Fast High-Dimensional Data Search in Incomplete Databases

Beng Chin Ooi Cheng Hian Goh Kian-Lee Tan
Dept. of Information Systems & Computer Science
National University of Singapore
Lower Kent Ridge, Singapore 119260
Email: {ooibc,gohch,tankl}@iscs.nus.edu.sg

Abstract

We propose and evaluate two indexing schemes for improving the efficiency of data retrieval in high-dimensional databases that are *incomplete*. These schemes are novel in that the search keys may contain missing attribute values. The first is a multi-dimensional index structure, called the Bitstring-augmented R-tree (BR-tree), whereas the second comprises a family of multiple one-dimensional one-attribute (MOSAIC) indexes. Our results show that both schemes can be superior over exhaustive search. Experimental results suggest that BR-trees have lower update and storage costs and are able to support range queries more efficiently under most circumstances, when compared to the MOSAIC indexing scheme. However, contrary to conventional wisdom, the MOSAIC structure outperforms the BR-tree in retrieval time for point queries, as well as in range queries over incomplete databases for *dimension-unrestricted* data distributions.

1 Introduction

We examine the problem of high-dimensional data search in incomplete databases. The widespread adoption of database technologies in various advanced applications (such as multimedia and medical systems) has given rise to the pressing need for efficient access methods for supporting data retrieval on multi-attribute (*high-dimensional*) search keys. Traditionally, this is accomplished via the use of *multi-dimensional indexes* (see [4] for a comprehensive survey). Unfortunately, these index structures, do not perform well when the database is *incomplete*: a scenario characterized by missing attribute values in some tuples of the database. This performance degradation is so severe that even an exhaustive search of the database would have yielded better performances.

In this paper, we address the issues pertaining to the design of fast mechanisms that avoid the costly alternative of performing an exhaustive search. A taxonomy of different index strategies is presented, and two representative index structures are singled-out and evaluated in a series of experimental studies. The first structure is a multidimensional index structure, called the *Bitstring-augmented R-tree (BR-tree)*, and the second structure is a family of multiple single-dimensional indexes, called the *MOSAIC structure*. Our results show that the proposed schemes are effective in reducing the search time for a wide range of queries as compared to exhaustive search. The BR-tree is shown to be more efficient in supporting range queries and have lower insertion and storage costs compared to the MOSAIC structure. However, contrary to conventional wisdom, the MOSAIC structure outperforms the BR-tree in point queries and also range queries when the underlying data distribution is *dimension unrestricted*.

The rest of this paper is organized as follows. In the next section, we provide a concise characterization of the problem addressed in this paper. We submit that the problem is much more prevalent than commonly believed, and review related work. Section 3 describes the framework and the proposed indexing structures.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

In Section 4, we present details of an experimental study, and the findings. Finally, we conclude in Section 5 with directions for future work.

2 High Dimensional Data Search in Incomplete Databases

2.1 Motivation

We believe that the problem of high dimensional data search in incomplete databases is becoming more prevalent, and deserves attention from the database community. Over the past decade, we have witnessed an increasing trend whereby database technologies are adopted for novel and advanced applications with complex data types. Such applications are characterized by three features. First, the data sets are usually high-dimensional. For example, in multimedia databases, feature vectors such as color histograms and shape descriptors extracted from the multimedia objects are usually mapped into points in a high-dimensional feature space. Other applications adopting similar approaches include CAD applications [15], molecular biology databases [1, 20] and Geographic Information Systems.

Second, such applications frequently involve *high-dimensional* search operations, i.e., data retrieval requiring restrictions on several attributes simultaneously. For example, similarity queries in multimedia applications often require comparisons across multiple features of objects stored in a database [5, 8, 12]; this is often translated to search operations on a multi-dimensional index that is used for organizing the corresponding feature space. As pointed out in [3], the number of dimensions is likely to remain high for a large class of applications, even if we should reduce the dimensionality of data for retrieval.

Third, most of the new applications are very large and may consist of several thousand attributes for each tuple. It is therefore not uncommon to find missing data occurring in some of the attributes. For example, Table 1 shows the distribution of missing attributes in a sample thyroid disease database [9]. As can be seen, almost all the tuples have at least one attribute value whose value is unknown. Out of over 31000 tuples with 28 dimensions in the database, only 20 tuples have no missing information; instead, more than 21000 tuples have one attribute with missing information, and as many as 1000 tuples have six attributes with missing information.

Although high-dimensional data retrieval has been extensively investigated in the literature, high-dimensional data retrieval in *incomplete* databases has not received much attention, probably because the problem is less likely to arise in conventional database systems where multi-dimensional searches are less common. In a conventional setting, missing values can be dealt with using a *fragmentation* strategy [10]. For example, in designing an employee database, some employees may have a home telephone while others do not; the missing data can be avoided by partitioning the database into two – one for employee data (without phone number) while the other for the phone numbers.

No. of missing attribute values	**No. of tuples**
0	20
1	21639
2	5980
3	2178
4	213
5	640
6	1110
7	24
8-28	0
Total	31804

Table 1: Distribution of missing information in a thyroid disease database.

Recently, different aspects of incomplete database (information) have also appeared in the literature. In [13], the incompleteness of a database is defined with respect to missing tuples, rather than attribute information. In [7], the concept of incompleteness is used in a data cube. Work on incomplete databases, as in having missing or unknown attribute values, have largely focused on the semantics of missing information [10].

