Slicing the Dimensionality: Top-k Query Processing for High-dimensional Spaces

Gheorghi Guzun, Joel E. Tosado, and Guadalupe Canahuate

Department of Electrical and Computer Engineering
The University of Iowa, Iowa City, IA, USA
{gheorghi-guzun,joel-tosadojimenez,guadalupe-canahuate}@uiowa.edu
http://www.uiowa.edu

Abstract. Top-k (preference) queries are used in several domains to retrieve the set of k tuples that more closely match a given query. For high-dimensional spaces, evaluation of top-k queries is expensive, as data and space partitioning indices perform worse than sequential scan. An alternative approach is the use of sorted lists to speed up query evaluation. This approach extends performance gains when compared to sequential scan to about ten dimensions. However, data-sets for which preference queries are considered, often are high-dimensional. In this paper, we explore the use of bit-sliced indices (BSI) to encode the attributes or score lists and perform top-k queries over high-dimensional data using bit-wise operations. Our approach does not require sorting or random access to the index. Additionally, bit-sliced indices require less space than other type of indices. The size of the bit-sliced index (without using compression) for a normalized data-set with 3 decimals is 60 times smaller than the size of sorted lists. Furthermore, our experimental evaluation shows that the use of BSI for top-k query processing is more efficient than Sequential Scan for high-dimensional data. When compared to Sequential Top-k Algorithm (STA), BSI is one order of magnitude faster.

Keywords: Top-k queries · Preference queries · High-dimensional data

1 Introduction

Top-k processing techniques attempt to efficiently find the top k results from data sources. Objects, or data items, pertaining to these data sources may be described through multiple numerically-valued attributes, or dimensions. An object's numeric value for a specific attribute is its local score for that attribute. Top-k techniques rely on ranking functions in order to determine an overall score for each of the objects across all the relevant attributes being examined [1]. This overall score may then be used to choose the top-k results. Top-k processing is a crucial requirement across several applications or domains. In the multimedia domain numerically-valued feature vectors describe the multimedia objects. An image, for example, may be described by thousands of feature vectors [2, 3, 4]. Users usually search for multimedia objects for which they only desire the best matching result that derive from the overall grade of match, or overall score of the features. [5, 6].

In the domain of information retrieval, consider a search engine tasked in retrieving the top-k results from various sources. In this scenario, the ranking function considers scores based on word-based measurements, as well as hyperlink analysis, traffic analysis, user feedback data, among others, to formulate its top-k result [7, 8].

Other domains and applications such as monitoring networks [9, 10], P2P systems [11], data stream management streams [12], restaurant selection systems [13], among others, also rely on top-k processing techniques [14, 15].

Data in these domains is often high-dimensional with over a hundred of attributes. The curse of dimensionality relates to the negative impact the increase of dimensionality has on query processing techniques. Top-k techniques express their effects through either high time or space complexity as dimensionality increases [16]. E.g. indexing structures are not efficient for dimensionality sometimes as low as 6 dimensions[17]. Moreover, this effect pervades to the extent where the top-k techniques perform worse than sequential scan [17].

A popular top-k processing technique, the Threshold Algorithm (TA) [14], and some of its optimizations [17, 18, 19, 20, 21] involve the use of sorted lists for each attribute. The idea is that the collection of these sorted lists allow an efficient computation of the top-k aggregated overall scores with as few accesses to the data as possible. These sorted lists avoid scanning of the entire list of entries for each of the attributes under consideration. Moreover, a threshold value is used to determine when the algorithm should stop while guaranteeing the top-k results. TA-based techniques extend the performance benefits to tens of dimensions.

In this paper, we explore the evaluation of top-k queries in high-dimensional spaces without the use of sorted lists or hierarchical indices. We propose the use of bit-sliced indices (BSI) to encode the score list and perform top-k queries over these high-dimensional data. Since our index is not sorted, new data is just appended to the end. The bit-slices are added and resulting in another BSI. From this aggregate ranking a top-k result can be derived using bit-wise operations.

The primary contributions of this paper can be summarized as follows:

– We developed efficient algorithms to evaluate top-k queries over bit-sliced indexing exclusively using bit-wise operations.
– We analyze the cost of the proposed algorithms in terms of algorithm complexity.
– We evaluate three types of top-k queries: *top-k queries*, *Boolean preference queries*, *weighted preference queries.*
– We evaluate the cost of the proposed approach in terms of index size and also query time.
– We compare performance gains against Sequential Scan (SS), Threshold Algorithm (TA), Sequential Top-k Algorithm(STA) and Best Position Algorithms (BPA and BPA2).
– We perform experiments over both synthetic and real datasets.

Table 1: Notation Reference

Notation	Description
n	Number of rows in the data
m	Number of attributes in the data
s, p	Number of slices used to represent an attribute
w	Computer architecture word size
Q	Query vector
$\|q\|$	Number of non-zero preferences in the query
b	Number of bits used to represent a query preference

The rest of the paper is organized as follows. Section 2 presents background and related work. Section 3 describes the problem formulation and the proposed solution using bit-sliced indices. Section 4 presents the cost analysis of the proposed algorithms. Section 5 shows experimental results. Finally, conclusions are presented in Section 6.

2 Background and Related Work

This section presents background information for bit-sliced indices and related work for top-k query processing. For clarity, we define the notations used further in this paper in Table 1.

2.1 Bit-Sliced Indexing

BSI [22, 23] can be considered a special case of the encoded bitmaps [24]. With bit-sliced indexing, binary vectors are used to encode the binary representation of the attribute value. Only $\lceil \log_2 values \rceil$ vectors are needed to represent all values. One BSI is created for each attribute.

Figure 1 shows an example of the BSIs for two attributes and their sum. Since each attribute has three possible values, the number of bit-slices for each BSI is 2. For the sum of the two attributes, the maximum value is 6, and the number of bit-slices is 3. The first tuple t_1 has the value 1 for attribute 1, therefore only the bit-slice corresponding to the least significant bit, $B_1[0]$ is set. For attribute 2, since the value is 3, the bit is set in both BSIs. The addition of the BSIs representing the two attributes is done using efficient bit-wise operations. First, the bit-slice $sum[0]$ is obtained by XORing $B_1[0]$ and $B_2[0]$ i.e. $sum[0] = B_1[0] \oplus B_2[0]$. Then $sum[1]$ is obtained in the following way $sum[1] = B_1[1] \oplus B_2[1] \oplus (B_1[0] \wedge B_2[0])$. Finally $sum[2] = Majority(B_1[1], B_2[1], (B_1[0] \wedge B_2[0]))$.

BSI arithmetic for a number of operations is defined in [23]. We adapt two of these bit-sliced operations (sum and topK) as components in our top-k query processing and define a new operation to multiply a BSI by the query preference.

	Raw Data		Bit-Sliced Index (BSI)				BSI SUM		
			Attrib 1		Attrib 2				
Tuple	Attrib 1	Attrib 2	$B_1[1]$	$B_1[0]$	$B_2[1]$	$B_2[0]$	$sum[2]$[3]	$sum[1]$[2]	$sum[0]$[1]
t_1	1	3	0	1	1	1	1	0	0
t_2	2	1	1	0	1	0	0	1	1
t_3	1	1	0	1	0	1	0	1	0
t_4	3	3	1	1	1	1	1	1	0
t_5	2	2	0	1	1	0	1	0	0
t_6	3	1	1	1	0	1	1	0	0

[1]$sum[0]$=$B_1[0]$ XOR $B_2[0]$, $C_0 = B_1[0]$ AND $B_2[0]$
[2]$sum[1]$=$B_1[1]$ XOR $B_2[1]$ XOR (C_0)
[3]$sum[2]$=C_1=$Majority(B_1[1],B_2[1],(C_0))$

Fig. 1: Simple BSI example for a table with two attributes and three values per attribute.

2.2 Top-k Queries

Numerous techniques have been proposed to process top-k queries using a variety of data and space partitioning indices. However, their performance degrades for high-dimensional spaces [1, 17, 25].

The preferred algorithms for multi-dimensional spaces stem from the Threshold Algorithm (TA). Thus, before addressing these optimizations, here we present the TA algorithm [14].

