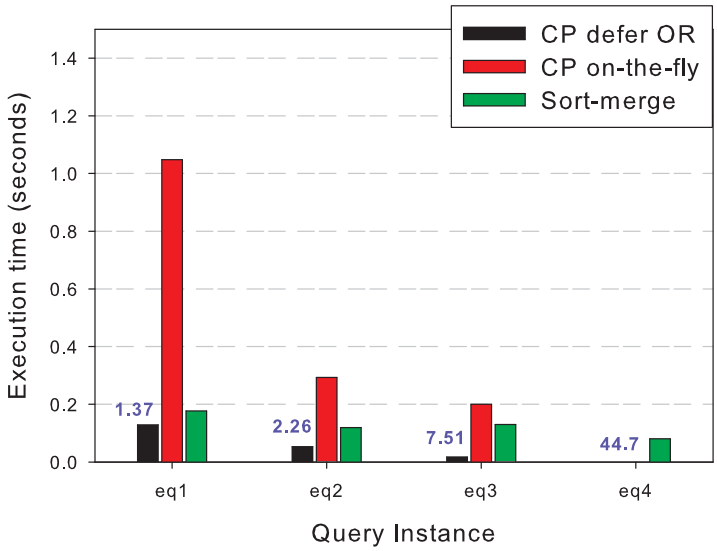

Figure 6: Performance of equijoin queries on a synthetic data set of 10 million tuples. The numbers above the bars indicate the speedup achieved by CP defer-OR over Sort-merge, and the queries on X-axis are ordered in increasing join selectivity values.

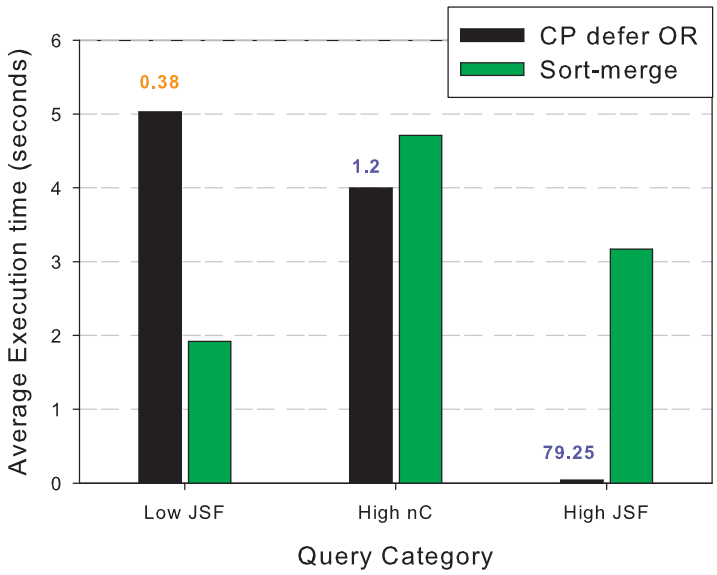

Figure 7: Average performance of join queries on the network data set. The numbers above the bars indicate the speedup achieved by CP defer-OR over Sort-merge, and three different query categories are indicated on the X-axis.

accordance with results in Figure 6). However, for the case of low join selectivity factor (in addition to l_{nzval}), the sort-merge algorithm is expectedly faster than the Defer OR variant. The observed performance is consistent with our theoretical analysis of the bounds. Note that we have not given the performance of sub-optimal approaches such as BCP-EqJoin and the nested loop algorithm. These approaches only get competitive with the CP Defer OR algorithm for cases where $h = \mathcal{O}(m \cdot n)$, and in these scenario we already demonstrate a significant speedup over the Sort-merge algorithm.

In Figure 8, we summarize the speedup achieved by our index based counting strategy (algorithm CP-JoinCount) over the sort-merge algorithm for the various query instances discussed previously. If we just need to determine the count of a join result, our approach is extremely fast and can result in a speedup up to three orders of magnitude in practice (for queries with high join selectivity factors). These speedup figures are also an indicator of the heuristic optimizations of our cross-product based approaches over the Sort-merge algorithm. For instance, the query class of low join selectivity factors may benefit from further reordering or deferred OR computation, as we obtain a speedup of 9.6× for a count-only query.

Figure 9 is an illustration of the optimization parameter space for the cross-product based algorithms. Based on the theoretical analysis and experimental results, we indicate regions where we observe one approach to perform better significantly better than the other. As the experiments indicate, there is still some performance to be realized from the bottom-right corner for large n_C and l_{nzcol} values, as well as low join selectivity factors.

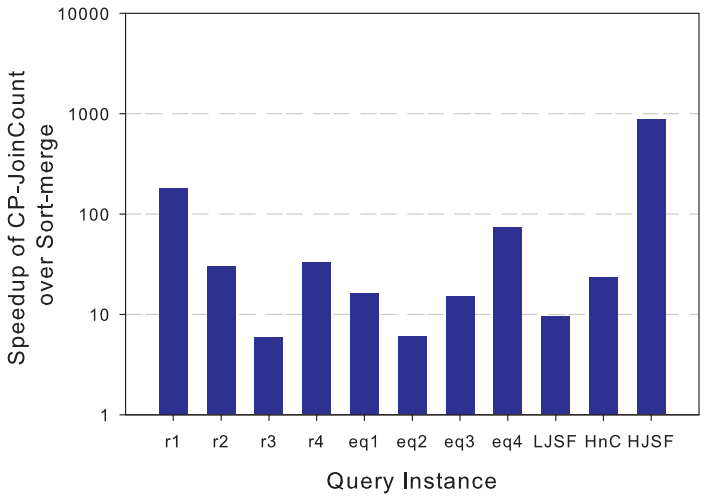

Figure 8: Average speedup (logarithmic scale) achieved by CP-JoinCount over Sort-merge for various query instances.