On the other hand, the design of indexes to support high-dimensional data search is an area of active research. Work on high-dimensional data search have focused on designing efficient high-dimensional indexing structures [4]. These structures include the skd-tree [18], grid file [17], SR-tree [12], R-tree [11], R*-tree [2], R^+-tree [19], TV-tree [14] and X-tree [3]. An alternative paradigm is to make use of multiple single-attribute indexes to facilitate multi-dimensional search (see, for example, [16]). The database folklore, however, have maintained that multi-dimensional index is more efficient compared to having a plethora of single-dimensional indexes. As we will demonstrate later, this turns out to be untrue in the context of incomplete databases.

2.2 Problem Definition

Let D be a database with a schema of the form $(X_1, X_2, \ldots, X_n)$. This database D is said to be *incomplete* if tuples in it are allowed to have missing attribute values, either because the values are not known at the time the data are being captured, or because the corresponding attributes are not relevant for the tuple at hand [10]. The exact semantics of the missing values are irrelevant to our discussion and we will denote missing information with the symbol '?'.

We assume that data retrieval is based on a k-dimensional *search key*, where $k \leq n$. For simplicity and without any loss of generality, we assume that the k dimensions of the search key are the first k attributes in the schema, the values of which are drawn from the set of non-negative integers. We refer to a query such as this as a *high-dimensional query*. In general, the search key associated with a query takes the form

$$([x_1^l,\ x_1^h],\ [x_2^l,\ x_2^h],\ \ldots,\ [x_k^l,\ x_k^h])$$

where $x_i^l \leq x_i^h\ \forall\ i \in \{1, \ldots, k\}$. The values x_i^l and x_i^h are the lower- and upper-bound respectively of the i-th attribute. This query is said to be a *point query* if $x_i^l = x_i^h$ for all $i \in \{1, \ldots, k\}$. Otherwise, it is a *range query*. Throughout this paper, we assume that query ranges are *well defined*, i.e., missing values are not allowed as part of the search key.

Suppose we are given a range query Q where the k-dimensional search key is $([x_1^l,\ x_1^h],\ [x_2^l,\ x_2^h],\ \ldots,\ [x_k^l,\ x_k^h])$. A tuple $t = (y_1, \ldots, y_k, \ldots, y_n)$ in the database is said to be an *answer* for Q if every y_i $(i \in \{1, \ldots, k\})$ that is not a missing value falls in the corresponding range defined in the query, i.e., $x_i^l \leq y_i \leq x_i^h$. For example, if the query search key is given by ([1,3],[4,7],[2,9]), then both tuples $(1, ?, 2, \ldots)$ and $(?, ?, 9, \ldots)$ are said to be answers to the query. It is not hard to see that the above definition of an answer extends trivially to point queries.

The simplest approach to extending a multi-dimensional index for use in incomplete databases is to treat missing values in the database as "distinguished". Thus, given a tuple $t(x_1, \ldots, x_k, \ldots, x_n)$, this can be mapped to a k-dimensional coordinate $(f(x_1), \ldots, f(x_k))$ where

$$f(x_i) = \begin{cases} x_i & \text{if } x_i \text{ is known} \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

For example, if $n = 4, k = 3$, then the tuple (1,?,2,3) will be mapped to the coordinate (1,-1,2). The resulting k-dimensional space can now be indexed using a multi-dimensional index (e.g., R-tree) as in a complete database. Intuitively, this approach merely extends the domain of attributes in the search key with the distinguished value -1, and maps all missing (unknown) values to this symbol.

With this scheme, a query based on the search key will have to be replaced by 2^k subqueries. This is easily verified as follows: consider a point query that requests for all answers that match the search key $(x_1, \ldots, x_k)$. For each attribute value y_i $(i \leq k)$ of a matching tuple $(y_1, \ldots, y_k, \ldots, y_n)$, either $y_i = x_i$ or the y_i-value is missing, i.e., each y_i value can take one of two values (x_i or -1). This gives rise to a total of 2^k variations for a search key of length k. Similarly, any range query can be decomposed to 2^k subqueries (each of which is a range query).

The problem with the approach described above lies in the observation that *all unknown values along a given dimension are now mapped to a single orthogonal hyperplane.* If the proportion of missing values in the database is high, this will result in a highly skewed data set. As reported in [19], a highly skewed data set (such as one produced under this mapping) gives rise to poor performances when used with traditional multi-dimensional indexes (e.g., the R-tree). Our goal in this paper is to examine alternative indexing strategies that will improve the efficiency of high-dimensional search in incomplete databases. To the best of our knowledge, this problem has not been discussed elsewhere and our contribution here will be the first to a novel and important problem.

3 High Dimensional Data Search Techniques

To provide us with greater insight into strategies for indexing high-dimensional data, we have identified a framework composing of four generic strategies derived from two orthogonal choice-sets. The first requires a decision on whether or not tuples with missing values should be indexed separately from those which do not have missing values. The second deals with choices concerning the nature of the index structures.

The first decision gives rise to two categories of strategies, which we refer to as *partitioned* versus *non-partitioned*. In the case of a *non-partitioned* indexing strategy, the index(es) is (are) built on the entire database. On the other hand, a *partitioned* strategy would have split the data into two partitions: the *missing information group* which contains tuples with some missing information, and the *full information group* which contains tuples whose attribute values are all well-defined. Note that this partitioning is conceptual and pertains only to how data are being indexed. Hence, both tuples from the missing information group, and the full information group can be stored in the same data page. The difference is that a partitioned strategy creates two groups of indexes, one for the missing information group and another for the full information group.

