Encoded Bitmap Indexing for Data Warehouses

Ming-Chuan Wu

DVS1, Computer Science Department
Technische Universität Darmstadt, GERMANY
wu@dvs1.informatik.tu-darmstadt.de

Alejandro P. Buchmann

DVS1, Computer Science Department
Technische Universität Darmstadt, GERMANY
buchmann@dvs1.informatik.tu-darmstadt.de

Abstract

Complex query types, huge data volumes, and very high read/update ratios make the indexing techniques designed and tuned for traditional database systems unsuitable for data warehouses (DW). We propose an encoded bitmap indexing for DWs which improves the performance of known bitmap indexing in the case of large cardinality domains. A performance analysis and theorems which identify properties of good encodings for better performance are presented. We compare encoded bitmap indexing with related techniques, such as bit slicing, projection-, dynamic-, and range-based-indexing.

1 Introduction

Complex query types, huge data volumes and very high read/update ratios play crucial roles in query processing in data warehouses (DW). These factors make the query processing/optimization techniques designed and tuned for On-Line Transaction Processing (OLTP) systems unsuitable for the DW environment. Many approaches have been proposed for query processing in DWs, such as, precomputation of summarized data, predefined access paths, special index techniques, *etc.* In this paper, we propose *encoded bitmap indexing* — an extension of known bitmap indexing, first proposed by O'Neil in the Model 204 DBMS [9].

In Section 2, we discuss bitmap indexing as proposed in [9, 11], and propose an encoded bitmap indexing to deal with large cardinality domains. We thus correct a shortcoming of simple bitmap indexing, which is best suited for low cardinality attributes. The basic idea of encoded bitmap indexing is to encode the attribute domain. Therefore, we also discuss how encoding affects the performance of the index. We define the concept *binary distance*, *chain* and *well-defined encoding* and derive theorems that define the properties of a *good* encoding. Some potential applications and variations of encoded bitmap indexing specific to the DW environment are also discussed, such as *hierarchy encoding*, *total-order preserving encoding* and *range-based indexes using encoded bitmap indexing*. In

Section 3 we give a comparative performance analysis of simple and encoded bitmap indexes. The result shows that encoded bitmap indexes perform better in most cases. Even if the problem size increases dramatically, the performance degrades logarithmically, while the performance of simple bitmap indexes degrades linearly. In Section 4 we discuss related indexing techniques, discuss the problems they solve and their differences with encoded bitmap indexing. In Section 5 we present conclusions and future work.

2 Bitmap Indexing Techniques

We present a brief overview of simple bitmap indexing and the application domain for which it is ideally suited. The limitations of simple bitmap indexing lead us to propose a new indexing technique — *encoded bitmap indexing*. The main advantages of encoded bitmap indexes are a drastic reduction in space requirements and corresponding performance gains.

2.1 Simple bitmap indexing revisited

The basic idea behind simple bitmap indexing is to use a string of bits (0 or 1) to indicate whether an attribute in a tuple is equal to a specific value or not. The position of a bit in the bit string denotes the position of a tuple in the table. The bit is set, if the content of an attribute is associated with a specific value. For example, a simple bitmap index on an attribute GENDER, with domain {Male, Female}, results in two bitmap vectors, say $\mathbb{B}_M$ and $\mathbb{B}_F$. For $\mathbb{B}_M$, the bit is set to 1, if the corresponding tuple has the attribute GENDER=Male, otherwise the bit is set to 0. For $\mathbb{B}_F$, the bit is set to 1, if the associated tuple has the attribute GENDER=Female, otherwise the bit is set to 0.[1] The simple bitmap index on the attribute GENDER, $\mathbb{B}^{GENDER}$, is the collection of bitmap vectors {$\mathbb{B}_M, \mathbb{B}_F$}.

B-trees and their variants (later simply denoted as B-trees) have been widely adopted in database systems as external indexing. They provide efficient mechanisms for searching and require time and space only logarithmic to the amount of indexed keys. Their

---

[1] Note that the negation of $\mathbb{B}_M$ must not necessarily be equivalent to $\mathbb{B}_F$ because of missing information and NULLs.

strength is their dynamic nature, performance and stability under update — properties that are not required in a DW. In the DW environment, simple bitmap indexing has advantages over B-trees, since 1) building/maintaining simple bitmap indexes *usually* costs less time and space, and 2) bitmap indexes can efficiently work together to reduce the search-space before really accessing the data.

COST ANALYSIS. Let $\mathfrak{T}$ be a table and $\mathfrak{T} = \{t_1, \ldots, t_n\}$. Define the cardinality of $\mathfrak{T}$ as $|\mathfrak{T}| = $ *the number of distinct tuples in* $\mathfrak{T}$. Then, building a simple bitmap index on an attribute A ($A \in \{a_1, \ldots, a_m\}$) of the table $\mathfrak{T}$ requires $\frac{|\mathfrak{T}| \times |A|}{8} = \frac{n \times m}{8}$ bytes, where m is the cardinality of A, defined as $|A| = $ *the number of distinct values in the domain of* A. On the other hand, building a B-tree on attribute A requires about $\frac{1.44 \times n}{M} \times p$ bytes, where p is the page size, and M is the degree of the B-tree [2, 1]. If $m < \frac{11.52p}{M}$, then simple bitmap indexes are more space efficient than B-trees. In other words, assume that $p = 4K$ and $M = 512$, then if the cardinality of A is smaller than 93 (*i.e.*, $m < 93$), building a simple bitmap index on A is more economic in size than building B-trees.

As for the time complexity, the complexity of building a B-tree on A is $\mathcal{O}(n \times \log_{\frac{M}{2}} m) + \mathcal{O}(n \times \log_2(\frac{p}{4}))$, where p is the page size and 4 is the size of a tuple-id. The first term denotes the cost of traversing from the root to the leaf nodes, and the second term denotes the cost of inserting the tuple-ids into the corresponding leaf nodes. On the other hand, the complexity of building a simple bitmap index on A is $\mathcal{O}(n \times m)$. If n (the cardinality of the indexed table) is very large, and m (the cardinality of the indexed attribute) is very small, then $\mathcal{O}(n \times \log_{\frac{M}{2}} m) + \mathcal{O}(n \times \log_2 \frac{p}{4}) > \mathcal{O}(n \times m)$, *i.e.*, the time complexity of building B-trees is larger than that of simple bitmap indexes.

COOPERATIVITY OF INDEXES. The main function of indexes is to accelerate query processing by sizing down the search space. Both B-trees and simple bitmap indexes can achieve this. However, if two or more selection conditions are given in a query, say $A = a_i$ AND $B = b_j$, separate B-trees on attribute A and attribute B cannot efficiently cooperate with each other.[2] We need to build another B-tree on the compound key (A, B). In contrast, separate simple bitmap indexes on A and B can efficiently work together to fetch the desired data by simply performing a logical operator, AND, on the corresponding bitmap vectors.

The impact of the cooperativity of simple bitmap

---

[2] Although multiple index accesses on value-list based indexes have been implemented in DB2 [5], the cost of multiple index accesses for bitmap indexing is much smaller than that of B-trees.

indexes is that if top n attributes with the highest referenced rate in users' queries are chosen, and indexes are to be built on them, we only need n simple bitmap indexes. Any combination of selection conditions involving any subset of the n attributes can be efficiently evaluated by applying corresponding logical operations on the bitmap vectors. If B-trees on compound keys are built, in order to cover all possible combinations of selection conditions among these n attributes, we need $C_1^n + C_2^n + \ldots + C_n^n = 2^n - 1$ B-trees. The cost of maintaining so many B-trees would be unacceptable. If we consider index cooperativity, simple bitmap indexes will have dominating advantages.