1. Do sorted access in parallel to each of the m sorted lists L_i. As an object R is seen under sorted access in some list, do random access to the other lists to find the grade x_i of object R in every list L_i. Then compute the grade $t(R) = t(x_1, \ldots, x_m)$ of object R. If this grade is one of the k highest we have seen, then remember object R and its grade $t(R)$ (ties are broken arbitrarily, so that only k objects and their grades need to be remembered at any time).
2. For each list L_i, let x_i^L be the grade of the last object seen under sorted access. Define the threshold value τ to be $t(x_1^L, \ldots, x_m^L)$. As soon as at least k objects have been seen whose grade is at least equal to τ, then halt.
3. Return the top-k objects as the query answer.

Correctness of TA follows from the fact that the threshold value τ represents the best possible score that any object not yet seen can have, and TA stops when it can guarantee that no unseen object might have a better score than the current top-k ones [1, 21, 26].

The Best Position Algorithms (BPA and BPA2) [15], were proposed as optimizations of TA where the threshold condition is improved. It is determined from the best positions from each of the lists above which all scores have been seen. BPA2 further improves over BPA by avoiding accessing the same position several times, by doing direct access to the position which is just after the best position. The IO-Top-k technique [27] involves a balancing of seek time and transfer rate by mixing the amount of sorted and random accesses performed. Moreover, random accesses are controlled given the probability of a certain record to be in the top-k.

While TA relies on sorted and random access, the NRA variant [14] has no random access for applications where random access is overly expensive. The approach consists of two phases, one where it extracts candidate objects gradually from the top of the sorted lists. This will continue until the threshold condition is met. The second phase will gradually establish that no remaining candidates are better than the current top-k and will then terminate. The Threshold Algorithm over Bucketized Sorted Lists with Bloom Filters (TBB) [26] improves upon NRA by estimating the sequential scanning required and thus reducing disk access. The 3-phased no random access algorithm (3P-NRA) also improves upon NRA [28] by reducing the expensive probing of candidates in the NRA algorithm. However, 3P-NRA and TBB generally incur the large maintenance cost of the potential candidates and sorted lists [21].

Numerous other techniques have been developed for the top-k query processing that also originate, or involve concepts, from the TA technique [18, 19, 20, 29, 30]. However, these and other top-k techniques such as view-based techniques do not perform well for high dimensional data [1, 17, 21].

Recently, Sequential Top-k Algorithm (STA) was proposed to preprocess the sorted lists in order to reduce the space complexity and allow for the early termination feature, by a threshold value, as in TA. It eliminates the need to store the object identifier for each attribute as is the case when using sorted lists. Instead it stores a single identifier with all the attributes and a leaner indicator attribute (1 byte to address 256 attributes). This indicator attribute determines when an object is first seen, essentially capturing when a new object is accessed as in the sorted access of TA. Additionally, the resulting table from the preprocessing is sequentially accessed and used to dynamically update the threshold element. Furthermore, STA also avoids random access, such as in NRA, when evaluating top-k queries. In its paper, STA was compared against TA, 3P-NRA, BPA, IO-Top-k, and TBB. The experimental results on synthetic data sets for increasing dimensionality (up to 16) state a 1-2 order of magnitude improvement over these other top-k processing techniques [21]. Thus, we compare STA against our approach.

Note that our approach does not maintain sorted lists nor does it use a threshold computation as a stopping condition. Bit-Sliced Indexing (BSI) is used to compute the scores and then process the top-K query using fast logical operations supported by hardware. Since the scores are computed for all the tuples, performance is not considerably affected by the value of k. This is in contrast to both sequential scan and other list-based approaches (TA, STA, etc.) which maintain an auxiliary data structure of size $O(k)$. Additionally, the BSI index being a variant of the bitmap index, can benefit from word-aligned compression and can efficiently support other types of queries such as selection and aggregation queries.

3 Proposed Approach

In this section we first formulate the top-k queries supported in this paper and then describe the query execution algorithm using bit-slice indexing.

3.1 Problem Formulation

Consider a relation R with m attributes or numeric scores and a preference query vector $Q = \{q_1, \ldots, q_m\}$ with m values where $0 \leq q_i \leq 1$. Each data item or tuple t in R has numeric scores $\{f_1(t), \ldots, f_m(t)\}$ assigned by numeric component scoring functions $\{f_1, \ldots, f_m\}$. The combined score of t is $F(t) = E(q_1 \times f_1(t), \ldots, q_m \times f_m(t))$ where E is a numeric-valued expression. F is monotone if $E(x_1, \ldots, x_m) \leq E(y_1, \ldots, y_m)$ whenever $x_i \leq y_i$ for all i. In this paper we consider E to be the summation function: $F(t) = \sum_{i=1}^{m} q_i \times f_i(t)$. The k data items whose overall scores are the highest among all data items, are called the top-k data items. We refer to the definition above as **top-k weighted** preference query.

In this paper we also consider two special cases of query vectors:

– **top-k baseline**: the query vector is all 1s.
– **top-k boolean**: the query vector is either 0 or 1 for each attribute.

These two specializations of the top-k preference query are less expensive to compute than the general top-k preference query but also have numerous applications in the literature, e.g. spatial searches with text constraints in geographical collections [31].

The rest of this section describes the proposed approach for answering the general top-k preference queries.

3.2 Top-k Preference Query Execution

Let us denote by B_i the bit-sliced index (BSI) over attribute i. A number of slices s is used to represent values from 0 to $2^s - 1$. $B_i[j]$ represents the j^{th} bit in the binary representation of the attribute value and it is a binary vector containing n bits (one for each tuple). The bits are packed into words, the storage requirement for each binary vector is n/w, where w is the computer architecture word size (64 in our implementation).

In order to compute the score for each data point we first multiply the attribute value by the query preference for that attribute using bit-wise operations. Given a query Q, the query vector is first converted to integer weights based on the desired precision. Let us denote by b, the number of bits used to represent a query preference. The preference query execution algorithm pseudo-code is given in Algorithm 1.

We identify three main parts in our main algorithm 1:

1. For all non-zero weights, multiply the BSI for the attribute with the corresponding query weight (Lines 3 and 6).

```
prefQuery (B,q,k)

1:  **if** ($k < 0$)
2:     **Error ("k is invalid")**
3:  S = Multiply ($q_j, B_j$) where j is the first non-zero weight in q
4:  **for** ($i = j + 1; i \leq m; i{+}{+}$)
5:      **if** $q_i > 0$
6:          S = SUM_BSI(S, Multiply ($q_i, B_i$))
7:  T = TopK (S, k)
8:  return T
```

Algorithm 1: Preference query execution using bit-slices. B is the set of all BSIs, q is the query vector, and k is the desired number of results.

2. Sum the partial scores produced by the Multiply algorithm into a BSI S (Line 6).
3. Find the top k data points given the final BSI score S.

Algorithm 1 calls the *Multiply* function on line 3. Algorithm 2 shows the pseudocode for multiplication of a query weight with the attribute values. It follows the same logic as a sequential multiplier with the difference that the multiplicand is a BSI, not a single number.

First, the three bit-arrays R, S and C are initialized to be all-zeros. R is the product BSI, while S and C represent the sum, and carry bit-vectors for every slice. When the multiplier's least significant bit is set (Line 5), the multiplicand's BSI is shifted and added to the result (Lines 6-13). Otherwise the multiplicand's BSI is shifted until it finds its least significant bit set. If the final bit-array C, representing the carry bits has at least one set bit (Line 14), then the number of slices in the result R is incremented by one (Lines 15-16). The multiplier is shifted before the next iteration of the loop (Line 17). The shift amount is stored as an offset counter (Line 18).

The return BSI R represents the partial score of each tuple for a single attribute. Shifting the bit-slices does not impact the performance cost of the addition operation, as the shifting offset is simply passed along when accessing the bit-slices.

Further, Algorithm 1 calls the SUM_BSI function on line 6. Algorithm 3 shows the pseudocode for the addition of two BSIs, A and B, with a number of slices s and p, respectively. The result is another BSI S with MAX(p, s)+1 slices. A "Carry" bit-slice C is used in Algorithm 3 whenever two or three bit-slices are added to form S_i, and a non-zero C must then be added to the next bit-slice S_{i+1}. Once the bit-slices in either A or B are exhausted, calculations of C are likely to result in zero in very few iterations soon after. An example of a BSI sum was shown earlier in Figure 1.