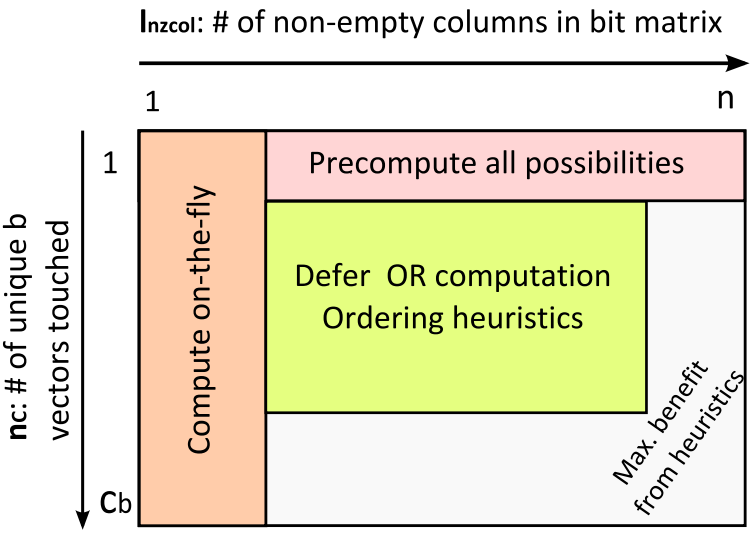

Figure 9: A depiction of the optimization state space for the CP-FastRangeJoin algorithm.

6 Conclusions

We present a set of new algorithms in this paper that utilize bitmap indexes efficiently to evaluate range joins. Our approaches are faster in practice than well-known algorithms such as sort-merge based approaches. With a combination of theoretical and empirical results, we attempt to characterize the join query and algorithm space, and reduce the performance gap between simple counting and full join result expansion.

In future work, we will conduct a deeper theoretical study and experiment with other heuristics for the deferred OR computation join algorithm. Several algorithms discussed in this paper are amenable to parallelization, and it would be interesting to see what the bottlenecks to performance would be on current parallel systems. We also wish to extend our current prototype implementations of join algorithms to process large-scale partitioned datasets and more diverse query workloads.

Acknowledgments

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] G. Antoshenkov. Byte-aligned bitmap compression. U.S. Patent number 5,363,098, 1994.

[2] P. Bizarro and H. Madeira. The Dimension-Join: A new index for data warehouses. In *Proc. XVI SBBD*, pages 259–273, Rio de Janeiro, Brazil, Oct 2001.

[3] C.-Y. Chan and Y.E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2):355–366, 1998.

[4] C.-Y. Chan and Y.E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *SIGMOD Rec.*, 28(2):215–226, 1999.

[5] D.J. DeWitt, J.F. Naughton, and D.A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. VLDB*, pages 443–452, Barcelona, Spain, Sep 1991.

[6] J. Gray et al. There goes the neighborhood: Relational algebra for spatial data search. Technical report, Microsoft Research, 2004. MSR-TR-2004-32.

[7] T. Johnson. Performance measurements of compressed bitmap indices. In *Proc. VLDB*, pages 278–289, Edinburgh, Scotland, Sep 1999. Morgan Kaufmann.

[8] N. Koudas. Space efficient bitmap indexing. In *Proc. CIKM*, pages 194–201, McLean, VA, Nov 2000. ACM.

[9] N. Koudas and K.C. Sevcik. Size separation spatial join. *SIGMOD Rec.*, 26(3):324–335, 1997.

[10] Z. Li and K.A. Ross. PERF join: an alternative to two-way semijoin and bloomjoin. In *Proc. CIKM*, pages 137–144, Baltimore, MD, Nov–Dec 1995. ACM.

[11] P. Mishra and M.H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[12] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, 1995.

[13] P. O'Neil and E. O'Neil. *Database: principles, programming, and performance*. Morgan Kaufmann, 2nd edition, 2000.

[14] P. O'Neil and D. Quass. Improved query performance with variant indexes. *SIGMOD Rec.*, 26(2):38–49, 1997.

[15] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[16] R. Power. Large catalogue query performance in relational databases. *Publications of the Astronomical Society of Australia (PASA)*, 24(1):13–20, 2007.

[17] T. Siqueira, R. Ciferri, V. Times, and C. Ciferri. Investigating the effects of spatial data redundancy in query performance over geographical data warehouses. In *Proc. GEOINFO*, Rio de Janeiro, Brazil, Dec 2008.

[18] H.K.T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proc. VLDB*, pages 448–457, Stockholm, Sweden, Aug 1985.

[19] K. Wu, E.J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, pages 24–35, Toronto, Canada, Sep 2004. Morgan Kaufmann.