Query Optimization for Selections using Bitmaps

Ming-Chuan Wu
Database Research Group, Computer Science Department
Technische Universität Darmstadt, Germany
wu@dvs1.informatik.tu-darmstadt.de

Abstract

Bitmaps are popular indexes for data warehouse (DW) applications and most database management systems offer them today. This paper proposes query optimization strategies for selections using bitmaps. Both *continuous* and *discrete* selection criteria are considered. Query optimization strategies are categorized into static and dynamic. Static optimization strategies discussed are the optimal design of bitmaps, and algorithms based on tree and logical reduction. The dynamic optimization discussed is the approach of inclusion and exclusion for both bit-sliced indexes and encoded bitmap indexes.

1 Introduction

Bitmap indexing has become a promising technique for query processing in DWs. Variations of bitmap indexes include bit-sliced indexes [14, 3], encoded bitmap indexes (EBI) [18], bitmapped join indexes [13], range-based bitmap indexes [20], and others[16].

For query operations, such as selections, aggregates, and joins, query evaluation algorithms using bitmaps have been proposed in recent years. In this paper, we further explore the issues of query optimization using bitmaps and concentrate on optimizing *selections*.

Indexes are used to speed up the evaluation of selection conditions followed by the retrieval of desired data. If no pipelining or parallelism is applied, the query response time can be expressed by the sum of the time of index processing plus the time of data retrieval. If the selectivity of a query, which is defined as the ratio of the cardinality of the final result to that of the base table, is high, the time of data retrieval may close in on the time of a costly table scan. For example, for a selectivity about 35%, over 99.8% data pages of the underlying table will be hit[1]. For such cases, using indexes has negative effects on query performance. Even for low selectivities, if the time of index processing is high, the total time spent on index processing and data retrieval may be longer than that of a table scan. Consequently, query optimization techniques discussed in this paper, which reduce the index processing time, do not only contribute to a better query performance at low selectivities, but also extend the feasibility of bitmap indexes at medium selectivities.

We divide query optimization techniques into *static* and *dynamic* ones. Static optimization is performed at design-time. It includes optimal index designs and improved algorithms for index processing. Dynamic optimization is performed at run-time. It is achieved by strategies which exploit run-time information, such as constraints, statistics, or distribution information of underlying data, to determine a better execution plan.

For index design, we explore the effects of two different selection types – *continuous range selections* and *discrete range selections*. In [3], design criteria for bit-sliced indexes with respect to continuous range selections were proposed. We extend their results to cover both types of selections and define a new design criterion for finding global time-optimal indexes. As for EBIs, we introduce *well-defined encoding*, which improves the time efficiency of the indexes without sacrificing space efficiency.

As for algorithm design, we develop a *tree-reduction* technique to improve the performance of the algorithm `RangeEval-Opt` (proposed by [3]) for bit-sliced indexes. For EBIs, *logical reduction* techniques are used to reduce the index processing time.

The dynamic query optimization technique introduced in this paper is the *principle of inclusion and exclusion*. We show how this principle can be applied to both bit-sliced indexes and EBIs. Cost models, both analytical and probabilistic, are defined to determine a better evaluation plan for selections.

The following example is used throughout this paper.

[1]The expected number of pages which are hit by selecting k tuples from a table of n pages is computed by $n \cdot \left(1 - \prod_{r=1}^{k}(pn \cdot \frac{n-1}{n} - r + 1)/(pn - r + 1)\right)$, where each page contains p tuples. The hit rate depends, of course, highly on the value p, clustering criteria, distribution of the indexed attribute, *etc.*

Example 1 *Given are two attributes A and B of a table T. Let the domain of A, denoted by $Dom(A)$, be $\{A|100 \leq A \leq 900,\ A \in \mathbb{Z}^+\}$ and $Dom(B) = \{a, b, c, d, e, f, t, u, v, w\}$. The cardinality of T is defined by $|T|$ and the cardinality of an attribute is defined by the cardinality of its domain.*

The rest of the paper is organized as follows: We revisit EBI and variants of bit-slicing briefly in section 2.1, and give a cost model for index performance analysis in section 2.2. In section 3, we discuss issues concerning static query optimization, including index design in section 3.1 and algorithm design in section 3.2. Strategies for dynamic query optimization are introduced in section 4, followed by conclusions in section 5.

2 Variants of bitmap indexes

The first member of the bitmap indexing family, named *simple bitmap indexing* here, was introduced in the 1960's [12]. A classical example of simple bitmap indexing is an index on an attribute GENDER. Suppose that the domain of the attribute is {M, F}. A simple bitmap index on it consists of two bit vectors, one for M and the other for F. The length of the bit vectors is equal to the cardinality of the indexed table. The bits of the bit vector M are set if the tuples of the corresponding bit-positions have GENDER=M. Likewise, the bit vector for F is set for GENDER=F. If deleted tuples are only tagged as "*deleted*" but kept in the database, as most of the database management systems do, an extra bit vector for "*existing*" tuples, E, is required. The bits of E are set if the corresponding tuples are not deleted.