Finally, the top k tuples are selected by calling the top-k algorithm presented in Algorithm 4 over the sum BSI representing the scores. The top-k algorithm

```
Multiply ($q_i$, B)

1:  R = ∅                                      //The product BSI
2:  offset=0                                   //Shift factor
3:  bitArray S = ∅,C = ∅                       //Sum and carry bits
4:  while ($q_i$ > 0)
5:     if (number & 1 == 1)                    //if the last bit is set
6:        for (i = 0; i < B.slices; i++)       //Add the slices
7:              S = B[i] XOR R[i+offset] XOR C
8:              C = Majority(B[i],R[i+offset],C)
9:              R[i+offset]=S;
10:       for (j = i+offset; j < B.slices; j++) //Add C to the remaining slices
11:             S = R[j] XOR C;
12:             C = R[j] AND C;
13:             R[j] = S;                      // add the slice S to the product
14:       if (count(C) > 0)
15:             R[B.slices]=C;                 // Carry bit
16:             R.slices++;
17:    $q_i$ >>= 1;                            // shift the query weight
18:    offset++;                               //update the offset
19: return R;
```

Algorithm 2: Multiplication algorithm. q_i is the integer representation of the query weight for attribute i and B is the BSI representing attribute i.

starts evaluating the scores from the most significant bits. Variables used in Algorithm 4 exist from one loop pass to the next. The bitArrays G, E and X, and positive integer *count* are only temporary, used to hold results within a loop pass for efficiency. The G bit-array represents the data points with larger values seen so far, and the E bit-array represents the data points that are currently tied. We initialize E to be an all 1s bit-array and G as an empty set.

In the first iteration, X gets the values from the most significant bit-slice. If the number of set bits in this slice is greater than k then E is also assigned $S[i]$. Otherwise E will store the tuples that do not have the most significant bit set. For the next iteration X is assigned G OR (E AND $S[i]$) (the tuples in E that also have set bits in $S[i]$). Then, if the number of set bits in X is greater than k $E = E$ AND $S[i]$ otherwise $E = E$ AND $S[i]$. Algorithm 4 will continue iterating through the bit-slices of S until *count* equals to k and will return the resulting k tuples in the form of a bit-vector (F).

Example: Putting It All Together. In this sub-section we explain how Algorithms 1, 2, 3 and 4 work together through a simple example. The numbers in parentheses reference to the numbers in Figure 2.

```
SUM_BSI(A,B)

1:  $S[0] = A[0]$ XOR $B[0]$                  // bit on in $S[0]$ iff bit on either $A[0]$ or $B[0]$
2:  $C = A[0]$ AND $B[0]$                     // $C$ is "Carry" bit-slice
3:  for $(i = 1; i <$MIN$(s,p); i{+}{+})$     // While there are bit-slices in $A$ and $B$
4:      $S[i] = (A[i]$ XOR $B[i]$ XOR $C)$     // one (or three) bit on gives bit on in $S_i$
5:      $C =$Majority$(A[i], B[i], C)$        // two (or more) bits on gives bit on in $C$
6:  if $(s > p)$                              // if $A$ has more bit-slices than $B$
7:      for $(i = p + 1; i \leq s; i{+}{+})$  // continue loop until last bit-slice
8:          $S[i] = (A[i]$ XOR $C)$            // 1 bit on gives bit on in $S[i]$; $C$ might be $\emptyset$
9:          $C = (A[i]$ AND $C)$               // two bits on gives bit on in $C$
10: else                                      // $B$ has at least as many bit-slices as $A$
11:     for $(i = s + 1; i \leq p; i{+}{+})$  // continue loop until last bit-slice
12:         $S[i] = (B[i]$ XOR $C)$            // one bit on gives bit on in $S[i]$
13:         $C = (B[i]$ AND $C)$               // two bits on gives bit on in $C$
14: if ($C$ is non-zero)                      // if still non-zero Carry after bit-slices end
15:     $S[MAX(s,p) + 1] = C$                 // Put Carry into final bit-slice of $S$
```

Algorithm 3: Addition of BSI's. Given two BSI's, A and B, we construct a new sum BSI, S = A + B, using the following pseudo-code. We must allow the highest order slice of S to be $S[MAX(s,p) + 1]$, so that a carry from the highest bit-slice in A or B will have a place.

As an illustrative example, consider a table with two attributes where each tuple represents a bag of white and black pebbles. Pebbles' counts correspond to attributes "White" and "Black". Figure 2 shows an example of a top-k preference query over this table using the Bit-Sliced Index Arithmetic. The query asks for the two bags, out of all of bags, that contain the most pebbles given a preference query with weights 0.4, and 0.6 for the white and black pebbles, respectively.

Initially, the data is indexed using a BSI, and each column in the attribute bitmap is saved as a bit-vector called bit-slice (the most significant bit-slice of attribute "White" is shaded in Figure 2). Since this data has a low value range, there are only three bit-slices per attribute. All the bit-slices form the BSI index. Using the BSI index and Algorithm 1, the query will be performed as illustrated in Figure 2. The weight multiplication is performed by shifting and adding slices, as described in Algorithm 2. The query weights are treated as integers, and thus 0.4 and 0.6 are scaled to 4 and 6 respectively. For every set-bit in the transformed query weight, the BSI index is shifted to the left by the number that describes the set-bit position in the query bit-string ($offset$ in Algorithm 2). For example, the query weight for the first attribute in the figure, has only one set bit. Thus Algorithm 2 "shifts" the BSI for this attribute to the left by two 2 (second rightmost is the position of the set-bit - starting from 0) (1). This result is then added to a sum BSI using algorithm 3 (8). For the second attribute, Algorithm 2 first shifts the BSI by 2 and keeps it as an intermediate result (R in Algorithm 2) (3,4), then shifts it by 1 and adds it to R again (5,6).

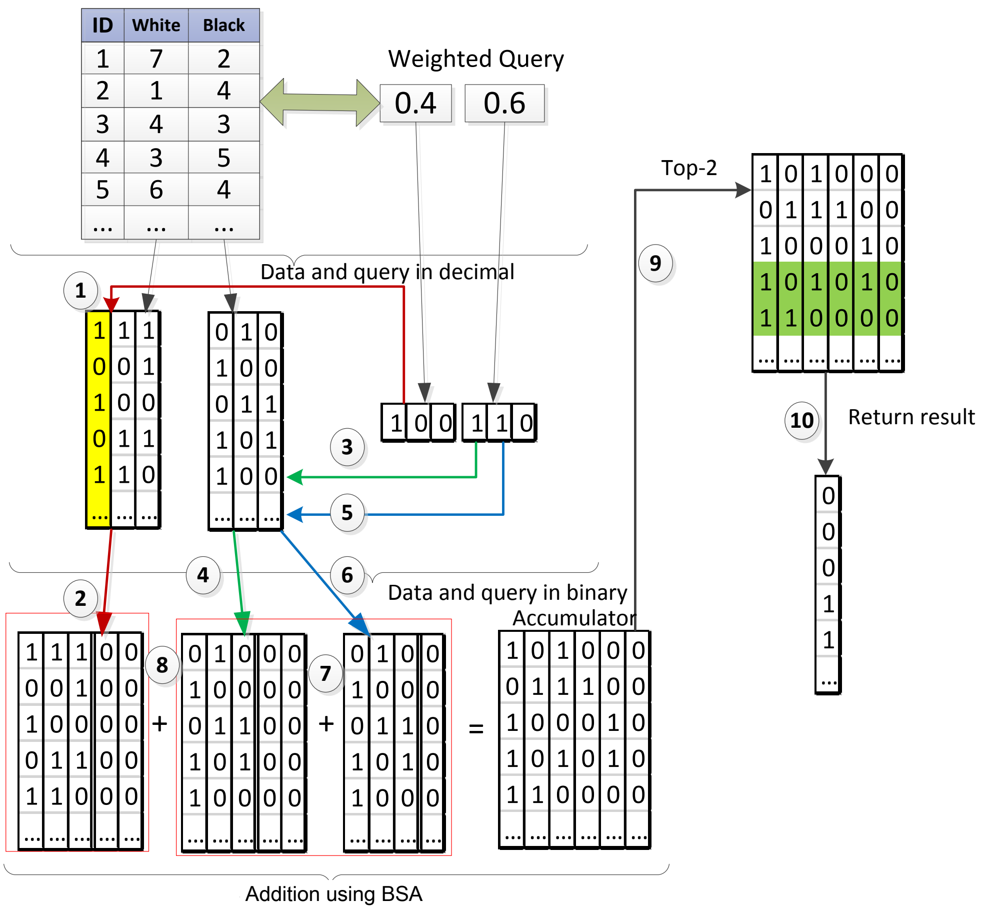
```
TopK(S,k)

1:  $G = \emptyset$
2:  $E$ =All1s                                  // Initialize an all ones bitVector
3:  for $(i = s-1; i \geq 0; i–)$               //While there are bit-slices in $S$
4:      $X = G$ OR ($E$ AND $S[i]$)             //X is trial set: G OR
                                                //(rows in E with 1-bit in position i)
5:      if (($count$ =COUNT($X$))> $k$)         //If more candidates than k
6:          $E = E$ AND $S[i]$                  //Intersection of E and current slice
7:      else if ($count < k$)
8:          $G = X$                             //G in next pass gets all rows in X
9:          $E = E$ AND NOT($S[i]$)             //E in next pass contains no rows r
                                                //with bit i on in S(r)
10:     else                                    // count==k
11:         $E = E$ AND $S[i]$                  //all rows r with bit i on in
                                                //S(r) will be in E
12:         break
13: $count$ =COUNT($G$)                         //Update count
14: if $count < k$                              //if count is greater than k
15:     $E$ = pick($E$,$k - count$)             //Pick k-ncount candidates to break ties
16: else
17:     $E = \emptyset$
18: $F = G$ OR $E$
19: return $F$
```