RESTRICTIONS. However, as the cardinalities of the keys increase, both the time and space complexity of building and maintaining simple bitmap indexes rapidly become higher. The sparsity of the bitmap vectors is another problem which comes with high cardinality. The sparsity of a bitmap vector is on average $\frac{m-1}{m}$, where m is the cardinality of the attribute. As m increases, the space utilization degrades. Second, for queries involving large range searches (range searches denote both IN-lists and range selections of the form $i < A < j$), the number of bitmap vectors which needs to be processed also increases. For large bitmap vectors, the cost cannot be ignored. In this case, simple bitmap indexes might perform worse than B-trees. To solve the problems derived from high cardinality, a new indexing technique — encoded bitmap indexing, is proposed.

2.2 Encoded bitmap indexing

Suppose that we have a fact table, SALES, with N tuples and a dimension table, PRODUCTS, containing 12000 different products. Traditionally, if we want to build a simple bitmap index on the PRODUCTS dimension, it will result in 12000 bitmap vectors of N bits in length. In encoded bitmap indexing, instead of 12000 bitmap vectors, $\lceil \log_2 12000 \rceil = 14$ bitmap vectors, plus a mapping table, are used. For example, suppose that the domain of attribute A of table $\mathfrak{T}$ is $\{a, b, c\}$. (As for the cases of NULL-values, or non-existing tuples, simple bitmap indexing uses separate bitmap vectors to represent them, while in encoded bitmap indexing, they are encoded together with other domain values. Further discussion can be found later in this section.) Instead of using 3 bitmap vectors, we use $\lceil \log_2 3 \rceil = 2$ bitmap vectors to build the index on attribute A.

As Figure 1 shows, we use 2 bits to encode the domain $\{a, b, c\}$, where a is encoded as 00, b as 01 and c as 10, respectively. For those tuples with $A = a$, we set corresponding positions in both bitmap vectors $\mathbb{B}_1$ and $\mathbb{B}_0$ to 0; for those with $A = b$, $\mathbb{B}_1 = 0$ and $\mathbb{B}_0 = 1$; and so on. In principle, the bitmap vector
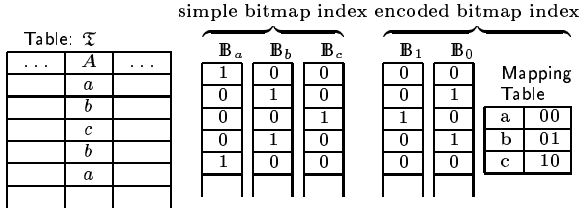
Figure 1: An example of encoded bitmap indexing

$\mathbb{B}_i$ stores the i-th bit (from the least significant bit, LSB, to the most significant bit, MSB) of the encoded value of attribute A. To retrieve data, we define the retrieval Boolean function for each value as follows. A retrieval Boolean function, or shortly *retrieval function*, is a k-variable min-term, where $k = \lceil \log_2 |A| \rceil = 2$ in this example. If a value v_0 is encoded as $b_1 b_0$ ($b_i \in \{0,1\}$, $i = 0, 1$), then the retrieval function for v_0 is defined as $x_1 x_0$, where $x_i = \mathbb{B}_i$, if $b_i = 1$, otherwise $x_i = $ *negation of* $\mathbb{B}_i$, i.e., $x_i = \mathbb{B}_i'$. For the above example, the retrieval functions for a, b and c should be $f_a = \mathbb{B}_1' \mathbb{B}_0'$, $f_b = \mathbb{B}_1' \mathbb{B}_0$, and $f_c = \mathbb{B}_1 \mathbb{B}_0'$, where x' denotes the negation of the variable x, xy denotes (x AND y), and $x + y$ denotes (x OR y). If we want to select data where $A = a$ OR $A = b$, then we simply apply an OR operator on f_a and f_b, i.e., $f_a + f_b = \mathbb{B}_1' \mathbb{B}_0' + \mathbb{B}_1' \mathbb{B}_0$, which can be further reduced to $\mathbb{B}_1'$. In other words, to retrieve tuples with $A = a$ OR $A = b$, we simply use the inverse of the bitmap vector $\mathbb{B}_1$ and the 1's indicate those tuples satisfying the selection conditions. We define the encoded bitmap index as follows:

Definition 2.1 (Encoded Bitmap Index) *Given a table $\mathfrak{T} = \{t_1, \ldots, t_n\}$, where t_j is a tuple of $\mathfrak{T}$ ($j = 1, \ldots, n$), let A be an attribute of $\mathfrak{T}$, denoted by $\mathfrak{T}.A$, and the domain of A be $\{a_1, \ldots, a_m\}$. Then, an encoded bitmap index, $\mathbb{B}^A$, on $\mathfrak{T}.A$ is a set of bitmap vectors $\{\mathbb{B}_{k-1}, \ldots, \mathbb{B}_0\}$, a one-to-one mapping ($\mathbb{M}^A : A \to \{\langle b_{k-1} \cdots b_0 \rangle | b_i \in \{0,1\}, i = 0, \ldots, k-1, and\ k = \lceil \log_2 m \rceil \}$) and a set of retrieval Boolean functions ($\{f_{a_1}, \ldots, f_{a_m}\}$). The bitmap vectors are defined as follows. $\forall\ \mathbb{B}_i$ ($i = 0, \ldots, k-1$), t_j ($j = 1, \ldots, n$), $\ni\ \mathbb{B}_i[j] = 1$, if $\mathbb{M}^A(t_j.A)[i] = 1$, else $\mathbb{B}_i[j] = 0$, where $\mathbb{B}_i[j]$ denotes the j-th bit of $\mathbb{B}_i$ and $\mathbb{M}^A(t_j.A)[i]$ the i-th bit (from LSB to MSB) of $\mathbb{M}^A(t_j.A)$. In addition, $\forall\ \alpha \in \{a_1, \ldots, a_m\}$, the retrieval function for α, f_α, is a k-variable min-term (fundamental conjunction) $x_{k-1} \cdots x_0$, where $x_i = \mathbb{B}_i$, if $\mathbb{M}^A(\alpha)[i] = 1$, otherwise $x_i = \mathbb{B}_i'$ ($i = 0, \ldots, k-1$).* ∎

● **MAINTENANCE OF ENCODED BITMAP INDEXES**
As data is updated, the encoded bitmap indexes need to be maintained. We discuss the maintenance for *updates without domain expansion* and *updates with domain expansion*.

UPDATES WITHOUT DOMAIN EXPANSION. Following the example above, if a tuple with $A = b$ is appended to table $\mathfrak{T}$, then we only need to append $\mathbb{B}_1[j] = 0$ and $\mathbb{B}_0[j] = 1$ at the end of bitmap vectors $\mathbb{B}_1$ and $\mathbb{B}_0$, where j is the position of the new inserted tuple in table $\mathfrak{T}$.

UPDATES WITH DOMAIN EXPANSION. If a tuple with $A = d$ is appended to $\mathfrak{T}$, i.e., the domain of A is now expanded to $\{a, b, c, d\}$, then the following equation should be first tested:

$$\lceil \log_2 |A^{(m-1)}| \rceil = \lceil \log_2 |A^{(m)}| \rceil, \qquad (1)$$

where $|A^{(m-1)}|$ denotes the cardinality of A before insertion, and $|A^{(m)}|$ denotes the cardinality of A after insertion. If Equation (1) is true, as is the case in our example, then add the mapping $\mathbb{M}^A(d) = 11$ into the mapping table and set $\mathbb{B}_i[j] = \mathbb{M}^A(d)[i]$ (where $i = 0, \ldots, k-1$ and $j = $ *the position of the new inserted tuple in* $\mathfrak{T}$), as Figure 2(a) shows, and set $f_d = \mathbb{B}_1 \mathbb{B}_0$. If another tuple with $A = e$ is further appended to $\mathfrak{T}$, i.e., the domain of A is now expanded to $\{a, b, c, d, e\}$, then $\lceil \log_2 |A^{(m-1)}| \rceil < \lceil \log_2 |A^{(m)}| \rceil$. The resulting bitmap vectors and the mapping table are shown in Figure 2(b).