The space requirement of a simple bitmap index is a linear function of the cardinality of the indexed attribute and of the indexed table, and the index processing time for a single value selection is a linear function of the length of bitmaps. The sparsity of the bit vectors increases with the cardinality resulting in poor space utilization and high processing cost.

Many variations of bitmap indexing have been proposed to solve the sparsity problems. Two common objectives of the proposed methods are (1) reducing the space complexity of the index and (2) improving the performance of index processing. Solutions include compressing bitmaps, *e.g.*, through *run-length* encoding, and transforming bitmap representation to tuple-id lists. Although these two methods are quite efficient in reducing the space requirements of bitmap indexes, they sacrifice the advantages of bitmap indexing in query processing — namely, the low-cost bitwise operations in index processing and the capability of *multiple index scans*[2]. In this paper, we discuss approaches that *do* preserve the advantages of bitmap indexing. They are *encoded bitmap indexing* (EBI) and *bit-slicing*.

[2]That is, combining multiple index structures to evaluate logical conjunction or disjunction of selection predicates.

2.1 Bit-slicing and EBI revisited

2.1.1 Bit-slices

A bit-sliced index (named *binary bit-sliced index* later) of an attribute is a bitwise projection of the attribute [14]. For example, suppose that the attribute A from Example 1 is defined as a two-byte short integer. A binary bit-sliced index on A consists of 16 bit vectors and is defined as shown in Figure 1[3]. Bits, b_0

$\cdots$	A	$\cdots$	$b_{15} \ldots b_{10}$	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	201		$0 \cdots 0$	0	0	1	1	0	0	1	0	0	1
	100		$0 \cdots 0$	0	0	0	1	1	0	0	1	0	0
	900		$0 \cdots 0$	1	1	1	0	0	0	0	1	0	0

Figure 1: A Binary Bit-Sliced Index on Attribute A

to b_{15}, store the internal binary representation of the corresponding attribute values. (b_{10} to b_{15} are all zeros, since $A \leq 900$). The number of bit vectors is equal to the length of the attribute's data type in bits, and the length of each bit vector is equal to the cardinality of the indexed table.

A bit-sliced index can also have non-binary or non-uniform base. For example, a decimal, uniform-based bit-sliced index on A has 3 *components*. Each component has 10 bit vectors, *i.e.*, each decimal digit forms one 10-bit-vector component. Figure 2 shows how these bits are set for the value 124. Bit-1 of the component 3, bit-2 of the component 2 and bit-4 of the component 1 are set to 1, and other bits are set to 0. The notation, $b^{\mathbf{i}}_j$, denotes the j-th bit vector of the component $\mathbf{i}$, and the above index is expressed by $I_{<10,10,10>}$. To evaluate a single value selection using $I_{<10,10,10>}$, *e.g.*, "$A = 124$", $\mathbf{b}^{\mathbf{3}}_1$, $\mathbf{b}^{\mathbf{2}}_2$ and $\mathbf{b}^{\mathbf{1}}_4$ are read and AND-ed to get the final bitmap for data retrieval.

	A		$b^{\mathbf{3}}_9$	$b^{\mathbf{3}}_8$	$b^{\mathbf{3}}_7$	$\cdots$	$b^{\mathbf{3}}_2$	$b^{\mathbf{3}}_1$	$b^{\mathbf{3}}_0$	$b^{\mathbf{2}}_9$	$b^{\mathbf{2}}_8$	$\cdots$	$b^{\mathbf{2}}_4$	$b^{\mathbf{2}}_3$	$b^{\mathbf{2}}_2$	$b^{\mathbf{2}}_1$	$b^{\mathbf{2}}_0$	$b^{\mathbf{1}}_9$	$b^{\mathbf{1}}_8$	$\cdots$	$b^{\mathbf{1}}_5$	$b^{\mathbf{1}}_4$	$b^{\mathbf{1}}_3$	$b^{\mathbf{1}}_2$	$b^{\mathbf{1}}_1$	$b^{\mathbf{1}}_0$
	$\cdots$																									
	124		0	0	0	$\cdots$	0	**1**	0	0	0	$\cdots$	0	0	**1**	0	0	0	0	$\cdots$	0	**1**	0	0	0	0
	$\cdots$																									
			component 3							component 2								component 1								

Figure 2: A Bit-Sliced Index on Attribute A with Base 10

In addition to *equality bit-encoding* (*i.e.*, the bit is set if the equality is satisfied), the bitmaps can be *range bit-encoded.* That is, for the attribute value 124, if the bitmaps are $\leq$-encoded, then all bit vectors, $b^{\mathbf{3}}_i$, of component 3 where $i \leq 1$ are set to 1, and so on, as Figure 3 shows. Since for the decimal base, all digits are less than or equal to 9, bit-9 of all components are all set and can therefore be ignored.