Algorithm 4: Find k tuples with the largest values in S (ties are arbitrarily broken), s is the number of slices in S.

Once Algorithm 2 completes the multiplication, its result is added to the sum BSI, using Algorithm 3 (7,8).

After the multiplications and additions are complete for all the attributes, Algorithm 4 is applied over the sum BSI (9). Algorithm 4 traverses the sum BSI starting from the most significant bit (from left to right), and finds the top-k tuples by eliminating candidates. First it eliminates all the tuples that do not have a set-bit in the left-most bit-slice, and saves the tuples that have a set-bit in an intermediate result (X in Algorithm 4). Then it moves to the second left-most bit-slice. In this example the number of tuples that have set-bits and also had set-bits in the previous bit-slice is smaller than two (only one). This single tuple is marked as to be saved in X. Everything else is also kept in G (Algorithm 4). For the next bit-slice, there are two tuples with set-bits, that are also in G. These tuples, plus the one in X are now saved to G. At this point the intermediate result contains 3 tuples. Algorithm 4 will continue this way until the desired number of tuples remains in the intermediate result. In Figure 2 the goal is to extract top-2 tuples, and thus Algorithm 4 will stop once this number is achieved. If the number of slices is exhausted and Algorithm 4 is not able to discriminate between some tuples (i.e. their score is the the same), ties are



ID	White	Black
1	7	2
2	1	4
3	4	3
4	3	5
5	6	4
...	...	...

Fig. 2: Example of BSI Arithmetic applied for finding top-2 tuples given a weighted preference query

broken arbitrarily. Finally, the result is returned in form of a bit-slice, having set-bits for the tuples that meet the user's criteria (10).

4 Cost Analysis

In this section we analyze the cost of computing top-k preference queries using the proposed BSI approach both in terms of index size and query execution time.

4.1 Index Size

The analysis is based on verbatim or uncompressed bit-sliced indexing (BSI). The BSI size is independent of data distribution and only depends on the number of rows n and the number of slices used to represent each attribute. Each bit slice has n bits, which are packed into words. The number of slices per attribute can be computed from the attribute cardinality c_i (number of distinct values) as: $s_i = \lceil \log_2 c_i \rceil$. The index size (in bits) can then be computed as:

$$IndexSize = \sum_{i=1}^{m} s_i n \tag{1}$$

where m is the number of attributes, n is the number of tuples, and s_i is the number of slices used to represent the value of attribute i. For top-k queries it makes sense to normalize the attribute values to avoid one attribute with extremely large values to dominate the final score. Assuming the attributes are normalized between 0 and 1, the same number of slices can be used for all the attributes. The number of slices depends on the desired resolution to represent the normalized attribute values. For approximations using 3, 4, or 6 decimals the number of slices s would be 10, 14, or 20, respectively. In this case, the BSI size formula can be simplified as:

$$IndexSize = m \times s \times n \tag{2}$$

As long as the number of slices used is less than the computer word size, BSI will always require less storage than the original data. The sorted lists, on the contrary require at least double the storage space than the original data. The reason is that each list stores the attribute value and the tuple id. BSI uses the bit position as the tuple id and since the slices are not sorted, the same position in all the slices always refer to the same tuple.

Note that the number of slices used to represent the attributes does not bound the number of slices to represent the BSI summation of these attributes. The resulting sum BSI is expected to have more slices, and since slices are added as bitVectors, there is no overflow in the computation.

4.2 Query Execution

The top-k preference queries are processed using bit-wise operations over the bit-slices. If the bit operations are performed on a computer with a 64-bit architecture, then the number of bit operations that we need to perform to operate two bitmaps is given by $n/64$. In general, with a word size equal to w-bits, each individual bit-wise operation would process w tuples simultaneously. The total cost of the query execution using bit-sliced indices is determined by the number of bitmaps that are operated together and the number of bit-wise operations performed.

Top-k Weighted Preference Queries. Algorithm 1 performs the generic top-k weighted preference query.The total cost of the preference query processing defined in Algorithm 1 is given by:

$$O\left(|q|bs\frac{n}{w}\right) \tag{3}$$

where $|q|$ is the number of non-zero preferences in the query (m in the worst case), b is the number of bits used to represent the preferences for each attribute query weight, s is the number of slices used to represent each attribute, and n is the number of tuples in the data set.

Algorithm 1 calls the Multiply Algorithm $|q|$ times, the SUM_BSI Algorithm $|q| - 1$ times, and the top-k Algorithm once.

The cost of Algorithm 2 in the worst case can be expressed in terms of BSI bit-wise operations $O(bs)$, where b is the number of bits used to represent the query preference and s is the number of slices used to represent each attribute. To represent preferences with 1 or 3 decimals, the number of slices would be 4 or 10, respectively. We can expect the number of set bits in each preference to be half the bits (on average). The cost of each bit-wise operation is $O(n/w)$, which allow us to express the cost of the multiply algorithm as:

$$O\left(bs\frac{n}{w}\right) \tag{4}$$

The number of slices in the result R is at most $s + p$ slices. Note that the number of slices per attribute depends on the value range of the attribute over the entire data set. In the cases where less slices are needed than the number of bits required to store the attribute data type, then our index is guaranteed to be smaller than the raw data. This is the case for most real data sets, which means that the BSI index size is usually smaller than the raw data even without applying compression.

The cost of Algorithm 3 is dependent on the maximum number of slices between A and B. The number of slices in B is (in the worst case) $p + s$, while the number of slices in A (the partial score computed so far) is $p + s + \lceil \log_2 m \rceil$ (in the worst case). The cost of computing the sum BSI can then be expressed as:

$$O\left((b + s + \lceil log_2 m \rceil)\frac{n}{w}\right) \tag{5}$$

Finally, the cost of computing the top-k scores is given by the number of slices in S $(p + s + \lceil \log_2 m \rceil\ )$:

$$O\left((b + s + \lceil log_2 m \rceil)\ \frac{n}{w}\right) \tag{6}$$

Equation 3 represents the upper bound for equations 4, 5, 6, and thus gives the upper bound on the total running time for the generic top-k weighted preference query.

Note that the top-k algorithm 4 is the only algorithm that is affected by the value of k. On the contrary, TA maintains a list of k elements. As k grows, the pruning benefit of TA and its variants decreases, and these algorithms become worst than sequential scan. Furthermore, since BSI computes and stores the scores for all the data points, it is suitable for interactive queries where users can increase the value of k to obtain immediate answers. It is also possible for the user to exclude data points from the result set as a bitArray. This bitArray can be used to initialized the E bitArray in the top-k Algorithm 4 to exclude those results.