The second decision deals with the dimensionality of the indexes. On one extreme, we can choose to construct a single multi-dimensional index for a k-dimensional search key. On the other hand, we can construct a single-attribute index for each dimension, i.e., there will be k one-dimensional indexes. Techniques for efficient merger of partial results from single-attribute indexes have been described in greater details in [16].

Based on the two choice sets, we can derive different strategies for high-dimensional data search on a k-dimension search key. The remaining discussion will be focused on non-partitioned strategies, since the observations can be easily generalized for partitioned approaches.

3.1 The Bitstring-augmented Multi-dimensional Index

As we have noted earlier in Section 2, an incomplete database can be indexed in a brute-force manner by replacing all missing information ('?') with a distinguished value (-1). However, this results in a highly-skewed data set that performs poorly using conventional multi-dimensional indexes such as the R-tree. To circumvent this problem, we introduce a novel mapping function that "randomly" scatters the points in the k-dimensional space defined by the search key, in an attempt to reduce this data skew. Note that the specific function described below is merely one of many possible forms.

Let $(x_1, \ldots, x_k)$ be the search key corresponding to a tuple t. We introduce a bit string $y_1 \cdots y_k$ as follows:

$$y_i = \begin{cases} 1 & \text{if } x_i \text{ is known} \\ 0 & \text{otherwise} \end{cases}$$

We define a mapping f on the search key $(x_1, \ldots, x_k)$ as follows:

$$f(x_i) = \begin{cases} x_i & \text{if } x_i \text{ is known} \\ \left\lceil \frac{\sum_{j=1}^{k} y_j \cdot x_i}{\sum_{j=1}^{k} y_j} \right\rceil & \text{if } x_i \text{ is missing and } \sum_{j=1}^{k} y_j \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Notice that it is possible for two distinct search keys to be mapped to the same point in the k-dimensional space under this mapping. For example, both the search keys (2,4,6) and (2,?,6) will be mapped to (2,4,6). To distinguish between two tuples that have different search keys but are mapped to the same value under the mapping function f, we associate with each index entry the bitstring $y_1 \cdots y_k$ where y_i is as defined earlier. The complete search keys corresponding to the above examples are therefore given by $<(2,4,6), 111>$ and $<(2,4,6), 101>$ respectively. We shall refer to the resultant multi-dimensional index structure as the *bitstring-augmented multi-dimensional index.*

As before, a query on the multi-dimensional database must be decomposed into a corresponding set of 2^k subqueries before it can be evaluated on the bitstring-augmented multi-dimensional index. For ease of exposition, we distinguish between point queries and range queries below.

A point query can be processed as a set of 2^k point subqueries as follows. First, the set of subqueries is generated by systematically identifying all the tuples (which may or may not contain missing values) that satisfy the query given. For example, if the search key given by a query is (1,2), this will match tuples with keys (1,2), (?,2), (1,?), and (?,?), suggesting that the original query should be transformed to $<(1,2), 11>$, $<(2,2), 01>$, $<(1,1), 11>$, and $<(0,0), 00>$ respectively. Each of the latter subqueries can now be evaluated as a point query, i.e., by probing the index structure for a search key with the same value.

For range queries, the query transformation process is identical except for the way the subqueries are generated. Consider a range query with a range-restricted search key:

$$([X_1, Y_1], [X_2, Y_2], \ldots, [X_k, Y_k])$$

As before, the original query needs to be decomposed into 2^k subqueries corresponding to all possible permutations of the bitstring $y_1 \cdots y_k$. This transformation is described in the algorithm below:

Algorithm Rewrite$(Q = [X_1, Y_1], \ldots, [X_k, Y_k])$
For each permutation $y_1 \cdots y_k$ construct the subquery $([X'_1, Y'_1], \ldots, [X'_k, Y'_k], y_1 y_2 \cdots y_k)$ where

$$X'_i = \begin{cases} X_i & \text{if } y_i = 1 \\ \left\lceil \frac{\sum_{j=1}^{k} y_j \cdot X_j}{\sum_{j=1}^{k} y_j} \right\rceil & \text{if } y_i = 0 \text{ and } \sum_{j=1}^{k} y_j \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$Y'_i = \begin{cases} Y_i & \text{if } y_i = 1 \\ \left\lceil \frac{\sum_{j=1}^{k} y_j \cdot Y_j}{\sum_{j=1}^{k} y_j} \right\rceil & \text{if } y_i = 0 \text{ and } \sum_{j=1}^{k} y_j \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

The subqueries are defined such that it retrieves all tuples (including those having missing values) that satisfy the query. As an illustration, we show below the subqueries generated for a range query with search key ([1,3],[5,7],[9,11]):

row	$y_1 y_2 y_3$	matching tuples	subquery search key
1	000	(?,?,?)	(0,0,0)
2	001	(?,?,[9,11])	([9,11],[9,11],[9,11])
3	010	(?,[5,7],?)	([5,7],[5,7],[5,7])
4	011	(?,[5,7],[9,11])	([7,9],[5,7],[9,11])
5	100	([1,3],?,?)	([1,3],[1,3],[1,3])
6	101	([1,3],?,[9,11])	([1,3],[5,7],[9,11])
7	110	([1,3],[5,7],?)	([1,3],[5,7],[3,5])
8	111	([1,3],[5,7],[9,11])	([1,3],[5,7],[9,11])

One can easily observe that the subqueries are constructed such that queries looking for tuples with missing values look only in a "tight" range as implied by the mapping function. For example, for tuples in row 7, we need only to look for points in the region ([1,3],[5,7],[3,5]) rather than the entire hyperplane defined by ([1,3],[5,7],[0,∞]). A pictorial representation of the search space is illustrated in Figure 1. As we will demonstrate later in Section 4, this gives rise to impressive performance of the bitstring-augmented multi-dimensional index for supporting range queries. We note however that other tuples not satisfying the original query may be mapped to the region. For example, the tuple with search key (2,6,4) does not satisfy the query (which is [1,3],[5,7],[9,11]) but will be retrieved by the subquery in row 7. This implies that the search procedure must filter these *false matches* by examining the bitstring. In effect, the search procedure is identical to that of the R-tree [11] except that additional filtering step is needed when examining the leaf nodes.