	A		$b^{\mathbf{3}}_8$	$b^{\mathbf{3}}_7$	$\cdots$	$b^{\mathbf{3}}_3$	$b^{\mathbf{3}}_2$	$b^{\mathbf{3}}_1$	$b^{\mathbf{3}}_0$	$b^{\mathbf{2}}_8$	$b^{\mathbf{2}}_7$	$\cdots$	$b^{\mathbf{2}}_4$	$b^{\mathbf{2}}_3$	$b^{\mathbf{2}}_2$	$b^{\mathbf{2}}_1$	$b^{\mathbf{2}}_0$	$b^{\mathbf{1}}_8$	$b^{\mathbf{1}}_7$	$b^{\mathbf{1}}_6$	$b^{\mathbf{1}}_5$	$b^{\mathbf{1}}_4$	$b^{\mathbf{1}}_3$	$b^{\mathbf{1}}_2$	$b^{\mathbf{1}}_1$	$b^{\mathbf{1}}_0$
	$\cdots$																									
	124		0	0	$\cdots$	0	0	**1**	**1**	0	0	$\cdots$	0	0	**1**	**1**	**1**	0	0	0	0	**1**	**1**	**1**	**1**	**1**
	$\cdots$																									
			component 3							component 2								component 1								

Figure 3: A Bit-Sliced Index on Attribute A with Base 10 and Range Bit-Encoding

[3]Later in this paper, we do not explicitly express the necessity of an *existing* bitmap, unless the existing bitmap is not needed.

The choice of the base affects the space requirements and the performance of the index in query processing. For the above example, the binary bit-sliced index on A has 16 bitmaps, while the 10-based bit-sliced index with equality bit-encoding consists of 30 bitmaps. To evaluate a selection predicate, such as "$A = 124$", requires 16 bitmap scans using the binary bit-sliced index, but only 3 bitmap scans with a decimal base bit-sliced index. Generally speaking, for bit-sliced indexes with uniform base, as the magnitude of the base increases, the index requires more space but performs better. Other query types are defined and discussed in section 2.2.

The base of bit-sliced indexes can be non-uniform. Non-binary, uniform-based bit-sliced indexes are often less efficient in both space and time than non-uniform-based indexes with the same number of components. For example, the smallest base for a 4-component uniform-based bit-sliced index on A is $< 6,6,6,6 >$. Using non-uniform bases, the index with the base $< 5,6,6,6 >$ requires less space than $I_{<6,6,6,6>}$, and the index with the base $< 2,8,8,8 >$ requires less bitmap scans than $I_{<6,6,6,6>}$ in query processing[4]. In [3], theorems are defined as guidelines for finding bases of either *time optimal* or *space optimal* n-component bit-sliced indexes. They are defined as follows:

Space-optimum Given an integer n, the space-optimal n-component bit-sliced index is the n-component bit-sliced index with the base $< \overbrace{b-1,\ldots,b-1}^{n-r},\overbrace{b,\ldots,b}^{r} >$, where $b = \lceil \sqrt[n]{|A|} \rceil$ and r is the smallest positive integer such that $b^r(b-1)^{n-r} \geq |A|$.

Time-optimum Given an integer n, the time-optimal n-component bit-sliced index is the index with the base $< \underbrace{2,\ldots,2}_{n-1}, \lceil \frac{|A|}{2^{n-1}} \rceil >$.

Three points need to be stated here. First, the optimal time is calculated based on the index processing algorithms proposed in [3, 14]. Second, the optimal time is only true for a subset of selection types, (*continuous range selections*), which will be discussed further in section 3. Third, the optimal space and time defined above are subject to a given number of components, *i.e.*, they describe a *local optimum*. For example, a 3-component space-optimal index might perform better than a 5-component time-optimal index with respect to both time and space, or vice versa. Globally, for all bit-sliced indexes, those with binary uniform bases, $I_{<\cdots,2,2>}$, are space optimal; in contrast, those with the base equal to the cardinality of the indexed attribute, $I_{<|A|>}$, are time optimal. However, the *global time optimum* (with or without space constraints), which is the main interest of query optimization, requires further classification of query types and performance analysis of query evaluation algorithms. These are discussed in detail in section 3.

[4]Both $< 5,6,6,6 >$ and $< 2,8,8,8 >$ are *well-defined* bases[3]. A well-defined base, $< b_n,\ldots,b_1 >$, consists of finite number of components, *i.e.*, $n \in \mathbb{Z}^+$, such that $b_n = \lceil |A|/(\prod_{i=1}^{n-1} b_i) \rceil$. In this paper, we discuss only bit-sliced indexes with well-defined bases.

2.1.2 Encoded bitmap indexes

From the above discussion, we see that there exists a dilemma between space and time in the design of bit-sliced indexes. Another variation of bitmap indexing, *encoded bitmap indexing* (EBI) proposed in [18], provides possibilities for solving such a dilemma between space and time, with help of *encoding functions*.