Top-k Boolean Preference Queries. For top-k Boolean preference queries, Algorithm 2 does not need to be called, thus the cost of computing the top-k preferences is lower and it is given by:

$$O\left(|q|s\frac{n}{w}\right) \tag{7}$$

Top-k Baseline Preference Queries. In this case all the query weights are equal to one. The number of non-zero query weights equals to the number of data attributes: $|q| = m$. Again, in this case Algorithm 2 is not called. Thus the cost of top-k preference queries is:

$$O\left(ms\frac{n}{w}\right) \tag{8}$$

5 Experimental Evaluation

In this section we evaluate our approach for executing top-k queries over bit-sliced index structures. We first describe the experimental setup and the data sets used, which include a set of synthetic data sets as well as four real data sets. Then we show the benefits of using the BSI index structures in terms of compression ratios when compared to sorted lists. We then conduct a set of experiments, where we show the performance advantages of the BSI structures when compared to Sequential Top-k Algorithm (STA) [21], Threshold Algorithm (TA) [14], Best Position Algorithms (BPA, BPA2) [15], and Sequential Scan (SS). Because STA has shown to outperform TA and its optimizations, we compare BSI against STA in most of our experiments.

5.1 Experimental Setup

The synthetic data sets were generated using two different distributions: `uniform` and `zipf`. The cardinality of the generated data is 1,000 unless otherwise stated. The `zipf` distribution is representative for many real-world data sets, it is widely assumed to be ubiquitous for systems where objects grow in size or are fractured through competition [32]. These processes force the majority of objects to be small and very few to be large. Income distributions are one of the oldest exemplars first noted by Pareto [33] who considered their frequencies to be distributed as a power law. City sizes, firm sizes and word frequencies [32] have also been widely used to explore the relevance of such relations while more recently, interaction phenomena associated with networks (hub traffic volumes, social contacts

[34], [35]) also appear to mirror power-law like behavior. The zipf distribution generator uses a probability distribution of:

$$p(k, n, f) = \frac{1/k^f}{\sum_{i=1}^{n}(1/i^f)}$$

where n is the number of elements determined by cardinality, k is their rank, and the coefficient f creates an exponentially skewed distribution. We generated multiple data sets for f varying from 0 to 2. Further, we varied the number of rows, number of attributes as well as their cardinality to cover a large number of possible scenarios.

We also use four real data sets to support our results obtained with synthetic data:

- `coil2000`[1]. This data set used in the CoIL 2000 Challenge contains information on customers of an insurance company. Information about customers consists of 86 variables and includes product usage data and socio-demographic data derived from zip area codes. The data was supplied by the Dutch data mining company Sentient Machine Research and is based on a real world business problem. It contains over 9,000 descriptions of customers, including the information of whether or not they have a caravan insurance policy.
- `internet`[2] This data comes from a survey conducted by the Graphics and Visualization Unit at Georgia Tech October 10 to November 16, 1997. We use a subset of the data that provides general demographics of Internet users. It contains over $10,000$ rows and 72 attributes with categorical and numeric values.
- `kegg-metabolic`[3] This is the KEGG Metabolic Relation Network (Directed) data set. It is a graph data, where Substrate and Product componds are considered as Edges while enzyme and genes are placed as nodes. There are $53,414$ tuples in this data set, and 24 attributes, with real and integer values.
- `poker-hand`[4] In this data set each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes, plus one Class attribute that describes the "Poker Hand". The data set contains $1,025,010$ instances and 11 attributes with categorical and numeric values.

For the experiments we generated three types of queries:

- `Boolean queries:` Every attribute of the query has a Boolean value (0 or 1), meaning that from the user perspective an attribute can be relevant or non-relevant.

[1] http://archive.ics.uci.edu/ml/machine-learning-databases/tic-mld/
[2] $http://www.cc.gatech.edu/gvu/user_surveys/survey-1997-10$
[3] $http://archive.ics.uci.edu/ml/machine-learning-databases/00220/$
[4] $http://archive.ics.uci.edu/ml/datasets/Poker+Hand$

– `Baseline queries:` Here every attribute is equally important for the user and hence the query vector is an all-ones vector.
– `Weighted queries:` Every attribute of the query has a weight between 0 and 1. The query weights are applied to each attribute before applying the top-k algorithm.

All experiments were executed over a machine with a 64-bit Intel Core i7-2600 processor (8MB Cache, 3.20 GHz) and 8 GB of memory, running Windows 7 Enterprise. Our code was implemented in Java. During the measurements, the queries were executed six times, and the result for the first run was discarded to prevent Java's *just-in-time* compilation from skewing results. The times from the other five runs were averaged and reported.

Query preferences are one decimal queries and follow a uniform random distribution. Each query set had 1,000 queries (unless otherwise noted). The query times reported corresponds to the average query time per query.

5.2 Index Size

Minimizing the index size is very important. A smaller index size results in less disk accesses and less main memory requirements. Often the index space reduction translates into faster processing of the index.

Figure 3 shows the index size of the original/raw data (SS, STA), the sorted lists (TA), and BSI using three different number of slices per attribute. For this experiment we generated a data set with 10 million tuples and varying number of attributes.

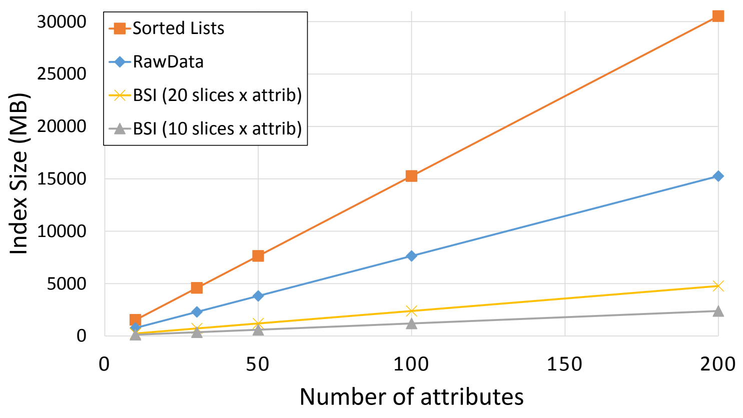


Fig. 3: Index size comparison for a data set with 10 million tuples and varying number of attributes. BSI is generated using 10 and 20 slices per attribute to represent 3 and 6 decimal numbers, respectively.

The storage requirement for the sorted lists is at least 2 times larger than the original data (assuming only the sorted lists are stored). The size further increases if the position list for each tuple is stored. In contrast, the total BSI size is smaller than the original data. If the normalized attribute values are represented using 3 decimal positions, then the number of slices used to represent each attribute is 10. In this case, for a 64-bit architecture, the BSI size is over 6 times smaller than the original data. When 6 decimals positions are used, 20 slices per attribute are generated but the index size is still one third the size of the original data.

5.3 BSI Performance Evaluation

In this section we evaluate our top-k query processing method using bit-sliced indices (BSI) by comparing it against the Sequential Top-k Algorithm (STA) [21], Threshold Algorithm (TA) [14], Best Position Algorithms (BPA, BPA2) [15], and Sequential Scan (SS). We designed a set of experiments that evaluate the usage of our approach for different types of data and queries.

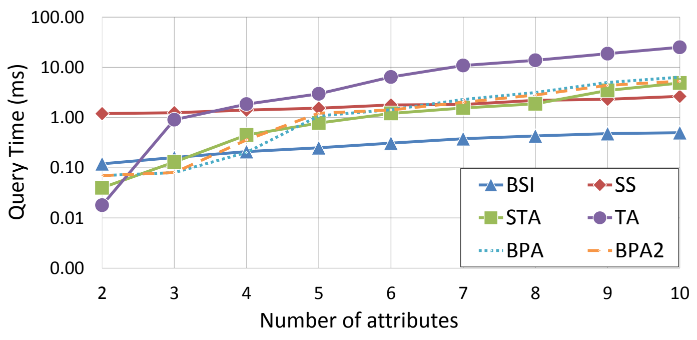
Query Time vs. Data Dimensionality. Most of the existing indexing structures have been evaluated and are known to perform efficiently for a small number of dimensions [26]. A more scalable approach is Sequential Top-k Algorithm [21], it has been proven to outperform TA and its variants even for a higher number of dimensions.

Figures 4a and 4b show the query times for top-k preferences queries over a synthetically generated data set, where we vary the number of attributes. Figure 4a shows the query times for STA, BPA, BPA2, TA, SS, and BSI over a small number of attributes: varying from 2 to 10. For this experiment we extract top-20 candidates, and the data set has a Zipf-1 distribution.