The following actions need to be taken to reflect the change to the encoded bitmap index.
1. Expand the mapping $\mathbb{M}^A : \{A | A \in \{a, b, c, d\}\} \to \{\langle b_1 b_0 \rangle | b_i \in \{0, 1\}, i = 0, 1\}$ to $\mathbb{M}^A : \{A | A \in \{a, b, c, d, e\}\} \to \{\langle b_2 b_1 b_0 \rangle | b_i \in \{0, 1\}, i = 0, 1, 2\}$.
2. Add a bitmap vector $\mathbb{B}_2$ to $\mathbb{B}_A$, and set $\mathbb{B}_2$ to 0.
3. Set $\mathbb{B}_i[j] = \mathbb{M}^A(e)[i]$, where $i = 0, 1, 2$ and $j = $ *the position of the new inserted tuple in* $\mathfrak{T}$.
4. Add the Boolean function $f_e = \mathbb{B}_2 \mathbb{B}_1' \mathbb{B}_0'$ for the value e and revise the Boolean functions for a, b, c and d by AND*ing* $\mathbb{B}_2'$ to them, i.e., $f_a = \mathbb{B}_2' \mathbb{B}_1' \mathbb{B}_0'$, $f_b = \mathbb{B}_2' \mathbb{B}_1' \mathbb{B}_0$, $f_c = \mathbb{B}_2' \mathbb{B}_1 \mathbb{B}_0'$ and $f_d = \mathbb{B}_2' \mathbb{B}_1 \mathbb{B}_0$.

A general algorithm for maintaining the encoded bitmap indexes with respect to both types of updates can be found in [18].

Some questions which still need to be clarified in the encoded bitmap indexing are the representations for tuples, which are deleted or non-existing, or tuples with NULL values.

A simple way of solving these problems is to add bitmap vectors, B_{NotExist} and B_{NULL}, indicating the non-existing (or deleted) tuples and the tuples with NULL values, by setting the corresponding bit to 1. Another method is to assign the non-existing tuples and the tuples with NULL value artificial key values, and to encode these values together with the other key values. Intuitively, the second method is expected to perform better, since it reduces the number of bitmap vectors which need to be accessed while processing
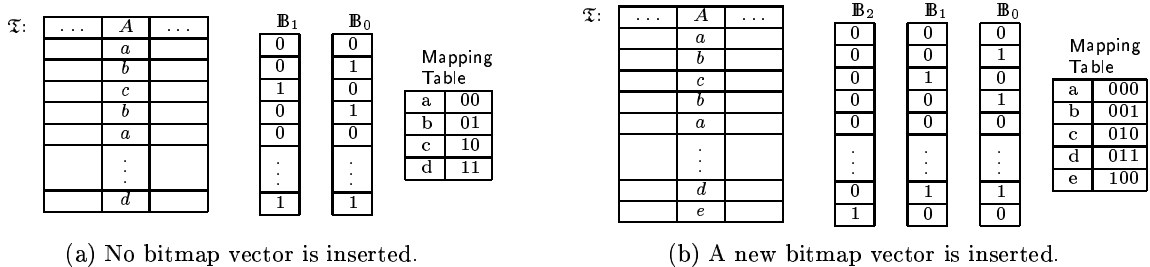
$\mathfrak{T}$:	...	A	...
		a	
		b	
		c	
		b	
		a	
		$\vdots$	
		d	

$\mathbb{B}_1$	$\mathbb{B}_0$
0	0
0	1
1	0
0	1
0	0
$\vdots$	$\vdots$
1	1

Mapping Table

a	00
b	01
c	10
d	11

(a) No bitmap vector is inserted.

$\mathfrak{T}$:	...	A	...
		a	
		b	
		c	
		b	
		a	
		$\vdots$	
		d	
		e	

$\mathbb{B}_2$	$\mathbb{B}_1$	$\mathbb{B}_0$
0	0	0
0	0	1
0	1	0
0	0	1
0	0	0
$\vdots$	$\vdots$	$\vdots$
0	1	1
1	0	0

Mapping Table

a	000
b	001
c	010
d	011
e	100

(b) A new bitmap vector is inserted.

Figure 2: Updates with domain expansions

queries. In the above example, the domain of attribute A, which is to be encoded, should be considered as $\{\mathsf{NotExist}, \mathsf{NULL}\} \cup \{a, b, c, d, e\}$.

The assignment of the encoded value for non-existing (void) tuples is arbitrary. Nonetheless, we suggest to reserve the value 0 for non-existing tuples for the sake of performance. For the above example, if we encode $\{\mathsf{NotExist}, \mathsf{NULL}, a, b, c, d, e\}$ as $\{000_{(2)}, 010_{(2)}, 011_{(2)}, 100_{(2)}, 101_{(2)}, 110_{(2)}, 111_{(2)}\}$, then, for the selection condition, "A IN $\{\mathsf{NULL}, a, b, c\}$", the retrieval Boolean expression will be $(\mathbb{B}_2'\mathbb{B}_1\mathbb{B}_0' + \mathbb{B}_2'\mathbb{B}_1\mathbb{B}_0 + \mathbb{B}_2\mathbb{B}_1'\mathbb{B}_0' + \mathbb{B}_2\mathbb{B}_1'\mathbb{B}_0)\overline{\mathbb{B}_2'\mathbb{B}_1'\mathbb{B}_0'}$, where the last term restricts the selections only to those existing tuples. The expression will be reduced to $(\mathbb{B}_2'\mathbb{B}_1 + \mathbb{B}_2\mathbb{B}_1')(\mathbb{B}_2 + \mathbb{B}_1 + \mathbb{B}_0)$, which will be further reduced to $(\mathbb{B}_2'\mathbb{B}_1 + \mathbb{B}_2\mathbb{B}_1')$. It results in an expression, as if we did not take the term, $\overline{\mathbb{B}_2'\mathbb{B}_1'\mathbb{B}_0'}$, into consideration. It is because all tuples with any of the three bitmap vectors, *i.e.*, $\mathbb{B}_2$, $\mathbb{B}_1$ and $\mathbb{B}_0$, set to 1, exist. The following theorem certifies our suggestion to reserve 0 for non-existing tuples.

Theorem 2.1 *Let void tuples of a table,* $\mathfrak{T}$*, be encoded as* 0*. Given any selection on attribute* A *of* $\mathfrak{T}$ *on any subset of existing tuples, the corresponding retrieval Boolean expression,* $f_{\sigma(A)}$ AND f_{void}'*, can be reduced to* $f_{\sigma(A)}$*, i.e., ignoring the selection condition on the existing tuples.* ∎

In other words, in such an encoding, any selection on any subset of non-void tuples can be evaluated without taking the function, f_{void}', into consideration. Therefore, it reduces the processing time, while in simple bitmap indexing, the *existence* bit vector must be always ANDed to the resulting bit vector to have the final bitmap for selection. For the proof of Theorem 2.1 please refer to [18].

● **THE ENCODING**
In Definition 2.1, we have defined that an encoded bitmap index includes a set of bitmap vectors, a one-to-one mapping and a set of retrieval functions. As the name suggests, the domain of the indexed attribute is

encoded by the mapping. So far, we did not mention how to define this mapping and how it would affect the performance of query processing. We will define *well-defined encoding* for the improvement of performance next. Let us first state the idea of well-defined encoding by the following example.

Given an attribute A with the domain $\{a, b, c, d, e, f, g, h\}$ and it is known that tuples with A in $\{a, b, c, d\}$, or $\{c, d, e, f\}$ are likely to be accessed together. Then, if we define the mapping as Figure 3(a) shows, to select tuples with conditions "A IN $\{a, b, c, d\}$" or "A IN $\{c, d, e, f\}$", only one bitmap vector is needed to be accessed for each case. For "A IN $\{a, b, c, d\}$", the retrieval Boolean function is $\mathbb{B}_2'\mathbb{B}_1'\mathbb{B}_0' + \mathbb{B}_2\mathbb{B}_1'\mathbb{B}_0' + \mathbb{B}_2'\mathbb{B}_1'\mathbb{B}_0 + \mathbb{B}_2\mathbb{B}_1'\mathbb{B}_0$, which can be reduced to $\mathbb{B}_1'$, and as for "A IN $\{c, d, e, f\}$", the retrieval Boolean function is $\mathbb{B}_2'\mathbb{B}_1'\mathbb{B}_0 + \mathbb{B}_2\mathbb{B}_1'\mathbb{B}_0 + \mathbb{B}_2'\mathbb{B}_1\mathbb{B}_0 + \mathbb{B}_2\mathbb{B}_1\mathbb{B}_0 = \mathbb{B}_0$.