EBI applies an *encoding function* on the attribute domain and builds a binary-based bit-sliced index on the encoded domain. Through its binary base and domain encoding, EBI minimizes the space requirement. At the same time, it provides possibilities for query optimization through well-defined encoding (discussed later).

For example, to build an EBI on attribute B of Example 1, we define an encoding function, $\mathbb{M}^B : B \rightarrow \{\langle b_3b_2b_1b_0\rangle | b_i \in \{0,1\}, 0 \leq i \leq 3\}$. The function $\mathbb{M}^B$ maps the domain of B onto a set of four-bit numbers, as Figure 4(a) shows. The number of bits is determined by $\lceil \log_2(|B| + 2) \rceil$. (The addition of 2 is due to the inclusion of *non-existing tuples* and Nulls in the attribute domain.)

B	$\mathbf{b}_3$	$\mathbf{b}_2$	$\mathbf{b}_1$	$\mathbf{b}_0$
a	0	0	1	0
c	0	1	1	0
f	0	1	0	1
t	1	1	1	1
v	1	1	0	1

Keys	Encoding	Keys	Encoding
void	$0000_{(2)}$	e	$0100_{(2)}$
NULL	$0001_{(2)}$	f	$0101_{(2)}$
a	$0010_{(2)}$	t	$1111_{(2)}$
b	$0011_{(2)}$	u	$1110_{(2)}$
c	$0110_{(2)}$	v	$1101_{(2)}$
d	$0111_{(2)}$	w	$1100_{(2)}$

(a) Mapping table

$$\begin{array}{llll}
f_{void} = \bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0 & f_b = \bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\mathbf{b}_1\mathbf{b}_0 & f_e = \bar{\mathbf{b}}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0 & f_u = \mathbf{b}_3\mathbf{b}_2\mathbf{b}_1\bar{\mathbf{b}}_0 \\
f_{\mathsf{NULL}} = \bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\mathbf{b}_0 & f_c = \bar{\mathbf{b}}_3\mathbf{b}_2\mathbf{b}_1\bar{\mathbf{b}}_0 & f_f = \bar{\mathbf{b}}_3\mathbf{b}_2\bar{\mathbf{b}}_1\mathbf{b}_0 & f_v = \mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1\mathbf{b}_0 \\
f_a = \bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\mathbf{b}_1\bar{\mathbf{b}}_0 & f_d = \bar{\mathbf{b}}_3\mathbf{b}_2\mathbf{b}_1\mathbf{b}_0 & f_t = \mathbf{b}_3\mathbf{b}_2\mathbf{b}_1\mathbf{b}_0 & f_w = \mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0
\end{array}$$

(b) Retrieval min-terms

Figure 4: An Example of Encoded Bitmap Indexing on B

Based upon the mapping table, four bitmaps and a set of retrieval Boolean functions are defined. A retrieval Boolean function (or *min-term*[5]) is defined for each attribute value, based on the encoded values. One bit corresponds to one Boolean variable, and "0" bits are expressed by the negation of the variables. For example, the min-term for value a is $\bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\mathbf{b}_1\bar{\mathbf{b}}_0$, which corresponds to its encoded value $0010_{(2)}$. Four bitmaps of length equal to the cardinality of the indexed table are constructed as follows. For any tuple in the indexed table with offset j, all j-th bits in all bitmaps $\mathbf{b}_i$ ($i = 0,\ldots,3$) are assigned its encoded value $\langle b_3b_2b_1b_0\rangle$, *e.g.*, for all tuples with $B = a$, $\mathbf{b}_3 = 0$, $\mathbf{b}_2 = 0$, $\mathbf{b}_1 = 1$ and $\mathbf{b}_0 = 0$.

Using the EBI to evaluate a selection, *e.g.*, "$B \in \{a,b,c,d\}$", the retrieval min-terms of selected values

[5]A min-term of n Boolean variables is a logical conjunction of all n variables, or their negations.

are selected to form a retrieval Boolean expression[6], *i.e.*, $f_a + f_b + f_c + f_d$, which is further reduced to $\bar{\mathbf{b}}_3\mathbf{b}_1$. That is, AND*ing* the negation of $\mathbf{b}_3$ to $\mathbf{b}_1$, the 1's bits indicate those tuples satisfying the selection condition.

In principle, an EBI is a binary bit-sliced index defined on the encoded attribute domain. EBIs have two advantages over binary bit-sliced indexes. *First*, the number of bit vectors of an EBI is no larger than that of bit-slices, since the number of bit vectors of an EBI is decided by the logarithmic of the indexed attribute's cardinality of base 2, while the number for binary bit-slices is decided by the size of the indexed attribute's data type. Furthermore, for deleted tuples, bit-sliced indexes need an extra bit vector for the existing tuples, while defining "*non-existing*" as *void*, 0, eliminates the extra processing of the bit vector E [18]. It reduces not only the space requirement but also the processing time.