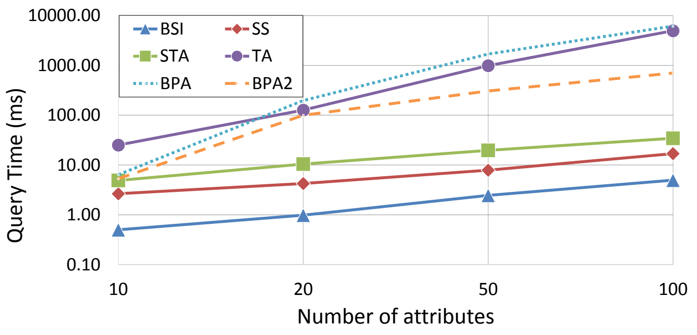
STA performs slightly better than BSI when the number of attributes is 3 or smaller. Then, as the number of attributes grows STA query times grow at a faster rate than the query times for BSI. Note that BSI is always faster than sequential scan. TA, BPA and BPA2 are less scalable and their query times are even slower than STA, with TA having a response time of 25 ms for 10 dimensions (aprox. 100 times slower than BSI!).

To go even further and demonstrate the scalability of our approach, we show in Figure 4b the query times for up to 100 attributes, for the same data set. As can be seen, as the dimensionality grows, STA, TA, and BPA are outperformed by SS, while BSI remains over 3 times faster than SS.

One of the main advantages for using BSI is that it executes sequential scan over bit-slices when performing Algorithms 2, 3 and 4. Thus, it only accesses one data attribute at a time, and makes very efficient use of available memory and cache. Furthermore, due to this property, our approach has the potential to partition the BSI index and run Algorithms 2 and 3 in parallel.



(a) Low-Dimensional Data



(b) High-Dimensional Data

Fig. 4: Query time for low and high-dimensional data. [Zipf-1, rows:100K, top-20]

Query Time vs. Number of Top Preferences Figure 5 shows how the value of k (the number of preferences) impacts the query time for the preference query. The measurements were taken using a synthetically generated data-set with 10 million rows, 20 attributes and *zipf-1* data distribution. The queries used have a 1-decimal precision. Because BSI and sequential scan compute the score for all tuples regardless of the value of k, the time for computing top-k preferences is not significantly affected when increasing k. However, the query time for STA increases with the increase in the value of k as it takes longer to meet the stopping the criteria.

Query Time vs. Number of Tuples. To show the scalability of our approach, we present in Figure 6 the query times for a synthetically generated data set with a Zipf-1 distribution. We vary the number of rows for this data set from 10 thousand to 10 million rows. The number of attributes is 20, and the data has been normalized with 3 decimal precision. For this experiment we used a set of 100 weighted, 1-decimal, queries to extract top-20 candidates. In the results we present the average time taken to process a single query.

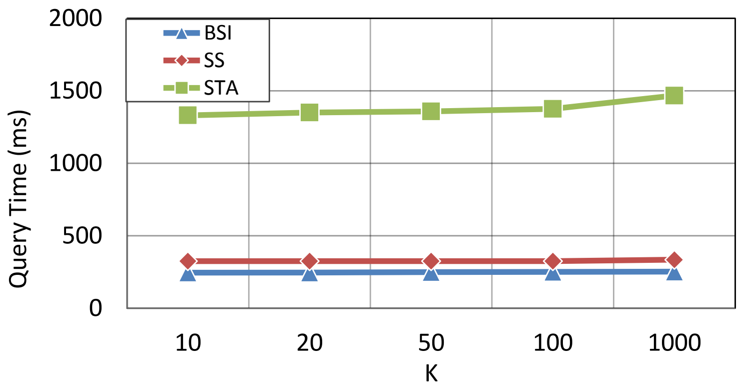


Fig. 5: Query times comparison when varying Top-k preferences. [Zipf-1, 20-attributes, 10M-rows]

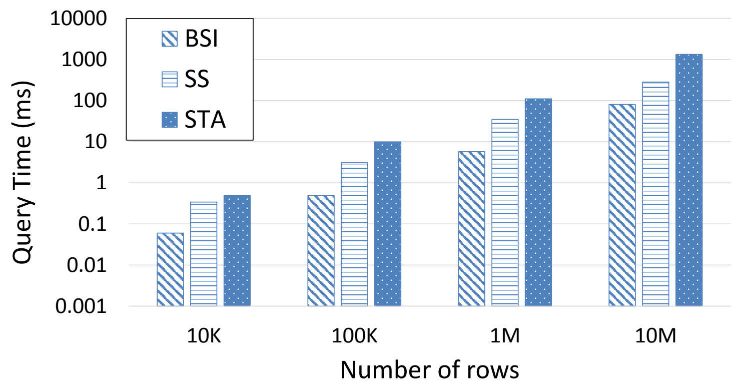


Fig. 6: Query times comparison when varying the number of rows. [Zipf-1, 20-attributes top-20]

As can be seen in Figure 6, not only BSI is faster than both STA and SS, but also the query time per row remains constant for BSI (aprox. 7 ns/1K rows). This is not the case for STA, where the query time per row increases by adding more rows (45 ns/1K rows for a 10K rows data set , and 115 ns/1K rows for a 10M rows data set). Generally, as the number of rows grows, STA takes longer to reach its stopping criteria and becomes slower.

Query Time vs. Data Skewness. To show the effect of data distribution over BSI query processing, we vary the distribution of a data set from a uniform (zipf-0) distribution to a Zipfian distribution with the skew factor $f = 2$ (zipf-

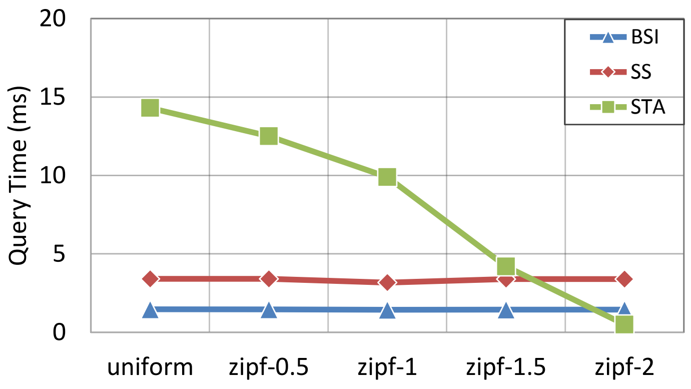


Fig. 7: Query time relative to data distribution. [100K-rows, 20-attributes, top-20]

2). The data set used in Figure 7 contains 100 thousand rows and 20 attributes with 10 bit-slices per attribute. The skewness of the data is towards the smaller values. For this experiment we use the same set of queries as in the previous one, for finding top-20 candidates.

As seen in Figure 7, BSI and Sequential Scan (SS) have constant processing times per query, irrespective of the data distribution, with BSI being more than twice as fast as SS. On the other hand, STA is highly sensitive to the data distribution and the query times are comparable to BSI only for highly skewed data. However, most real data sets have a skew factor lower than two. Zipf's Law states that the frequency of terms in a corpus conforms to a power law distribution where f is around 1 [36]. Also the same tendency has been observed in network graph data [34].

Query Time vs. Attribute Cardinality. The cardinality of the data represents an important aspect for the BSI index. The higher the cardinality, the more slices need to be created per attribute. Figure 8 shows the query times for a uniformly distributed, synthetically generated data set that varies the attribute cardinality from 10 to 1,000,000. The data set has 10 million rows and 5 attributes. We used a set of 100 weighted queries to extract top-20 candidates. We use 1-decimal queries.

Figure 8 shows that the query times for both, STA and BSI, are dependent on the cardinality of the data. Both perform similarly with the increase in cardinality. However, for normalized data that is up to 6 decimals, BSI is still faster than SS, while STA is more than 4 times slower than SS. For lower data cardinalities, BSI multiplication algorithm 2 performs fewer iterations and thus translate to faster query times.

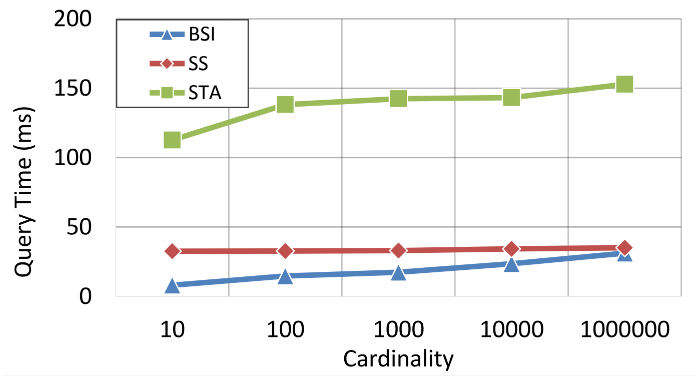


Fig. 8: Query time for varying data cardinality. [Uniform, 10M-rows, 5-attributes, top-20]

Query Time vs. Query Sparsity. Figure 9 compares the query time performance for SS, STA and BSI using sparse and non-sparse weighted queries. We vary the non-zero query attributes from 1% to 100%. For this measurement we use a synthetically generated data set with uniform distribution. It contains 100, 000 rows and 1, 000 attributes with 3 decimal precision.