Mapping Table	
a	000
c	001
g	010
e	011
b	100
d	101
h	110
f	111

Mapping Table	
a	000
b	001
c	010
d	011
g	100
h	101
e	110
f	111

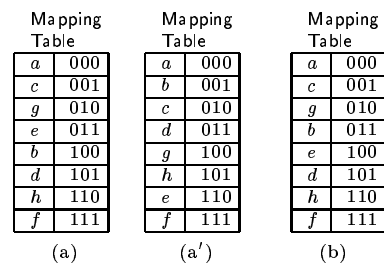| Mapping Table	
a	000
c	001
g	010
b	011
e	100
d	101
h	110
f	111

(a)   (a')   (b)

Figure 3: Examples of *proper* and *improper* mappings

In contrast, subject to the two selections above, the mapping in Figure 3(b) is not well-defined. The retrieval functions for "A IN $\{a, b, c, d\}$" and "A IN $\{c, d, e, f\}$" are $\mathbb{B}_2'\mathbb{B}_1 + \mathbb{B}_2'\mathbb{B}_0 + \mathbb{B}_1'\mathbb{B}_0$ and $\mathbb{B}_1'\mathbb{B}_0 + \mathbb{B}_2\mathbb{B}_1' + \mathbb{B}_2\mathbb{B}_0$, respectively, and they cannot be further reduced, *i.e.*, to evaluate the two selections three bitmap vectors must be read instead of one.

The idea is that, by a *well-defined* encoding (with respect to certain selection conditions), the number of bitmap vectors accessed is minimized, as a result shortening the processing time. Before going to the definition of well-defined encoding, let us first define *binary distance* and *chain*.

Definition 2.2 (Binary Distance) *Given two bi-*

nary numbers, x and y, the binary distance of x and y is a function, $\lambda()$, defined by $\lambda(x,y) = \mathsf{Count}(x \oplus y)$, where $\mathsf{Count}(z)$ is a function which returns the number of 1 bits in z, and $\oplus$ is the bitwise XOR operation. ∎

For example, if $a = 011_{(2)}$ and $b = 111_{(2)}$, then the binary distance of a and b is $\lambda(a,b) = 1$.

Definition 2.3 (Chain) *Given a set of distinct binary numbers, $\mathfrak{s} = \{c_0, \ldots, c_{n-1}\}$ $(n \geq 2)$. A chain in $\mathfrak{s}$ is defined as a sequence on $\mathfrak{s}$, say $< c_{o_0}, \ldots, c_{o_{n-1}} >$, such that $\lambda(c_{o_i}, c_{o_{i+1}}) = 1$ $(i = 0, \ldots, n-2)$ and $\lambda(c_{o_{n-1}}, c_{o_0}) = 1$.* ∎

Definition 2.4 (Prime Chain) *Given a set of distinct binary numbers, $\mathfrak{s} = \{c_0, \ldots, c_{n-1}\}$ and $|\mathfrak{s}| = 2^p$ $(p \in \mathbb{N} \cup \{0\})$. A chain on $\mathfrak{s}$ is said to be a prime chain, if $\forall \ c_i, c_j (i, j = 0, \ldots, n-1), \ni \lambda(c_i, c_j) \leq p$.* ∎

For example, a prime chain can be defined on $\{000, 110, 010, 100\}$ as $< 000, 100, 110, 010 >$, while no chain can be defined on $\{001, 011, 111\}$. Now, we define the *well-defined encoding* as follows.

Definition 2.5 (Well-Defined Encoding) *Given is a subdomain, $\mathfrak{s} = \{v_0, \ldots, v_{n-1}\}$ $(n \geq 2)$, of an attribute A, and let $p = \lfloor \log_2 n \rfloor$. A mapping on attribute A, $\mathbb{M}^A : A \rightarrow \{\langle b_{k-1} \cdots b_0 \rangle | b_i \in \{0, 1\}, i = 0, \ldots, k-1, \ k = \lceil \log_2 |A| \rceil\}$, is said to be well-defined with respect to the selection "A IN $\{v_0, \ldots, v_{n-1}\}$", if the following conditions are satisfied.*

i) If $n = 2^p$, then there exists a prime chain in $\{\mathbb{M}^A(v) | v \in \mathfrak{s}\}$.

ii) If $2^p < |\mathfrak{s}| < 2^{p+1}$, and $|\mathfrak{s}|$ is even, then $\exists \ \mathfrak{s}' \subseteq \mathfrak{s}, |\mathfrak{s}'| = 2^p$, such that there exists a prime chain in $\{\mathbb{M}^A(v) | v \in \mathfrak{s}'\}$, and there exists a chain in $\{\mathbb{M}^A(v) | v \in \mathfrak{s}\}$, and $\forall \ v, v' \in \mathfrak{s}, \ni \lambda\left(\mathbb{M}^A(v), \mathbb{M}^A(v')\right) \leq p + 1$.

iii) If $2^p < |\mathfrak{s}| < 2^{p+1}$, and $|\mathfrak{s}|$ is odd, then $\exists \ \mathfrak{s}' \subseteq \mathfrak{s}, \ |\mathfrak{s}'| = 2^p, \ni$ there exists a prime chain in $\{\mathbb{M}^A(v) | v \in \mathfrak{s}'\}$, and $\exists \ w \notin \mathfrak{s}$, but $w \in A, \ni$ there exists a chain in $\{\mathbb{M}^A(v) | v \in \mathfrak{s} \cup \{w\}\}$, and $\forall \ v, v' \in \mathfrak{s} \cup \{w\}, \ \lambda\left(\mathbb{M}^A(v), \mathbb{M}^A(v')\right) \leq p + 1$. ∎

Theorem 2.2 *A well-defined encoding minimizes the number of bit vectors which need to be accessed.* ∎

The proof of the theorem can be found in [18]. Obviously, Theorem 2.2 results in a local optimum, since Definition 2.5 takes only one subdomain (or, one selection condition) into consideration. Theorem 2.3 is a revision of Theorem 2.2 for describing the optimum relative to a set of selection conditions.

Theorem 2.3 *Given a set of (range) selection predicates on attribute A, $\mathbf{P}_{(A)} = \{p_1, \ldots, p_n\}$, and each p_i*

$(1 \leq i \leq n)$ corresponds to one subdomain of A, i.e., $\mathfrak{s}_1, \ldots, \mathfrak{s}_n$. The number of bit vectors which must be read while evaluating the selection predicates is minimized, if the encoding on A is well-defined with respect to all p_i $(1 \leq i \leq n)$. ∎

Again, the proof can be found in [18]. The subdomains, $\mathfrak{s}_1, \ldots, \mathfrak{s}_n$, are not necessarily disjoint, and the optimal solution must not necessarily exist, or be unique. In the above example, both the mappings in Figure 3(a) and (a') are optimal to both selections, "A IN $\{a, b, c, d\}$" and "A IN $\{c, d, e, f\}$".