Second, EBIs have more optimization potential than binary bit-slices. Through some *proper* encoding functions, the number of bit vectors accessed in query processing can be reduced to one, while all the bit vectors, including the bit vector E, in bit-sliced indexes must be read. Cases for *proper* functions in specific application environments in DWs are discussed in [18].

So far, we have discussed the two major approaches to variate bitmap indexes. In the next subsection, we introduce the cost models to evaluate index performance.

2.2 Cost models for performance analysis

Two metrics are used to evaluate the performance of an index in this paper: *space* and *time*. Both are denoted in terms of the number of bitmaps. We use $space(I)$ to denote the space requirement of the index I, and $time(I)$ to denote the number of bitmap scans for a selection predicate evaluation using I. We discuss in this paper only the cost of selection operations, and selection predicates are defined by "$A\ op\ V$", where A is a single attribute, $op \in \{<, \leq, >, \geq, =, \neq, \in, \notin\}$ and V is a single value for $op \in \{<, \leq, >, \geq, =, \neq\}$ or a set of values for $op \in \{\in, \notin\}$.

In the following discussion, we must distinguish the *time*-function of evaluating *continuous range operators* ($\{<, \leq, >, \geq, =, \neq\}$)[7] from that of evaluating *discrete range operators* ($\{\in, \notin\}$). Therefore, we define $time^C(I)$ and $time^D(I)$ to denote the *time*-functions of the index I for evaluating continuous and discrete selection predicates, respectively. A global *time*-function is defined by the weighted sum of $time^C(I)$ and $time^D(I)$:

$$time(I) = \rho \cdot time^C(I) + (1-\rho) \cdot time^D(I), \qquad (1)$$

where ρ is the probability of occurrence of continuous range predicates.

[6]In this paper, we use + to denote the logical operator OR and · to denote the logical operator AND.

[7]Without loss of generality, we consider *equality operators* $\{=, \neq\}$ as continuous range operators. This does not affect the result of performance analysis.

3 Static query optimization

Query optimization using bitmap indexing can be achieved by two different, complementary approaches. *Static optimization* is a design-time optimization. It includes optimal index design and improved algorithms for index processing. *Dynamic optimization* is performed at run-time. It is achieved by strategies which exploit run-time information, such as constraints, statistics, or distribution information of underlying data, to determine a better execution plan.

3.1 Index Design: time optimal index

The following discussion concentrates on finding time optimal indexes for different types of range selections. We propose two algorithms for evaluating discrete range selections and define their time functions. Based upon the time functions and the time functions introduced in [3], we define a new design criterion of global time optimal indexes for both types of selections, with and without space constraints. Finally, we explore the design issues involving EBI.

Continuous range selections In [3], the algorithm, `RangeEval-Opt`, for evaluation of continuous range selection predicates using bit-sliced indexes was proposed. Based upon this algorithm, the time-function of an n-component bit-sliced index I with base $<b_n, \ldots, b_1>$ and range bit-encoding was derived as

$$\begin{aligned} time^C(I) &= 2(n - \sum_{i=1}^{n} \frac{1}{b_i} + \frac{1}{3}(\frac{1}{b_1} - 1)) &(2)\\ &\approx 2n - \theta, \text{for the worst case,} &(3)\end{aligned}$$

where $\theta = 1$, if the range operator is one of $\{<, \leq, >, \geq\}$, and $\theta = 0$, if the range operator is one of $\{=, \neq\}$.

From Equation (3), we can see that the time efficiency of a bit-sliced index is getting worse, as the number of components, n, increases. In principle, *the fewer components a bit-sliced index has, the more time-efficient the index is.*

However, the above time efficiency analysis is only true, if the range operators are confined to the set $\{<, \leq, >, \geq, =, \neq\}$, *i.e.*, continuous range selections. Anomalies arise for discrete range selections. Although discrete range predicates ($A \in V$, or $A \notin V$) could be replaced by a disjunction of equality selection predicates ($A = v$), it would be very inefficient to evaluate discrete range predicate in such a way using the algorithm `RangeEval-Opt`. Therefore, it is unreasonable to use the time function based on `RangeEval-Opt` to analyze the performance of the index in answering discrete range selections.

Suppose that a bit-sliced index, $I_{<10,10,10>}$, on attribute A using a range bit-encoding scheme is defined. For the selection predicate "$A \in \{864, 764\}$", 12 bitmap scans are required using `RangeEval-Opt`, since `RangeEval-Opt` treats and evaluates each value in the operand set separately, namely $A = 864$ OR $A =$

764. Six bitmap scans are required for evaluating each of the values. Obviously, discrete range predicates can be evaluated more efficiently by simply avoiding reading the same bitmap more than once. In the above example, the bitmaps for evaluating the two least significant digits — "6" and "4" are read twice using `RangeEval-Opt`.