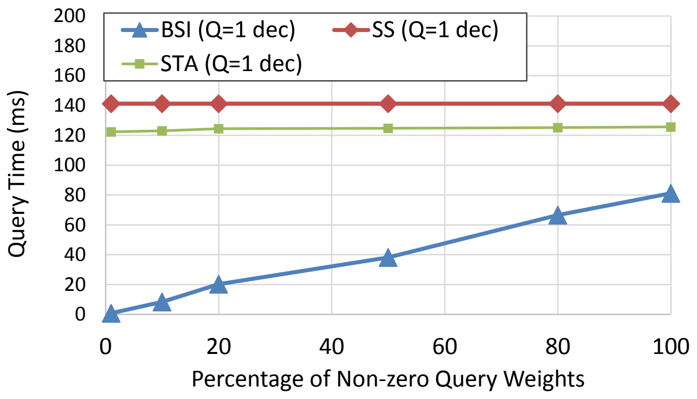


Fig. 9: Query time for sparse queries. [Uniform, 100K-rows, 1K-attributes, top-20]

Both, Sequential Scan (SS) and Sequential Top-k Algorithm (STA) scan sequentially the table for extracting top-k preferences. This means that regardless of the sparseness of the query, the query time will still be dominated by the scan of the table. On the other hand, TA and BSI are able to exploit the query sparseness, however TA is about two orders of magnitude slower than BSI. BSI query times increase only linearly-proportional with the number of non-zero query attributes added. This is mostly due to the fact that BSI does not need to load the entire index in memory when performing weight multiplication and finding top-k. Thus it works with smaller data structures and can also better exploit the availability of the main memory and the CPU cache. BSI Algorithms 2 and 3 are only invoked for non-zero query attributes. Hence, high-dimensional data is treated as low dimensional data when the queries are sparse.

Query Time vs. Query Slices. The results in Figure 10 are obtained using the same data set as in Figure 9. Here we look closer how BSI performs for different types of preference queries. More precisely, we vary the number of non-zero query attributes for Boolean queries, and for weighted queries with 1,2 and 3 decimal weights. The Boolean queries become all-ones queries(Baseline preference) for 100% non-zero boolean query weights. As the figure shows, by increasing the number of decimals for queries, the BSI Algorithm 2 will run slower, and this is reflected in the query times. However, for the queries with 50% or less non-zero attribute weights (500 attributes queried), BSI is still at least 50% faster than SS and about 25% faster than STA for 3-decimal queries.

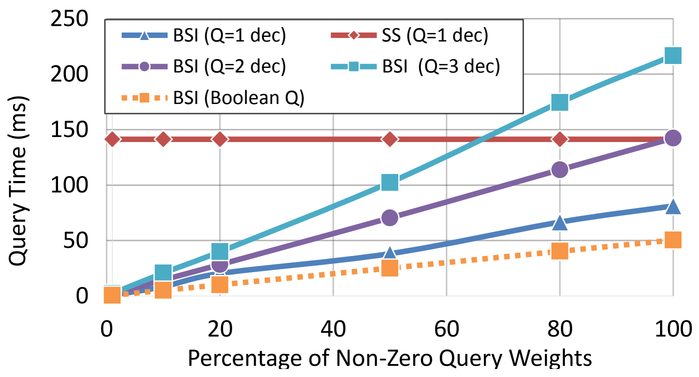
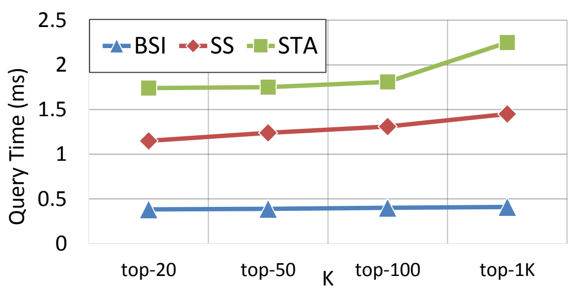


Fig. 10: Query time for different types of queries. [Uniform, 100K-rows, 1K-attributes, top-20]

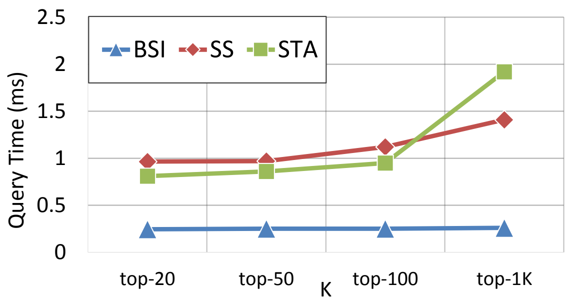
Results for Real Datasets. Figure 11 shows the query times for the four real data sets described in section 5.1 as the number of query results increases (increasing k). For querying these data sets we use a set of $1,000$ weighted queries, uniformly generated, with 1-decimal query weights.

As the figure shows, increasing k when extracting top-k candidates, does not impact significantly the query time for BSI. In fact, the query time increases by only 0.01 - 0.03 ms when changing $k = 10$ to $k = 1000$ for all four data sets.

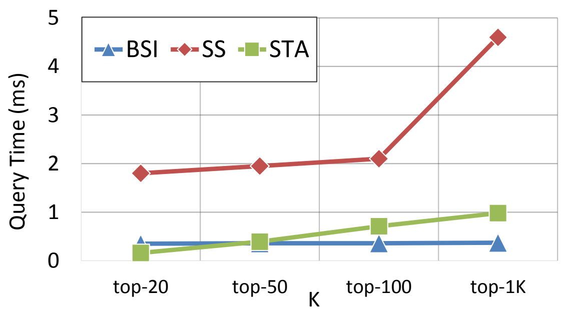
The reason is that the top K BSI algorithm is only invoked once after the scores for all the tuples have been computed. As expected, SS and STA performance increases linearly with k.



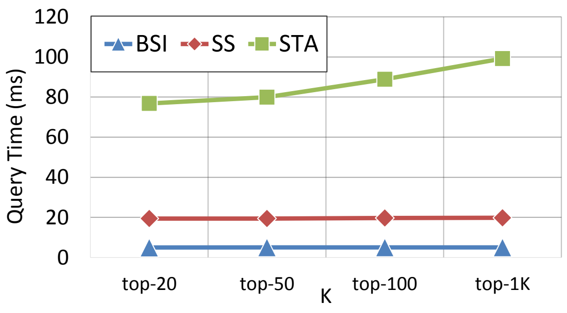
(a) Top-k for the `coil2000` data set.
Rows: 9822, Attributes: 86



(b) Top-k for the `internet` data set.
Rows: 10104, Attributes: 72



(c) Top-k for the `kegg-metabolic` data set.
Rows: 53413, Attributes: 24



(d) Top-k for the `poker-hand` data set.
Rows: 1,000,000; Attributes: 11

Fig. 11: Top-k weighted query on real data (K= 20 - 1,000)

6 Conclusion

We have introduced a novel algorithm to perform top-k and preferences queries using bit-sliced indices for high-dimensional data. With the proposed indexing technique, there is no need to maintain a list of sorted attributes and it is easy to combine top-k queries with other types of queries for which bitmap indices have been traditionally recognized for, such as point and range queries. BSI uses only the number of bit-slices required by data value-range, thus in general, the index size is always smaller than, the size of the raw data.

In addition since attributes are indexed independently, our techniques can take advantage of the columnar storage and have the potential to be executed in parallel. We introduce several algorithms for processing top-k, and top-k weighted queries while exploiting the fast bit-wise operations enabled by the BSI index. This approach is robust and scalable for high dimensional data. In our experimental evaluation we show that by increasing the dimensionality of the data, BSI query times increase only linearly-proportional to the number of attributes added. Moreover, the distribution of the data does not affect the query performance for BSI, while TA and other threshold algorithms using sorted lists are very sensitive to data distribution.