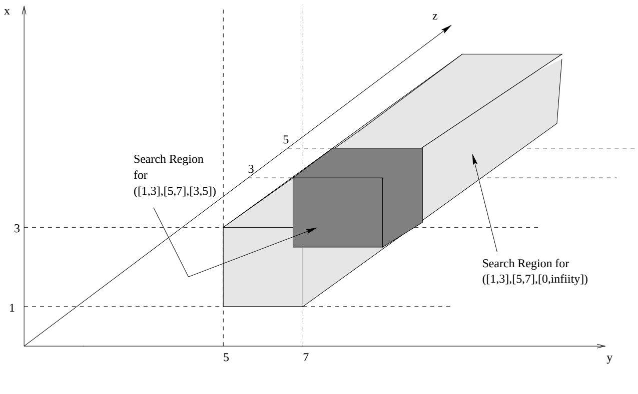


Figure 1: A reduced search space using bitstring-augmented multidimensional index.

To verify that the proposed mapping does improve the efficiency of search operations in incomplete multi-dimensional databases, we conducted a preliminary experimental study on the relative performance between the two mapping schemes, using the R-tree as the underlying multi-dimensional index structure. The experiment is conducted on a database size of 100K tuples with 8-dimension search keys. We vary the percentage of coverage in range queries from 0 to 5% of the domain space. We denote the scheme under Equation 1 as Single Value Transformation (SVT) and that under Equation 2 as Bitstring-augmented Transformation (BAT). As comparison, we also included the cost for sequentially scanning the entire input file (denoted as SCAN). The result is shown in Figure 2. From the result, it is clear that SVT is very much worse than BAT because of the skew data points as a result of using Equation 1. It also shows that BAT can be very effective in randomizing the points in a multidimensional space. Because scheme SVT performs poorly, for the rest of this paper, we shall focus on using the BAT scheme when multi-dimensional indexes are used.

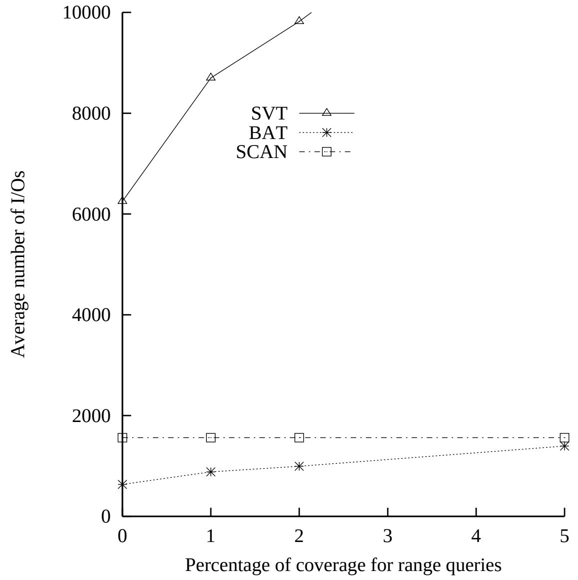


Figure 2: Comparison of SVT, BAT and SCAN.

3.2 Multiple One-attribute One-dimension Indexes

In this category, a one-dimensional index is built on each attribute (dimension) of the search key. This is essentially an *inverted index* that allows for rapid identification of the set of tuples having a given value in the dimension being indexed. Thus, there will be as many inverted indexes as there are dimensions in the search key. Under this scheme, the transformation under Equation 1 suffices since every value in one single-attribute index is either a null or a non-null.

It turns out that processing point and range queries are very straightforward with this method. Consider a range (point being a special case) query, say ($[X_1, Y_1]$, $[X_2, Y_2]$, ..., $[X_k, Y_k]$). Then for any arbitrary dimension, say i, the set of tuples that can contribute to the final result are those indexed by values in the range $[X_i, Y_i]$ and those indexed by -1. This

Table 2: Parameters and their values.

Parameter	Default Values	Variations
System Parameters		
page size	4K page	
index node size	4K page	
buffer size	128 pages	
Database Parameters		
no. of tuples	1 million	100,000
no. of dimensions	8	2,4,16
domain of dimensions	[1..100000000]	
distribution of missing attribute values	Dimension-restricted	Unrestricted
Query Parameters		
query type	point	range — 1%,2%,5%
no. of queries/query type	1000	

means that a total of only $2k$ subqueries need to be constructed; this number is significantly smaller than the 2^k subqueries that are needed if the bitstring-augmented multi-dimensional index had been used. Suppose the results obtained from a subquery i is given by Res_i. The final result (satisfying the original query) is given by the intersection of all candidate tuples in all dimensions, i.e., $Res_1 \cap Res_2 \cap \ldots Res_k$.