A well-defined encoding is desirable for optimization but not essential. An efficient algorithm for finding a well-defined encoding is needed, since the *brute-force* approach has a complexity that is an exponential function of the cardinality of the attribute and the number of selection conditions. We have explored some heuristics for finding a well-defined encoding. However, they are beyond the scope of this paper. *Second,* intuitively, whether an encoding is well-defined is subject to the types of selections. In Definition 2.5, we define the *well-defined encoding* with respect to range selections of the form "$\mathsf{Attribute}$ IN $\{\ldots\}$". For other selection conditions, e.g., "$j < \mathsf{Attribute} < i$", we have to redefine the well-defined encoding, though, for discrete domains, conditions of the form — "$j < \mathsf{Attribute} < i$" can be expressed as "$\mathsf{Attribute}$ IN $\{\ldots\}$". In the next subsection, we give examples of handling range searches on numeric data types. *Third,* as stated above, whether an encoding is well-defined is specific to selections. As the selections change over time, a model is needed to evaluate when a re-mapping is desirable, or how to make use of *don't-care* values in the encoding to optimize the performance.[3]

2.3 Applications and variations of encoded bitmap indexing

● HIERARCHY ENCODING

The warehouse data is usually modeled as a star schema, which consists of one (or more) fact table(s) and some dimensions. Hierarchies might exist in dimensions. As Figure 4 shows, the dimension SALES-POINT of the sales data can be classified into three categories (*hierarchy elements*) — branch, company and alliance.

---

[3]For optimization of the retrieval Boolean expression, we might take the *don't-care* conditions into account. For example, if we want to select data with the selection condition $A = b$ OR $A = c$, then we consider the following two expressions: $f_b + f_c$ and $f_b + f_c + f_{don't\text{-}care}$, in the example in the beginning of Section 2.2, $f_{don't\text{-}care} = \mathbb{B}_1 \mathbb{B}_0$. Since $f_b + f_c = \mathbb{B}'_1 \mathbb{B}_0 + \mathbb{B}_1 \mathbb{B}'_0 = \mathbb{B}_1 \oplus \mathbb{B}_0$ and $f_b + f_c + f_{don't\text{-}care} = \mathbb{B}'_1 \mathbb{B}_0 + \mathbb{B}_1 \mathbb{B}'_0 + \mathbb{B}_1 \mathbb{B}_0 = \mathbb{B}_1 + \mathbb{B}_0$, for computers without hardware implementation of bitwise XOR operation, we might use $\mathbb{B}_1 + \mathbb{B}_0$ to retrieve the data.
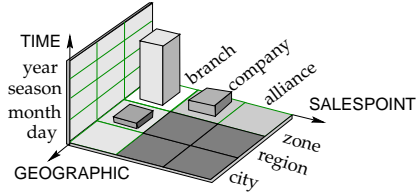
Figure 4: Hierarchies along dimensions

Suppose that we have 12 branches – $\{1, 2, 3, \ldots, 12\}$, 5 companies – $\{a, b, c, d, e\}$, and 3 alliances – $\{X, Y, Z\}$. Some branches belong to a company, and some companies form an alliance, e.g., branches $\{1, 2, 3, 4\}$ belong to company a, branches $\{5, 6\}$ belong to company b, ..., companies $\{a, b, c\}$ form the alliance X, and so on, as Figure 5(a) shows.

company	branches
a	$\{1, 2, 3, 4\}$
b	$\{5, 6\}$
c	$\{7, 8\}$
d	$\{3, 4, 9, 10\}$
e	$\{9, 10, 11, 12\}$

alliance	companies
X	$\{a, b, c\}$
Y	$\{c, d\}$
Z	$\{d, e\}$

(a)Members of hierarchy elements – company and alliance

Mapping Table	
1	0000
2	0001
3	0100
4	0101
5	0010
6	0011
7	0110
8	0111
9	1100
10	1101
11	1111
12	1110

(b)Hierarchy encoding

Figure 5: SALESPOINT hierarchy and its encoding

Note that some companies join two different alliances. In the real world, the relationships between hierarchy elements are not necessarily $1 : N$, they could also be $m : N$ as is the case in the above example.

One essential operation of OLAP is the manipulation along dimensions [17], e.g., roll-ups/drill-downs, data analysis along dimension hierarchies. All these operations are based on selections on dimensions, or on dimension elements, e.g., selecting sales data of all companies in alliance Z. Therefore, data of the same dimension hierarchies is very likely to be accessed together in the DW environment.

The idea of hierarchy encoding is to build encoded bitmap indexes with respect to selections on hierarchy elements. For the above example, the domains of hierarchy elements, "company" and "alliance", are $\{a, b, c, d, e\}$ and $\{X, Y, Z\}$, respectively, and the set of selection predicates on either "company" or "alliance" will be $\mathbf{P} = \{\sigma_{\mathsf{company}=i} | i \in \{a, b, c, d, e\}\} \cup \{\sigma_{\mathsf{alliance}=j} | j \in \{X, Y, Z\}\}$. A well-defined encoded bitmap index with respect to $\mathbf{P}$, as Figure 5(b) shows, is optimized for selections along dimension elements, "company" or "alliance". For example, for selection "alliance = X", only one bit vector is accessed.

This idea can be further extended to build a *groupset* index using encoded bitmap indexes. A groupset index corresponds to the Group-By clauses

in users' queries. Because of the limitation of space, we do not further discuss this case.

● **TOTAL-ORDER PRESERVING ENCODING**
Another type of range selection, such as "$j <$ Attribute $< i$", is performed on *numeric* or *ordinal* type of attributes. Numeric or ordinal types have a special property, namely, there exists a *total-order* relation in their domain. As a result, if the encoding in encoded bitmap indexes destroys the total-order relation, then selections in form of "$j <$ Attribute $< i$" must be rewritten to ones in form of "Attribute IN $\{\ldots\}$".

An encoding which preserves the total-order property of the attributes is called a *total-order preserving encoding*. A simple total-order preserving encoding is the internal representation of integers in computers, e.g., "8" is encoded as "1000", "17" as "10001". If we define the encoding as the internal representation of computers, the resulting encoded bitmap index is a set of bit slices of the original attribute. In [11], O'Neil and Quass defined such an index as *bit-sliced* index and proposed algorithms for evaluating some query types directly from the bit-sliced index.

Mapping Table	
101	000
102	001
103	010
104	100
105	101
106	110

Figure 6: Total-order preserving encoding

However, bit-sliced index is not the only answer to numeric (or ordinal) attributes. For example, given an attribute A with domain $\{101, 102, 103, 104, 105, 106\}$, where there exists a total-order in A, i.e., $101 < 102 < 103 < 104 < 105 < 106$. In addition, tuples with A in $\{101, 102, 104, 105\}$ are usually accessed together. The mapping in Figure 6 preserves on one hand the total-order property, and on the other hand, is optimized for the selection "A IN $\{101, 102, 104, 105\}$".

● **RANGE-BASED ENCODING**
A possible variation of encoded bitmap indexing is to use it for range-based indexing. Because of space limitations, instead of giving a formal definition of range-based encoded bitmap indexes, we demonstrate the idea by a simple example.

Given an attribute A with the domain $6 \leq A < 20$, $A \in \mathbb{N}$. Suppose that the following range selections are pre-defined by the end users — "$6 \leq A < 10$", "$8 \leq A < 12$", "$10 \leq A < 13$" and "$16 \leq A < 20$". According to the predefined selections, the domain of attribute A should first be divided into 6 disjoint partitions, as Figure 7 shows.

Next, we encode the set of intervals — $\{[6,8), [8,10), [10,12), [12,13), [13,16), [16,20)\}$ as Figure 8(a) shows.
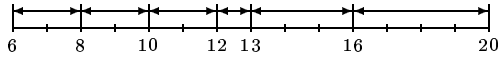
Figure 7: Pre-defined ranges

Then, for example, for range selection "$8 \leq A < 12$", the retrieval function is $\mathbb{B}_2' \mathbb{B}_1' \mathbb{B}_0 + \mathbb{B}_2 \mathbb{B}_1' \mathbb{B}_0$, which can be reduced to $\mathbb{B}_1' \mathbb{B}_0$. The (reduced) retrieval functions for all predefined range selections are listed in Figure 8(b).