Since the index is not sorted, updates to the index are more efficient than when keeping a sorted list. Also, since all the bits are sliced it is possible to further control the precision of the result and the query time performance by shedding the least significant bits. Further investigations are required for this matter, and will be a future research direction. Another future work direction is the exploration of bitmap compression for top-k queries. This can further reduce the space requirement of the BSI indices and still allow for fast bit-wise logical operations.

7 Acknowledgments

We would like to thank reviewers for their insightful comments on the paper, as these comments led us to an improvement of the work.

References

[1] I. F. Ilyas, G. Beskales, M. A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Comput. Surv. 40 (4) (2008) 11:1–11:58. doi:10.1145/1391729.1391730.
URL `http://doi.acm.org/10.1145/1391729.1391730`

[2] M. Pagani, Encyclopedia of Multimedia Technology and Networking, 2nd Edition, Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2008.

[3] C. Böhm, S. Berchtold, D. A. Keim, Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases, ACM Comput. Surv. 33 (3) (2001) 322–373. doi:10.1145/502807.502809.
URL `http://doi.acm.org/10.1145/502807.502809`

[4] I. Daoudi, S. E. Ouatik, A. E. Kharraz, K. Idrissi, D. Aboutajdine, Vector approximation based indexing for high-dimensional multimedia databases (2008).

[5] S. Chaudhuri, L. Gravano, A. Marian, Optimizing top-k selection queries over multimedia repositories, IEEE Trans. on Knowl. and Data Eng. 16 (8) (2004) 992–1009. doi:10.1109/TKDE.2004.30.
URL `http://dx.doi.org/10.1109/TKDE.2004.30`

[6] R. Fagin, Combining fuzzy information from multiple systems (extended abstract), in: Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '96, ACM, New York, NY, USA, 1996, pp. 216–226. doi:10.1145/237661.237715.
URL `http://doi.acm.org/10.1145/237661.237715`

[7] X. Long, T. Suel, Optimized query execution in large search engines with global page ordering, in: Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '03, VLDB Endowment, 2003, pp. 129–140.
URL `http://dl.acm.org/citation.cfm?id=1315451.1315464`

[8] M. Persin, J. Zobel, R. Sacks-davis, Filtered document retrieval with frequency-sorted indexes, Journal of the American Society for Information Science 47 (1996) 749–764.

[9] P. Cao, Z. Wang, Efficient top-k query calculation in distributed networks, in: Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, PODC '04, ACM, New York, NY, USA, 2004, pp. 206–215. doi:10.1145/1011767.1011798.
URL `http://doi.acm.org/10.1145/1011767.1011798`

[10] M. Wu, J. Xu, X. Tang, W.-C. Lee, Top-k monitoring in wireless sensor networks, IEEE Trans. on Knowl. and Data Eng. 19 (7) (2007) 962–976. doi:10.1109/TKDE.2007.1038.
URL `http://dx.doi.org/10.1109/TKDE.2007.1038`

[11] W.-T. Balke, W. Nejdl, W. Siberski, U. Thaden, Progressive distributed top-k retrieval in peer-to-peer networks, in: Proceedings of the 21st International Conference on Data Engineering, ICDE '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 174–185. doi:10.1109/ICDE.2005.115.
URL `http://dx.doi.org/10.1109/ICDE.2005.115`

[12] A. Metwally, D. Agrawal, A. E. Abbadi, An integrated efficient solution for computing frequent and top-k elements in data streams, ACM Trans. Database Syst. 31 (3) (2006) 1095–1133. doi:10.1145/1166074.1166084.
URL `http://doi.acm.org/10.1145/1166074.1166084`

[13] A. Marian, N. Bruno, L. Gravano, Evaluating top-k queries over web-accessible databases, ACM Trans. Database Syst. 29 (2) (2004) 319–362. doi:10.1145/1005566.1005569.
URL `http://doi.acm.org/10.1145/1005566.1005569`

[14] R. Fagin, A. L. Y, M. N. Z, Optimal aggregation algorithms for middleware, in: In PODS, 2001, pp. 102–113.

[15] R. Akbarinia, E. Pacitti, P. Valduriez, Best position algorithms for top-k queries, in: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, VLDB Endowment, 2007, pp. 495–506.
URL `http://dl.acm.org/citation.cfm?id=1325851.1325909`

[16] A. Yu, P. K. Agarwal, J. Yang, Topk preferences in high dimensions (2014).

[17] P. Gurský, P. Vojtáš, Speeding up the nra algorithm, in: Proceedings of the 2Nd International Conference on Scalable Uncertainty Management, SUM '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 243–255.

doi:$10.1007/978-3-540-87993-0_20$.
URL `http://dx.doi.org/10.1007/978-3-540-87993-0_20`

[18] N. Mamoulis, K. H. Cheng, M. L. Yiu, D. W. Cheung, Efficient aggregation of ranked inputs, in: In ICDE, IEEE Computer Society, 2006, p. 72.

[19] A. Natsev, Y. chi Chang, J. R. Smith, C.-S. Li, J. S. Vitter, Supporting incremental join queries on ranked inputs, in: In VLDB, 2001, pp. 281–290.

[20] U. Güntzer, W.-T. Balke, W. Kießling, Optimizing multi-feature queries for image databases, 2000, pp. 419–428.

[21] W. Jin, J. M. Patel, Efficient and generic evaluation of ranked queries, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, ACM, New York, NY, USA, 2011, pp. 601–612. doi:10.1145/1989323.1989386.
URL `http://doi.acm.org/10.1145/1989323.1989386`

[22] P. O'Neil, D. Quass, Improved query performance with variant indexes, in: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, ACM Press, 1997, pp. 38–49. doi:http://doi.acm.org/10.1145/253260.253268.

[23] D. Rinfret, P. O'Neil, E. O'Neil, Bit-sliced index arithmetic, SIGMOD Rec. 30 (2) (2001) 47–57. doi:http://doi.acm.org/10.1145/376284.375669.

[24] M.-C. Wu, A. P. Buchmann, Encoded bitmap indexing for data warehouses, in: ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1998, pp. 220–230.

[25] R. Fagin, R. Kumar, D. Sivakumar, Comparing top k lists, in: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '03, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003, pp. 28–36.
URL `http://dl.acm.org/citation.cfm?id=644108.644113`

[26] H. Pang, X. Ding, B. Zheng, Efficient processing of exact top-k queries over disk-resident sorted lists, The VLDB Journal 19 (3) (2010) 437–456. doi:$10.1007/s00778-009-0174-x$.
URL `http://dx.doi.org/10.1007/s00778-009-0174-x`

[27] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum, Io-top-k: Index-access optimized top-k query processing, in: Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB '06, VLDB Endowment, 2006, pp. 475–486.
URL `http://dl.acm.org/citation.cfm?id=1182635.1164169`

[28] P. Gurský, P. Vojtáš, On top-k search with no random access using small memory, in: Proceedings of the 12th East European Conference on Advances in Databases and Information Systems, ADBIS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 97–111. doi:$10.1007/978-3-540-85713-6_8$.
URL `http://dx.doi.org/10.1007/978-3-540-85713-6_8`

[29] K. C. chuan Chang, S. won Hwang, Minimal probing: Supporting expensive predicates for top-k queries, in: In SIGMOD, 2002, pp. 346–357.

[30] G. Das, D. Gunopulos, N. Koudas, D. Tsirogiannis, Answering top-k queries using views, in: Proceedings of the 32Nd International Conference on Very

Large Data Bases, VLDB '06, VLDB Endowment, 2006, pp. 451–462.
URL http://dl.acm.org/citation.cfm?id=1182635.1164167

[31] G. Cong, C. S. Jensen, D. Wu, Efficient retrieval of the top-k most relevant spatial web objects.

[32] A. Clauset, C. R. Shalizi, M. E. J. Newman, Power-law distributions in empirical data, 2009. doi:10.1137/ 070710111.
URL http://dx.doi.org/10.1137/070710111

[33] V. Pareto, Manual of political economy (1906).

[34] A. lászló Barabási, R. Albert, Emergence of scaling in random networks, Science.

[35] A.-L. Barabasi, The origin of bursts and heavy tails in human dynamics, Nature 435 (2005) 207.
URL http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0505371

[36] G. Zipf, Human behaviour and the principle of least-effort, Addison-Wesley, Cambridge, MA, 1949.
URL http://publication.wilsonwong.me/load.php?id=233281783