Compared with a single k-dimensional index, this approach is clearly less space efficient. In fact, it has been traditionally recognized that this approach may not be efficient (because of union and intersection operations) for high-dimensional data search as compared to a single high-dimensional index. However, this may no longer be true for incomplete databases, whereby the choice of a multi-dimensional index requires a query to be rewritten into a large number of subqueries. This suggests that having a family of multiple one-dimension single-attribute indexes may not be such a bad idea. As we shall see later in our experimental results, this turns out to be the case.

From the above discussion, we note that we can actually design a hybrid of the above two approaches. In most applications, we can expect some attributes to be constrained by a non-null clause during insertion. Let the number of such attributes in the search key be m, $m < k$. Thus, we can build a m-dimensional index on the attributes with no missing attribute values, and multiple one-dimensional indexes on each of the remaining dimensions.

4 A Performance Study

In this section, we present an experimental study on high dimensional data search in an incomplete database. In our study, we restrict our discussion to two mechanisms, and investigate their relative performance in terms of number of disk accesses. We also compare the two approaches with respect to their insertion cost and storage cost.

Among the possible schemes presented in Section 3, we pick the following two for further study:

1. **Single k-dimensional index.** We adapt the basic R-tree index [11] by introducing the bitstring information. Recall that we are using the transformation function given by Equation 2. We refer to this variation as the Bitstring-augmented R-tree, BR-tree in short.

2. **Multiple one-dimensional one-attribute indexes.** In this category, we implemented the B$^+$-tree as the underlying indexing mechanism. The "extra" unknown value (-1) for each dimension is treated separately. We shall refer to this structure as MOSAIC since *m*ultiple *o*ne-dimension *s*ingle-*a*ttribute *i*ndexes are to be used *c*ollectively to answer a query.

For purpose of comparison, we also include the cost for sequentially scanning the data file. This approach is denoted as SSCAN.

4.1 Experimental Setup

Several parameters are used in our experiments and these and their default settings are shown in Table 2.

The data set used in our experiment consists of Fourier points in a high-dimensional space for contours of industrial parts. The database is the same as that used in the experimental study for the X-tree [3] except that we restrict the size to no more than 1 million tuples. The domain of each dimension is in the range [1..100000000] (The original data set used in [3] contain floating points, which we converted to integers in our study.)

No. of missing attribute values	% of tuples
0	9/25
1	7/25
2	5/25
3	3/25
4	1/25

Table 3: Distribution of missing attribute values in 8-dimensional data set ($n = 4$).

We note that the original data set does not contain missing information. To model missing attribute values, we adopt the following process:

- Determine the number of dimensions that will contain missing values. Let us denote this by n. In our study, the number of missing attribute values per tuple is restricted to *at most* half of that of the number of dimensions in the search key, i.e., $1 \leq n \leq \lceil 0.5 \cdot k \rceil$.

- Determine the percentage of tuples in the database that will have i missing attribute values, $i = 0, 1, \ldots, n$. This is given by the following expression:

$$N(i) = \left(\frac{(n-i)}{n}\right)^2 - \left(\frac{(n-i-1)}{n}\right)^2$$

  The function $N(i)$ is chosen such that the percentage of i missing values is inversely proportional to i (i.e., a larger number of tuples having small number of missing values and vice versa). In addition, it also satisfies the property that $\sum_{j=0}^{n} N(j) = 1$.

As default, we employ the *dimension-restricted distribution method*, which is to predetermine the dimensions that will contain missing attribute values, i.e., only these predetermined dimensions shall contain missing attribute values while the other dimensions will be well defined. As pointed out earlier, this is not uncommon since most applications will require some attribute values to be non-null. For example, in our default setting of 8-dimensions, the percentages of tuples having correspondingly different number of missing attribute values are given by Table 3.

We note that under the dimension-restricted distribution, the size of the bitstring is also reduced to just n bits. Furthermore, the number of subqueries is also reduced to 2^n (as compared to 2^k).

For a million tuples, we can expect the indexes to be fairly large, and hence it is unlikely that the entirety of an index fits in memory. Instead, some index pages are paged out as we traverse the tree, and re-fetched at a later time when they are re-referenced. For simplicity, we employ the priority-based *least recently used* buffer replacement strategy [6]. As default, the buffer size is 128 pages, each page being 4K pages. The index nodes are also 4K pages.

In each experiment, 1000 queries are generated. The average number of I/Os is used as a metric for comparative study. For point queries, each point is randomly selected from the respective test data. Range queries are formed from point queries by enlarging the point such that the extent of each dimension covers a certain percentage of the domain space of that dimension. In our study, we restrict three range query coverages — 1%, 2% and 5%.