Mapping Table

[6,8)	000
[8,10)	001
[10,12)	101
[12,13)	100
[13,16)	010
[16,20)	110

$6 \leq A < 10$	:	$\mathbb{B}_2' \mathbb{B}_1'$
$8 \leq A < 12$	:	$\mathbb{B}_1' \mathbb{B}_0$
$10 \leq A < 13$	:	$\mathbb{B}_2 \mathbb{B}_1'$
$16 \leq A < 20$	:	$\mathbb{B}_2 \mathbb{B}_1$

(a)Range encoding   (b)Retrieval functions

Figure 8: Range-based encoded bitmap index

If the ranges of selections are not pre-definable, or the ranges are so evenly scattered on the attribute domain (*which will result in many 1-element disjoint partitions*), then the range-based bitmap index will reduce to an encoded bitmap index on a set of single values, instead of on a set of ranges.

3 Performance Analysis: Analytical Approach

In Section 2, we have discussed the advantages of simple bitmap indexes over B-trees in the DW environment under some restrictions, and proposed encoded bitmap indexing to compensate for the limitations of simple bitmap indexing. In the following, we compare encoded bitmap indexing with simple bitmap indexing. By showing the advantages of encoded bitmap indexing over simple bitmap indexing, the advantages of encoded bitmap indexing over B-trees can be inferred.

3.1 Comparing encoded bitmap indexing with simple bitmap indexing

The space requirement of building both simple and encoded bitmap indexes is $\frac{|\mathfrak{T}| \times h}{8}$ bytes, and the time complexity is $\mathcal{O}(|\mathfrak{T}| \times h)$, where h is the number of bitmap vectors. In addition, the time complexity for maintenance with respect to updates without domain expansion is $\mathcal{O}(h)$ for both simple and encoded bitmap indexing. As for updates with domain expansion, the time complexity is $\mathcal{O}(|\mathfrak{T}|) + \mathcal{O}(h)$ for simple bitmap indexing, and between $\mathcal{O}(h)$ and $\mathcal{O}(|\mathfrak{T}|) + \mathcal{O}(h)$ for encoded bitmap indexing. The main difference is that for simple bitmap indexing, $h = |A|$, while for encoded bitmap indexing, $h = \lceil \log_2 |A| \rceil$. Obviously, $|A| > \lceil \log_2 |A| \rceil$, for all $|A| > 1$ and $|A| \in \mathbb{N}$. $|A| \gg \lceil \log_2 |A| \rceil$, if the cardinality of A is large.

Besides, the sparsity of simple bitmap indexes is on average $\frac{m-1}{m}$, where m is the cardinality of the indexed attribute, while the sparsity of encoded bitmap indexes is about $\frac{1}{2}$ (independent of m).

As a result, building/maintaining encoded bitmap indexes is more economical than building/maintaining simple bitmap indexes, as the cardinality of the indexed attribute increases.

However, maintenance cost is not the only factor when evaluating the performance of indexes. We should also compare the complexity of query processing with respect to both bitmap indexings. For both bitmap indexing techniques, the complexity is a function of the number of bitmap vectors which are accessed and the number of logical operations performed on the bitmaps.[4]

Following the example at the beginning of Section 2.2, suppose that we have an attribute A, with domain $\{a, b, c\}$. Both simple bitmap index and encoded bitmap index are built on A, as Figure 1 shows. Consider the following two queries:

Q1:   SELECT A Q2:   SELECT A
      FROM $\mathfrak{T}$ FROM $\mathfrak{T}$
      WHERE $A = a$ WHERE A in $\{a, b\}$

If the simple bitmap index is used, then ($\mathbb{B}_a$) and ($\mathbb{B}_a$ OR $\mathbb{B}_b$) are used for retrieving tuples for Q1 and Q2, respectively. If the encoded bitmap index is used, then ($\mathbb{B}_1' \mathbb{B}_0'$) and ($\mathbb{B}_1'$) are used to select the tuples for Q1 and Q2, respectively. Generally speaking, for single value selection, simple bitmap indexing performs better than encoded bitmap indexing. However, for range searches, *especially* large range searches, encoded bitmap indexing performs better than simple bitmap indexing. As the above example shows, for single value selection (Q1), one bitmap vector is accessed if simple bitmap indexing is used, while two bitmap vectors are accessed if encoded bitmap indexing is used. In contrast, for range search (Q2), one bitmap vector is accessed if encoded bitmap indexing is used, while two bitmap vectors are accessed if simple bitmap indexing is used.

Let c_s and c_e denote the number of bitmap vectors accessed by simple bitmap indexing and encoded bitmap indexing, respectively. Obviously, $1 \leq c_s \leq |A|$ and $1 \leq c_e \leq \lceil \log_2 |A| \rceil$. For simple bitmap indexing, $c_s = \delta$, where δ denotes the size of the interval of the range search and $1 \leq \delta \leq |A|$. For example, $\delta = 2$ in

---
[4]Comparing with the disk access costs, it is reasonable to ignore the CPU time needed for performing logical operations, such as AND , OR . In addition, in the following discussion, we consider the number of bitmap vectors which must be accessed for query processing using encoded bitmap indexing as the number of bitmaps after performing logical reduction on the retrieval Boolean expressions, *e.g.*, if the retrieval Boolean expression is $\mathbb{B}_1' \mathbb{B}_0 + \mathbb{B}_1 \mathbb{B}_0$, then it is first reduced to $\mathbb{B}_0$, and the number of bitmaps which need to be accessed is considered as one.

Q2.

For encoded bitmap indexing, c_e is a function of δ, where $1 \leq \delta \leq |A|$, and of the distribution of selected values. For worst cases, $c_e = \lceil \log_2 |A| \rceil$. For best cases, $c_e = \lceil \log_2 |A| \rceil - \gamma'$, where γ' is the number of bitmap vectors reduced by performing logical reduction. (For details please refer to Property 3.1 in [18].)

From the above discussion, we can see that $c_e < c_s$, if $\delta > \log_2 |A| + 1$. In addition, the cost of processing simple bitmap indexes is a linear function of δ, while the cost of processing encoded bitmap indexes is upper-bounded by a step function — $\lceil \log_2 |A| \rceil$. In other words, the encoded bitmap indexes perform stably, even when δ is large, while simple bitmap indexes degrade relatively fast. Figure 9(a) and (b) depict c_e and c_s with $|A| = 50$ and 1000, respectively. (c_e is calculated according to Property 3.1 in [18], i.e., of the best cases. For worst cases, $c_e = \lceil \log_2 |A| \rceil$, namely, $c_e = 6$ in Figure 9(a), and $c_e = 10$ in Figure 9(b), which are still much less than c_s.)

(a)$|A| = 50$, $\lceil \log_2 |A| \rceil = 6$

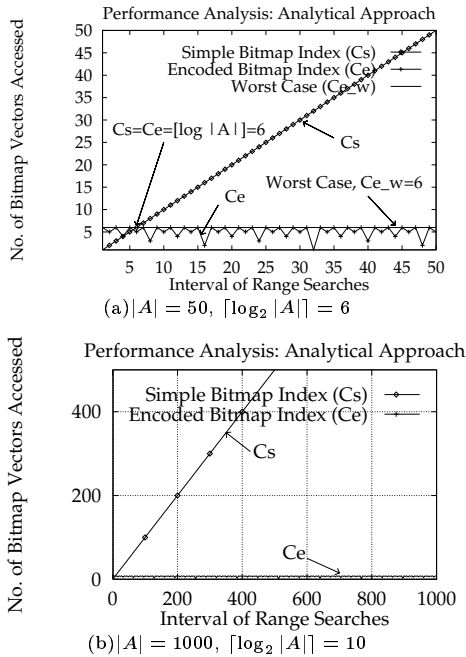(b)$|A| = 1000$, $\lceil \log_2 |A| \rceil = 10$

Figure 9: Performance analysis

Figure 10 depicts the number of bit vectors required for building both simple and encoded bitmap indexes with respect to the cardinality of indexed attributes. Again, the space requirement of a simple bitmap index is linear to the cardinality of the attribute, while that of an encoded bitmap index is logarithmic to the cardinality of the attribute.