In order to define a reasonable time-function for evaluating discrete range predicates, we first develop two algorithms for evaluating discrete range predicates using bit-sliced indexes, and then define the time-function, $\mathit{time}^D(I)$, based upon them.

Discrete range selections Algorithms 1 and 2 evaluate discrete range predicates using bit-slices with range bit-encoding and equality bit-encoding, respectively. The basic idea of these algorithms is to avoid rescanning the same bit vector for consequent equality tests. Every required bitmap is scanned exactly once, and all the comparisons involving the memory-resident bit-segments are performed at a time.

The algorithms work as follows: Before evaluating the predicate "$A \in \mathbb{V}$", the values in the set $\mathbb{V}$ are parsed once to examine what bitmaps are required for the evaluation. Then, all the required bitmaps are read into the buffer, and the algorithms loop for each value in $\mathbb{V}$ and perform the equality comparisons digit by digit. In reality, the total size of required bitmaps might not fit into memory, therefore an implementation of the algorithms might loop for the reading of bitmaps and comparisons page by page.

Let us define the time function of the algorithms now. For evaluation of each distinguished digit in each component, Algorithm 1 reads two bit vectors (except one bit vector for 0 or $v_{j,i}-1$), while Algorithm 2 reads only one bit vector for each digit. For example, to evaluate the equality-test involving the digit "6" of the component-2, the bitmaps, $\mathbf{b}_6^\mathbf{2}$ and $\mathbf{b}_5^\mathbf{2}$, are read in Algorithm 1, and $\mathbf{b}_6^\mathbf{2}$ is read in Algorithm 2.

Generally speaking, to evaluate the digit, α, in component i, the bit vector representing α in the i-th component, $\mathbf{b}_\alpha^\mathbf{i}$, is read. (For range bit-encoded indexes, $\mathbf{b}_{\alpha-1}^\mathbf{i}$, is also read, if $0 < \alpha < b_i - 1$.) Assuming that the digits, $v_{j,i}$ $(1 \le j \le k)$, are evenly distributed within the range, $0 \le v_{j,i} < b_i$ $(1 \le i \le n)$, we define the time functions of Algorithms 1 and 2 as follows.

For range bit-encoding,

$$\mathit{time}^D(I) = \sum_{i=1}^{n} \sum_{j=0}^{k-1} \frac{1}{C_k^{b_i+k-1}} \left(\min(2(k-j), b_i) \cdot C_j^{k-1} \cdot C_{k-j}^{b_i} \right), \quad (4)$$

and for equality bit-encoding,

$$\mathit{time}^D(I) = \sum_{i=1}^{n} \sum_{j=0}^{k-1} \frac{1}{C_k^{b_i+k-1}} \left((k-j) \cdot C_j^{k-1} \cdot C_{k-j}^{b_i} \right), \quad (5)$$

where $C_b^a = \frac{a!}{b! \cdot (a-b)!}$ denotes the number of combinations of choosing b from a.

For the worst cases, or with a large number of k (the cardinality of the selection range) both Algorithm 1 and 2 read all the bit vectors of the index. In such cases, the *time*-function for discrete range evaluation is defined by its *space*-function.

$$\mathit{time}^D(I) = \mathit{space}(I) = \begin{cases} n & , \text{ if } b_i = 2, \ 1 \le i \le n \\ \sum_{i=1}^{n} (b_i - \theta), & \text{if } b_i > 2, \ 1 \le i \le n \end{cases} \quad (6)$$

where $\theta = 0$ for equality bit-encoding, and $\theta = 1$ for range bit-encoding. As a result, the n-component time optimal indexes for discrete selections are those n-component space optimal indexes.

Global time optimal indexes The above discussion shows that conflicting index design criteria exist for continuous and discrete range selections. Since most attributes might be involved in both types of selections, it is a dilemma to choose either of the design criteria. One straightforward solution is to design for each type of selections an index, *i.e.*, an index with possibly fewest components for continuous range selections, and another index with binary bases for discrete range selections.

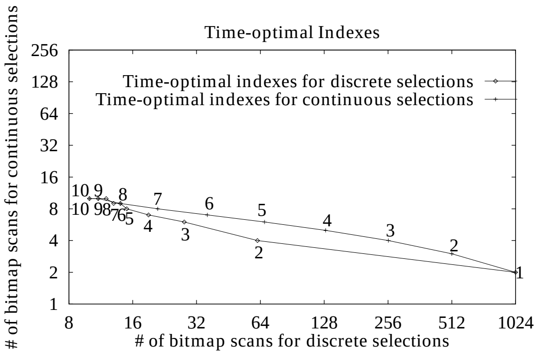


Figure 5: Time-Optimal Indexes, $|A| = 1024$

A better approach is to find a global time optimal index for both types of selections. In Figure 5, time optimal indexes for either continuous selections or discrete selections on an attribute A with $|A| = 1024$ are illustrated. A point on a curve represents an index (labeled with the number of components), the x-distance denotes the time of evaluating a discrete selection predicate using the index, *i.e.*, $\mathit{time}^D(\cdot)$, and the y-distance denotes the time of evaluating a continuous selection predicate using the index, *i.e.*, $\mathit{time}^C(\cdot)$. The computation of the time functions is based on Equations (2) and (6). As Figure 5 shows, choosing an index which reduces the time of evaluating continuous selection types will increase the time of evaluating discrete ones, and vice versa. In order to find the global time optimal index, we define the following time function:

$$\rho \cdot \mathit{time}^C(I) + (1-\rho) \cdot \mathit{time}^D(I), \ I \in \mathcal{I},$$

where $\mathcal{I}$ is the universal set of bit-sliced indexes, and ρ is the probability of occurrence of continuous range selections. The minimum of the function, which is defined as the *break-even* point, characterizes the global time optimal index for both types of selections.

Algorithm 1 (Discrete Selection, $\leq$-encoded)

```
Input:  A bit-sliced index with the base, $< b_n, \ldots, b_1 >$, where $n$ is
        the number of components and $\mathbf{b}^{\mathbf{i}}_j$ denotes the $j$-th bit vector
        of $i$-th component.
        Selection predicate $A \in \mathbb{V}$, where $\mathbb{V} = \{v_1, \ldots, v_k\}$, where each
        value $v_j$ $(1 \leq j \leq k)$ is represented as $v_{j,n} \cdots v_{j,1}$ $(0 \leq v_{j,i} <
        b_i, 1 \leq i \leq n)$.
Output: A bitmap vector representing the set of tuples which satisfy
        the range-selection predicate, $A \in \mathbb{V}$.
Begin
1)  $\mathbb{B} = 1$ and $\mathbb{B}^r = \emptyset$
2)  initialize $n$ arrays of bits $M_i[b_i - 1]$,
        $1 \leq i \leq n$, /* $M_i[0] \cdots M_i[b_i - 2]$ */
3)  for $i = 1$ to $n$
4)      for $j = 1$ to $k$
5)          if $(v_{j,i} < b_i - 1)$ then $M_i[v_{j,i}] = 1$
6)          if $(v_{j,i} > 0)$ then $M_i[v_{j,i} - 1] = 1$
7)  for $i = 1$ to $n$
8)      for $j = 0$ to $b_i - 2$
9)          if $(M_i[j] = 1)$ then read $\mathbf{b}^{\mathbf{i}}_j$
10) for $j = 1$ to $k$
11)     for $i = 1$ to $n$
12)         if $(v_{j,i} = b_i - 1)$ then $\mathbb{B} = \mathbb{B} \cdot (\overline{\mathbf{b}^{\mathbf{i}}_{v_{j,i}-1}})$
13)         else if $(v_{j,i} = 0)$ then $\mathbb{B} = \mathbb{B} \cdot (\mathbf{b}^{\mathbf{i}}_{v_{j,i}})$
14)         else $\mathbb{B} = \mathbb{B} \cdot (\mathbf{b}^{\mathbf{i}}_{v_{j,i}} \cdot \overline{\mathbf{b}^{\mathbf{i}}_{v_{j,i}-1}})$ /* $0 < v_{j,i} < b_i - 1$ */
15)     $\mathbb{B}^r = \mathbb{B}^r + \mathbb{B}$
16) return $\mathbb{B}^r$ /* filter out non-existing tuples before return */
End.
```

Algorithm 2 (Discrete Selection, $=$-encoded)

```
Input:  A bit-sliced index with the base, $< b_n, \ldots, b_1 >$, where $n$ is
        the number of components and $\mathbf{b}^{\mathbf{i}}_j$ denotes the $j$-th bit vector
        of $i$-th component.
        Selection predicate $A \in \mathbb{V}$, where $\mathbb{V} = \{v_1, \ldots, v_k\}$, where each
        value $v_j$ $(1 \leq j \leq k)$ is represented as $v_{j,n} \cdots v_{j,1}$ $(0 \leq v_{j,i} <
        b_i, 1 \leq i \leq n)$.
Output: A bitmap vector representing the set of tuples which satisfy
        the range-selection predicate, $A \in \mathbb{V}$.
Begin
1)  $\mathbb{B} = 1$ and $\mathbb{B}^r = \emptyset$
2)  initialize $n$ arrays of bits $M_i[b_i]$,
        $1 \leq i \leq n$, /* $M_i[0] \cdots M_i[b_i - 1]$ */
3)  for $i = 1$ to $n$
4)      for $j = 1$ to $k$
5)          $M_i[v_{j,i}] = 1$
6)  for $i = 1$ to $n$
7)      for $j = 0$ to $b_i - 1$
8)          if $(M_i[j] = 1)$ then read $\mathbf{b}^{\mathbf{i}}_j$
9)  for $j = 1$ to $k$
10)     for $i = 1$ to $n$
11)         $\mathbb{B} = \mathbb{B} \cdot \mathbf{b}^{\mathbf{i}}_{v_{j,i}}$
12)     $\mathbb{B}^r = \mathbb{B}^r + \mathbb{B}$
13) return $\mathbb{B}^r$ /* filter out non-existing tuples before return */
End.
```