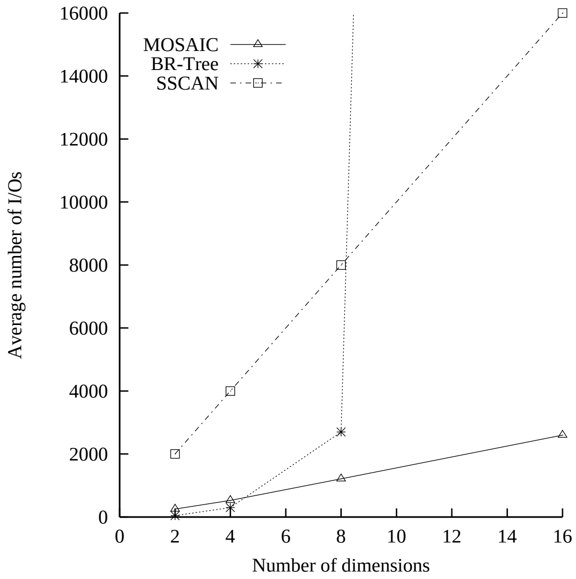
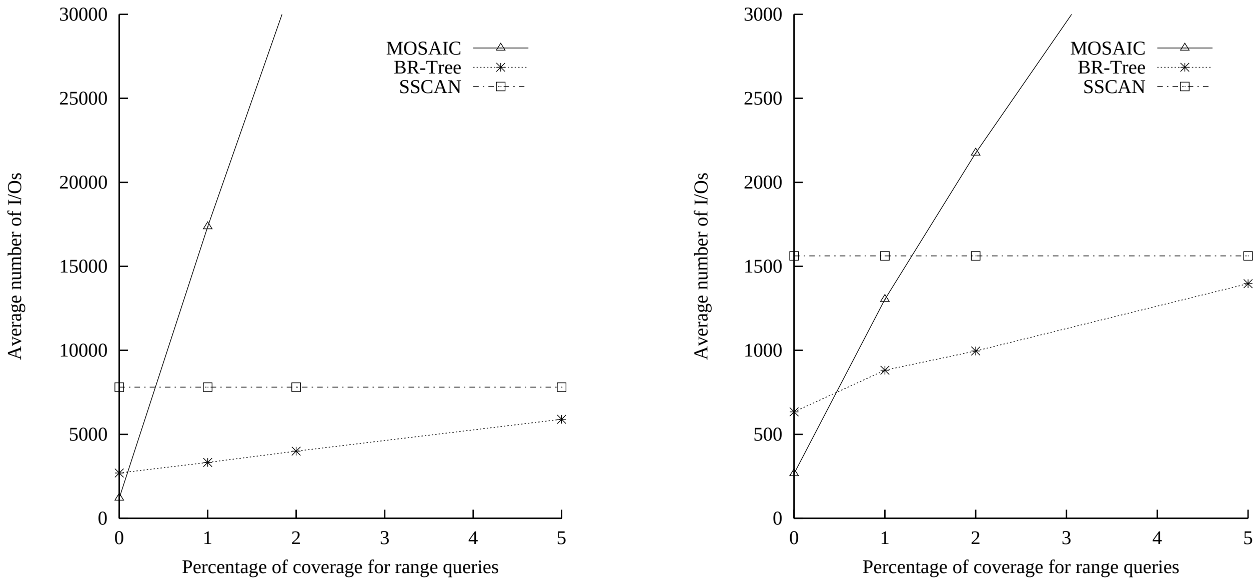


Figure 3: On point queries.

4.2 Experiment 1: Point Queries

In this set of experiment, we examine the performance of the two proposed schemes for point queries. Figure 3 shows the result for different data sets with different number of dimensions. From the result, we note that when the number of dimensions is small, the BR-tree performs well. However, as the number of dimensions increases, its performance degrades drastically. This is because of the large number of subqueries that have to be processed. In fact, the performance of BR-tree is worse than sequentially scanning the data file when the number of dimensions increases to 16. On the other hand, we note that the average number of I/Os for the MOSAIC structure is the least among the three methods studied. Furthermore, we note that the number of I/Os grows almost linearly with the number of dimensions of the index. Such a property makes it



(a) Relation size = 1M

(b) Relation size = 100K

Figure 4: On range queries.

a very promising mechanism for searching high dimensional data in incomplete databases.

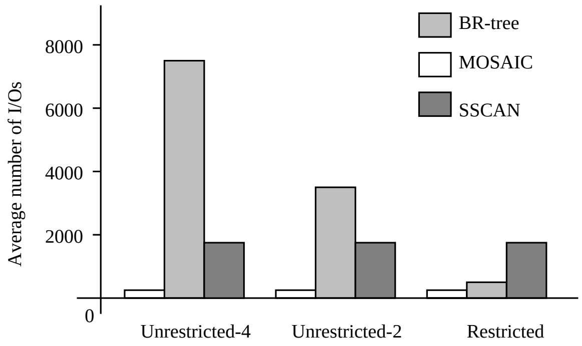
4.3 Experiment 2: Range Queries

In this set of experiments, we evaluate the performance of the two schemes for range queries under the default setting. We vary the range from 0 to 5% of the domain range. The result is shown in Figure 4(a). From the result, we see a different picture compared to the experiments on point queries. While MOSAIC performs well for point queries, it degenerates quickly for range queries, so much so that it performs worse than SSCAN. This is because in a range query, the number of potential tuples in a dimension that satisfy the final answer is fairly large. This result in many swapping in and out of pages when performing the union and intersection operations. On the other hand, for BR-tree, because all the dimensions are examined simultaneously, the search covers a smaller search space. The linear increment in performance as the percentage of coverage increases gives the BR-tree a competitive edge over the MOSAIC structure.

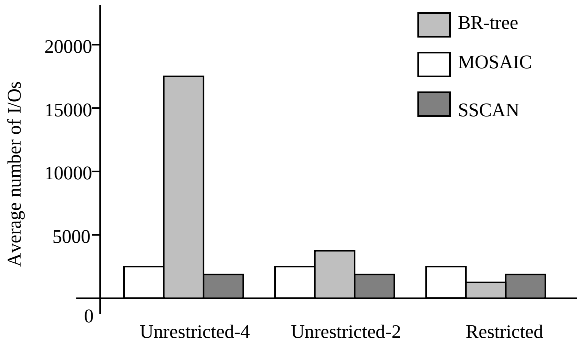
We also repeated the experiment for smaller size relations with 100K tuples. The result, shown in Figure 4(b) is largely similar to Figure 4(a) demonstrating that MOSAIC is effective for point queries while BR-tree is best for range queries.