3.2 Worst case analysis

Even for the worst case scenario, encoded bitmap indexing performs better than simple bitmap indexing,
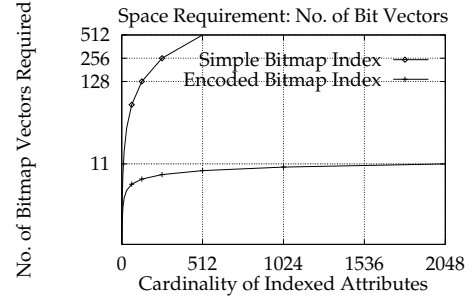

Figure 10: Space requirements

if $\delta > \log_2 |A|$, as Figure 9(a) shows. Two reasons that might lead to such behavior are discussed below.

Improper Encoding Given a selection, σ_1, if the encoding was not well-defined with respect to σ_1, for the worst cases, the number of bit vectors, which must be accessed in query processing, is $\lceil \log_2 |A| \rceil$. An extreme case will be that, for all types of selections there does not exist any selection, such that the encoding is well-defined. The line $c_{e\_w} = 6$ in Figure 9(a) depicts the extreme case.

The ratio between the areas under the curve of the best case and the line $c_{e\_w} = 6$ denotes the average benefit gained from well-defined encodings. The ratio for the case in Figure 9(a) is 0.84, i.e., 16% saving of the processing cost is gained, and the ratio for the case in Figure 9(b) is 0.90, i.e., 10% saving of processing cost is gained.

Note that the above calculation did not take the frequencies of selection types into consideration. The average savings are not very large in magnitude, so that having a well-defined encoding is desirable but not essential. For specific situations, the saving could be up to 83% (for the case where $\delta = 32$ in Figure 9(a)), or even up to 90% (for the case where $\delta = 512$ in Figure 9(b)).

Logical Reduction A well-defined encoding only makes sense together with the logical reduction of the retrieval functions. The complexity of performing logical reduction using a *brute-force* method is, however, exponential to the number of bit vectors. For the performance gain from well-defined encoding, we have to pay the price of finding a well-defined encoding and the cost of the logical reduction in exchange.

We do not think it is unfeasible, though the complexities for both finding a well-defined encoding and performing logical reduction are exponential to the problem size. We have explored some heuristics to solve the problem, but discussion of these preliminary results is beyond the scope of this paper. Another straightforward, but effective, approach will be: since the ranges of selection predicates are pre-definable

(*well-defined encodings* are subject to predefined selections), the retrieval functions for all the predefined predicates can also be reduced by human experts, and be verified with assistance of computers.

Furthermore, the cost for finding a well-defined encoding is a one-time cost, unless *dynamic re-encoding* is desired, which is also beyond the scope of this paper.

The above analysis shows that, for range searches, encoded bitmap indexes perform essentially better and more stable, even when the range of selection increases. For single value selections, encoded bitmap indexing is second to simple bitmap indexing. However, according to TPC-D [14], from 17 query types, 12 query types involve range search.(They are Q1, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q12, Q14 and Q16.) Therefore, we believe that the encoded bitmap indexing can play an important role for improving query processing in the DW environment. In addition, the worst case analysis shows that even if the best cases cannot be reached, the worst-case performance of encoded bitmap indexes in range searches is still better and more stable than that of simple bitmap indexes.

In practice, in order to improve both the performance of B-trees and simple bitmap indexes, a hybrid method is implemented, *i.e.*, instead of storing tuple-ids (value-lists) at the leaf-nodes of B-trees, bitmap vectors are stored. As the sparsity increases (one consequence of high cardinality), the bit vectors are expressed as value-lists. However, when the cardinality is very high (exactly the circumstance where encoded bitmap indexing is well-suited), the hybrid method might degrade to a pure B-tree. In such cases, the co-operativity of bitmap indexing in the hybrid method is lost.

4 Other Indexing Techniques

In this section, we discuss some other indexing techniques which are proposed in the literature for DWs.

Projection Indexing In [11], a projection index is defined as materialization of all values of an attribute in the tuple-id order. We can treat a projection index as an encoded bitmap index, where the mapping table is simply the table of internal codes, *i.e.*, the mapping function is

$$\mathbb{M}^A(\cdot) = b_{k-1} \cdots b_0$$

where k is the length in bits of the internal representation of attribute A, and b_i is the i-th bit (from LSB to MSB) of the attribute value ($i = 0, \ldots, k-1$).

One difference between a projection index and an encoded bitmap index, which uses the table of internal codes as the mapping table, is the physical storage allocation. A projection index stores the values *horizontally*, while an encoded bitmap index stores the

values *vertically*. A projection index stores the bits of a value continuously, while an encoded bitmap index stores the bits of the same position of different values continuously, which resembles the physical storage allocation of *bit-sliced indexes*.

Bit Slicing In [11], a bit-sliced index is defined as a set of bitmap slices which are orthogonal to the data held in a projection index. In other words, a bit-sliced index is a *bitwise* vertical partition of a projection index. Bit-sliced indexes are suitable for numeric (fixed-point) or ordinal attributes, and are especially good for wide-range searches. Bit-sliced indexing with non-binary base was also introduced in [11]. Bit-sliced indexes can also be treated as special cases of encoded bitmap indexing. They are encoded bitmap indexes with a total-order preserving encoding, which is trivially the internal representation of fixed-point numbers.

Both projection indexes and bit-sliced indexes are comparable to the *transposed files* [16]. A transposed file stores one column from a main table in a row, namely, one row per transposed column. A projection index stores only one transposed column, and a bit slice is a transposed representation of a column of bits from the same bit position of an attribute.

Value List Indexes Traditionally, a value list index stores key values and a list of tuple-ids for each key value. A value list index can be structured as a B-tree, or simply as an *inverted* file.

A hybrid indexing using simple bitmap indexes and value list indexes was said to resolve the problems of sparsity in simple bitmap indexes caused by high cardinality domains. The B-tree structure is first used to index the key values, and at the leaf nodes simple bitmap vectors are stored. However, if one bit vector is *too* sparse, a list of tuple-ids, instead of a bit vector, is stored. A contradiction arises: B-tree is efficient for random access, if the number of key values is large. However, if the number of key values is large, *i.e.*, the cardinality of the indexed attribute is large, then the problem of sparsity is more severe. As a result, instead of bit vectors stored at the leaf nodes, value lists are stored. Then, the so-called hybrid index reduces to a B-tree. On the other hand, if the cardinality of the indexed attribute is very small, the benefit of building a B-tree on top of the bitmap vectors is also small.

In [11], the authors have proposed algorithms for evaluating some aggregate functions and range selections directly on projection indexes, bit-sliced indexes and value list indexes. The range selection predicates considered in [11] was only of the form "$i < A < j$", while in our paper, we have generalized the cases, by taking both in-lists and conventional range pred-

icates into consideration. For the special cases of numeric/ordinal attributes, if the encoding is total-order preserving, the algorithms proposed by O'Neil and Quass are also applicable to the encoded bitmap indexes. Slight changes might, however, be required.

Group-Set Indexes Group-By operations are often used for grouping the results of queries for better understanding and analyzing. A groupset bitmap index was introduced in [11] to select tuples which satisfy the group-by condition.

The proposed groupset bitmaps face the same sparsity problem of simple bitmap indexes. Some other approaches, such as clustering, or segmentation, can help to process Group-By operations more efficiently. However, clustering can be performed according to only one selection condition or one grouping condition. Therefore, secondary indexes are needed. An eligible candidate for group-set indexing will be the encoded bitmap index. If we had 3 attributes in the Group-By clause, and the cardinalities of the attributes are 100, 200, 500, respectively. Then, the number of all possible combinations will be 10^7, which means 10^7 bit vectors if simple bitmap indexing is used, and only 20 bit vectors if encoded bitmap indexing is used.[5]

Furthermore, if *hierarchy encoding* (discussed in Section 2.3) is applied, groupset indexes can be *dynamically* calculated at run-time, which results in more flexibility, since it is not feasible to pre-compute all possible Group-By combinations if the number of dimensions is large.