4.4 Experiment 3: Effect of Data Distribution

We also study how different distributions of the missing attribute values can impact the performance of the two schemes. We relax the constraint of fixing the dimensions that should contain missing attribute values, i.e., missing attribute values can appear in any dimension of the search keys. We call this method the unrestricted distribution approach. As in the dimension-restricted distribution approach, the percentage of tuples with i missing attribute values is given by $((n-i)/n)^2 - ((n-i-1)/n)^2$ where n is the maximum number of missing attribute values per tuple. We denote the approach by *unrestricted-n*. We studied two such distributions for a database size of 100K tuples: unrestricted-4 and unrestricted-2. The result for point queries is shown in Figure 5(a), whereas the result for 2% range coverage is shown in Figure 5(b). As shown in the figure, we note that the MOSAIC structure is marginally affected by the distribution used. However, the performance of the BR-tree is very dependent on the distribution. This is because of the number of subqueries that have to be generated. When the distribution is restricted, the number of subqueries is $2^4 = 16$. When the distribution is unrestricted-2, the number of subqueries increases to $({}^8C_2 + {}^8C_1 + {}^8C_0) = 37$. The number of subqueries further increases to $({}^8C_4 + {}^8C_3 + {}^8C_2 + {}^8C_1 + {}^8C_0) = 163$ when the distribution changes to unrestricted-4.

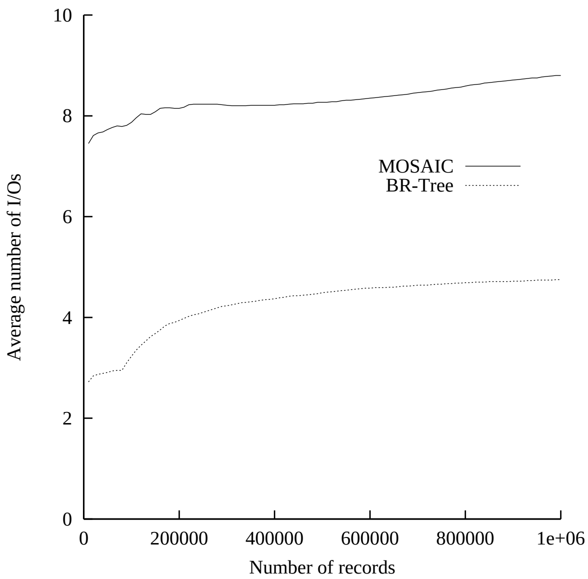


(a) Point queries.

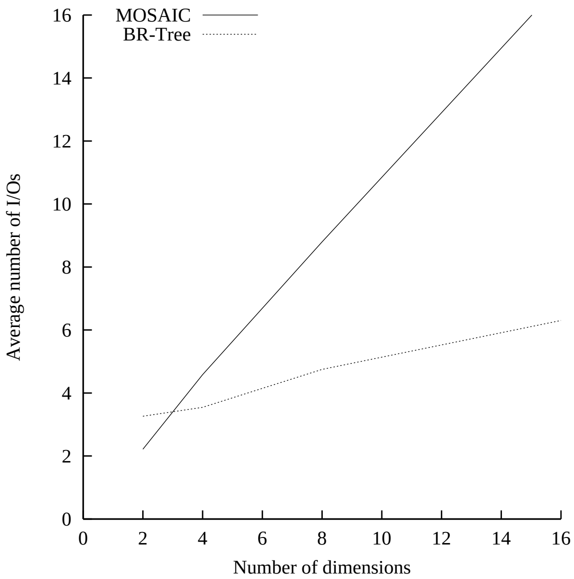


(b) Range queries with 2% range coverage.

Figure 5: Effect of different distributions for missing attribute values.



(a) Vary number of tuples with 8 dimensions.



(b) Vary number of dimensions with 1 million tuples.

Figure 6: Comparison of insertion cost.

To summarize the retrieval results, we have observed from the experiments that neither the BR-tree nor the MOSAIC structure is superior in all cases. While the BR-tree is effective for range queries on data with dimension-restricted data distribution, the MOSAIC structure performs better for unrestricted data distribution and point queries. These observations suggest that both the BR-tree and the MOSAIC structure are appropriate for different application domains. For example, in the case of multimedia applications, most if not all queries are range queries (as opposed to point queries), making the BR-tree a very valuable index in these contexts. On the other hand, the MOSAIC structure is expected to be better in the medical domain such as the thyroid database in our motivational example since missing values are distributed over a large number of attributes (dimensions).

4.5 On Insertion Cost

Figure 6 compares the average I/O cost in building the two indexes. The insertion cost provides an indication on the update performance of each indexing method. In Figure 6(a), we see the average I/O cost per tuple inserted for the default setting, i.e., a data set of 1 million tuples with 8 dimensions. We note that as the number of tuples increases, the average cost per insertion increases for both methods. This is expected since the index structures grow with more tuples. We also note that it is more costly to insert a tuple to the MOSAIC structure than to the BR-tree. The reason is because in MOSAIC, the insertion involves multiple (in this case, 8) indexes, whereas in BR-tree there is only one single tree structure to traverse. In fact, the insertion cost in MOSAIC is between two to three times more costly than that in BR-tree. This is despite the fact that the code is optimized for MOSAIC in the sense that the input data is sorted.