Dynamic Bitmaps Dynamic bitmaps are built *dynamically* from high cardinality attributes. [13] If there are n different values in the attribute domain, they are encoded onto n ($\log_2 n$)-bit continuous binary integers.

Dynamic bitmaps are special cases of encoded bitmap indexes, where the encoding trivially maps the domain onto a continuous integer set. The significance of encoding was not discussed in dynamic bitmaps.

Range-Based Indexing A dynamic range-based bitmap indexing for high cardinality attributes with skew was proposed in [19]. The idea is to partition the domain into some equal population subsets, and simple bitmap vectors are constructed, one for each subset. In that work, the authors also took the distribution of the attributes into consideration.

In Section 2.3, we have also introduced a similar idea of building range-based indexes using encoded bitmap indexing. The two approaches differ from each other in the following aspects: (1) In [19], partition is done by distribution of the attribute values, while we propose to partition according to pre-defined range selections. (2) In [19], Wu and Yu investigate how to dynamically adjust the partition of the ranges to balance the population of all *buckets* with respect to the distribution of attribute values. However, we do not have the problem of imbalance. Because we use the pre-defined selection predicates to partition the attribute domains, the retrieval functions will, therefore, exactly match the desired tuples. Even in the cases that selection predicates are not pre-definable, or the predicates result in a very large number of small partitions, encoded bitmap indexing can handle a much larger number of small partitions than simple bitmap indexing can do. As a matter of fact, if the ranges of the selections are not pre-definable, range-based indexes do not make any sense. In this case, we propose to use an encoded bitmap index with a total-order preserving encoding, such that any range selection predicates can be efficiently evaluated directly on the bit vectors.

Other Techniques Other indexing techniques for the warehouse environment include *multidimensional B-trees* [8, 4], compression techniques (*e.g.*, run-length) for simple bitmap indexes, *hierarchical indexes* [6, 7], *join indexes* [15, 10] and multidimensional indexing for spatial data [12]. Index techniques used in Sybase IQ, Red Brick Warehouse and Oracle are discussed in [3].

5 Concluding Remarks and Future Work

We introduced *encoded bitmap indexing* for the DW environment. The merits of this technique are:

1. It inherits the good properties of simple bitmap indexing, such as *cooperativity of different bitmap vectors, low cost of construction and maintenance*, and *low processing cost*.

2. It differs from simple bitmap indexing in encoding. Because of encoding, it *solves the problems of sparsity*, at the same time, *improves the space utilization, shortens the maintenance and processing time*, and also *improves the performance of processing range searches*. Most of all, the cardinality of the indexed attribute has no longer dramatical effects on the maintenance and processing cost of the encoded bitmap indexes.

3. With customized definitions of encodings, the encoded bitmap indexes are suitable for and capable of (but not limited to) indexing OLAP data. We have discussed some of its applications, such as the *hierarchy encoding* for indexing dimensions with hierarchies, *total-order preserving encoding* for numeric/ordinal attributes, *range-based encoded bitmap indexes*, etc. Theorems were derived for identifying the properties

---

[5]Naturally, in this problem, the density of the products of the dimensions should also be considered, *e.g.*, although there are 10^7 combinations, there might only be 10^6 meaningful combinations, *i.e.*, the density is only 10%.

for a *well-defined encoding* with respect to a given set of predefined selection predicates. Under this encoding, the number of bit vectors, which must be accessed in query processing, is minimized. We have given a comparative performance analysis of both simple and encoded bitmap indexes using an analytical approach. The result is satisfactory and shows that as the cardinality and the range of selections increase, encoded bitmap indexes perform better and more stable than simple bitmap indexes (even if the best cases described by the theorems 2.2 and 2.3 cannot be reached).

There are still some problems to be solved. *First*, an efficient algorithm for logical reduction of the retrieval Boolean functions is needed. *Second*, an efficient algorithm for finding well-defined encodings is required to take full advantages of optimization. *Third*, for application domains where the set of predefined selection predicates changes over time, a model for evaluating the cost-effectiveness of a reconstruction of the encoded bitmap indexes is desirable. *Fourth*, if selection predicates are not predictable, a *proper* encoding is, however, achievable through an analysis of the history of users' queries. In other words, in such an environment, data mining might be applied for finding a good encoding. *Fifth*, in the text, we have concentrated on how range selections are evaluated directly on the encoded bitmap indexes, since selections are the very basic operation for other operations. However, in addition to range predicates, some aggregate functions, or operations can also be evaluated directly on the bitmaps, such as sum($\cdot$), average($\cdot$), median, N-tile, column-product aggregations, joins, *etc.* Algorithms for performing these functions, or operations using encoded bitmap indexes, though of no difficulty, must be defined.

References

[1] J-H. Chu, G. Knott, *An Analysis of B-Trees and Their Variants*, Information Systems, Vol. 14, No. 5, 1989.

[2] D. Comer, *The Ubiquitous B-Tree*, Computing Surveys, Vol. 11, No. 2, 1979.

[3] H. Edelstein, *Technology Analysis: Faster Data Warehouses*, Information Week, Dec 4, 1995.

[4] M. Freeston, *A General Solution of the n-dimensional B-tree Problem*, SIGMOD Conf., San Jose, CA, 1995.

[5] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[6] T. Johnson, D. Shasha, *Hierarchically Split Cube Forests for Decision Support: description and tuned design*, TR 727, NYU, *file://cs.nyu.edu/pub/tech-reports/*, Nov 1996.

[7] T. Johnson, D. Shasha, *Some Approaches to Index Design for Cube Forests*, Bulletin of the Technical Committee on Data Eng., Vol. 20, No. 1, Mar 1997.

[8] H. Leslie, R. Jain, D. Birdsall, H. Yaghmai, *Efficient Search of Multidimensional B-Trees*, VLDB Conf., Zurich, Switzerland, 1995.

[9] P. O'Neil, *Model 204 Architecture and Performance*, Springer-Verlag LNCS359, 2nd Intl. Workshop on High Performance Transactions Systems, Asilomar, CA, Sept 1987.

[10] P. O'Neil, G. Graefe, *Multi-Table Joins Through Bitmapped Join Indices*, SIGMOD Record, Vol. 24, No. 3, September 1995.

[11] P. O'Neil, D. Quass, *Improved Query Performance with Variant Indexes*, SIGMOD Conf., Tucson, AZ, May 1997.

[12] S. Sarawagi, M. Stonebraker, *Efficient Organization of Large Multidimensional Arrays*, ICDE, Houston, 1994.

[13] S. Sarawagi, *Indexing OLAP Data*, Bulletin of the Technical Committee on Data Eng., Vol. 20, No. 1, Mar 1997.

[14] Transaction Processing Performance Council (TPC), *TPC Benchmark D, Decision Support*, Standard Specification Revision 1.2.1, Dec 15, 1996.

[15] P. Valduriez, *Join Indices*, ACM TODS, 12(2), June 1987.

[16] G. Wiederhold, *Database Design*, 2nd Ed., McGraw-Hill Book Co., 1983.

[17] M.C. Wu, A. Buchmann, *Research Issues in Data Warehousing*, Datenbanksysteme in Büro, Technik und Wissenschaft, Editor: K.R.Dittrich and A. Geppert, Springer Verlag, 1997.

[18] M.C. Wu, A. Buchmann, *Encoded Bitmap Indexing for Data Warehouses*, Tech. Report DVS97-3, CS Dept., Technische Universität Darmstadt, (*http://www.informatik.tu-darmstadt.de/DVS1/staff/wu.german.html*), July 1997.

[19] K.L. Wu, P.S. Yu, *Range-Based Bitmap Indexing for High Cardinality Attributes with Skew*, Research Report, IBM Watson Research Center, May 1996.