Figure 6(b) shows the relative performance of the insertion cost between the BR-tree and MOSAIC structure as the number of dimensions is varied. We observe that the insertion cost for MOSAIC grows faster than BR-tree as the number of dimensions increases. This follows from the same observation made earlier that there are more indexes to be updated for an insertion into the MOSAIC structure.

4.6 On Storage Cost

Though storage cost is getting cheaper and affordable, storage efficiency remains an important parameter for evaluating the effectiveness of an index structure. We present the storage cost (in terms of KB) for the MOSAIC structure and BR-tree in Figure 7. As expected, the MOSAIC structure is less storage efficient than the BR-tree. Its space consumption increases linearly but more steeply than that of the BR-tree as the number of dimensions increases As such, the relative difference between the two approaches widens quickly. For example, for small number of dimensions (e.g. 2), the storage cost is almost the same, but for large number of dimensions (e.g., 16), MOSAIC consumes up to 3 times more storage space than the BR-tree.

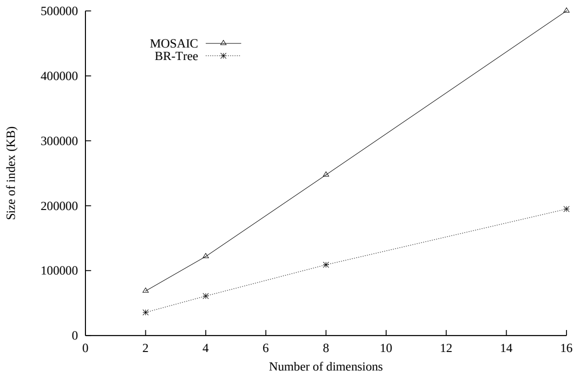


Figure 7: Comparison of storage cost.

5 Conclusion

In this paper, we have addressed the problem of high-dimensional data search in incomplete databases. Records and search keys in such a database may contain missing attribute values, which are not efficiently supported by existing high dimensional indexes. We have proposed and evaluated two indexing schemes called the Bitstring-augmented R-tree (BR-tree) and the MOSAIC structure. Experimental results showed that both methods are advantageous. In particular, the BR-tree has vastly superior performance compared to a regular R-tree that treats missing values as a distinguished token denoted by '-1'. This can be attributed to the novel mapping function we introduced to scatter the search keys "randomly" in the multi-dimensional search space that are being indexed. Moreover, it performs more efficiently compared to the MOSAIC structure for range queries and is superior in terms of update and storage costs. On the other hand, the MOSAIC structure outperforms the BR-tree in retrieval *for point queries*.

We are currently extending the work reported here in several ways. First, we have started to examine the impact of *partial queries*, i.e., queries that may contain missing information. Second, we also plan to study a number of other indexing approaches as suggested by the framework proposed in Section 3. In particular, a mix of multi-dimensional and single-dimensional indexes may provide a good balance in retrieval, update and storage cost. Last but not least, we will examine other methods to handle the large number of subqueries that are generated for approaches that are based on a single high-dimensional index. One promising approach is to "cluster" several subqueries together. For example, for a point query, its subqueries can be grouped into a smaller set of range queries.

Acknowledgment

This work is partially supported by NUS Research Grants RP950658. We like to thank Haoyu Dai for his research assistantship, and Kriegel and his colleagues for allowing us to use the dataset in [3] for our experiments.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–140, 1990.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[3] S. Berchtold, D.A. Keim, and H-P Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd VLDB Conference*, pages 28–39, Mumbai, India, September 1996.

[4] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shilovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, August 1997.

[5] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the 1997 ACM-SIGMOD Conference*, pages 357–368, Tucson, Arizona, May 1997.

[6] C.-Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proceedings of the 18th VLDB Conference*, pages 444–455, August 1992.

[7] C. Dyreson. Information retrieval from an incomplete data cube. In *Proceedings of the 22nd VLDB Conference*, pages 532–542, Mumbai, India, September 1996.

[8] C. Faloutsos, Ron Barber, Myron Flickner, Jim Hafner, Wane Biblack, Dragutin Petkovic, and William Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3):231–262, 1994.

[9] Garavan Institute and J.R. Ross. Thyroid disease dataset. In *Garavan Institute*, 1987.

[10] G.H. Gessert. Four valued logic for relational database systems. *SIGMOD RECORD*, 19(1):29–35, 1990.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[12] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the 1997 ACM-SIGMOD Conference*, pages 369–380, Tucson, Arizona, May 1997.

[13] A. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd VLDB Conference*, pages 402–412, Mumbai, India, September 1996.

[14] K. Lin, H.V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[15] R. Mehrotra and J.E. Gray. Feature-based retrieval of similar shapes. In *Proceedings of 9th Data Engineering Conference*, pages 108–115, Vienna, Austria, 1993.

[16] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings of EDBT'90*, pages 29–43, March 1990.

[17] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, 1984.

[18] B. C. Ooi, R. Sacks-Davis, and K. McDonelli. Spatial indexing by binary decomposition and spatial bounding. *Information Systems*, 16(2):211–237, 1991.

[19] T. Sellis, N. Roussopoulous, and C. Faloutsos. R^+-trees: A dynamic index for multi-dimensional objects. In *Proceedings of the 16th VLDB Conference*, pages 507–518, Brighton, England, August 1987.

[20] B.K. Shoichet, D.L. Bodian, and I.D. Kuntz. Molecular docking using shape descriptors. *Journal of Computational Chemistry*, 13(3):380–397, 1992.