Note that in our example, for the sake of clarity, we confine $\mathcal{I}$ to the set of $I^C \cup I^D$, where I^C (I^D) denotes the time optimal indexes for continuous (discrete) selections. As a matter of fact, this restriction provides an efficient way to find an approximate optimal solution, since I^C and I^D describe boundaries of the bit-sliced index space. One describes the optimum for continuous selection evaluation, and the other describes the optimum for discrete selection evaluation.

The two functions — $\rho \cdot time^C(I^C) + (1-\rho) \cdot time^D(I^C)$ and $\rho \cdot time^C(I^D) + (1-\rho) \cdot time^D(I^D)$ — are plotted in Figure 6(a), and the minimum point of the curves, $(10, 10)$, is the break-even point. That means a 10-component bit-sliced index on attribute A ($|A| = 1024$) is optimized for both types of selections. (At the point $n = 10$, the index is a a binary uniform bit-sliced index with ten components.)

Figure 6(b) reveals the curves with $\rho = 0.75$, *i.e.*, 75% of the selections are continuous. The point $(5, 9.75)$ is the break-even point, which means the 5-component time optimal index for discrete selections is also the global time optimal index.

Global time optimal indexes under space constraint Under a space constraint M, the global time optimal index is defined by

$$\min\Big(\rho \cdot time^C(I) + (1-\rho) \cdot time^D(I)\Big),\ I \in \mathcal{I}',$$

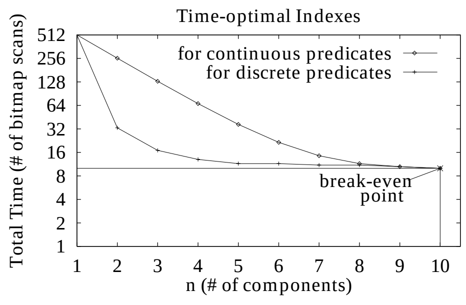
where $\mathcal{I}' = \{I | space(I) \leq M\}$.

To avoid the exhaustive search in the whole index space, we also confine the search space to the boundaries, *i.e.*, find the smallest n and n', such that $space(I^C_n) \leq M$ and $space(I^D_{n'}) \leq M$, and let $\mathcal{I}'' = \{I^C_i | i \geq n\} \cup \{I^D_j | j \geq n'\}$. I^C_n and $I^D_{n'}$ denote the n-component time optimal index for the continuous selection type and the n'-component time optimal index for the discrete selection type, respectively. Then, the approximate global time optimal indexes under the space constraint M is given by $\min\left(\rho \cdot time^C(I) + (1-\rho) \cdot time^D(I)\right),\ I \in \mathcal{I}''$.

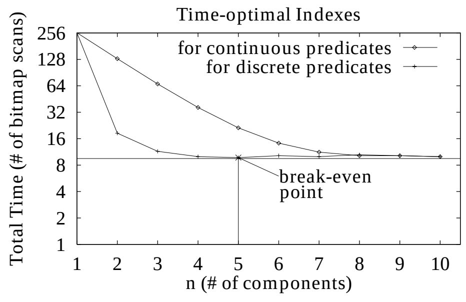
Following the example in Figure 6(b), suppose that $M = 50$ bitmaps, *i.e.*, maximal 50 bitmaps can be stored in the system, then $n' = 3$ and $n = 6$, since $space(I^D_3) = 28$ and $space(I^C_6) = 36$. Figure 7 shows that among the bitmap indexes in $\{I^C_i | i \geq 6\} \cup \{I^D_j | j \geq 3\}$, the minimal point is $(5, 9.75)$.

Domain encoding An EBI is a binary bit-sliced index on the encoded attribute domain. In principle, all bit vectors of an EBI must be read in a query evaluation. However, for a pre-defined set of selections, performance of EBIs can be improved through *well-defined encoding* [18]. With a well-defined encoding, the number of bitmap scans in query processing is minimized, while the space requirement of the index is unchanged. Let us demonstrate how it works.

Let an EBI be defined on attribute A of Example 1 as Figure 8 shows, and a frequently asked selection predicate is given as "$100 \leq A \leq 107$". To evaluate the selection, the retrieval min-terms for all the values in the selection range are taken and form a logical disjunction of min-terms — $f_{100}+f_{101}+f_{102}+f_{103}+f_{104}+f_{105}+f_{106}+f_{107}$, which can be further reduced to $\bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$. That is, for evaluating "$100 \leq A \leq 107$", instead of 10



(a) $\rho = 0.5$



(b) $\rho = 0.75$

Figure 6: Global Time Optimal Indexes

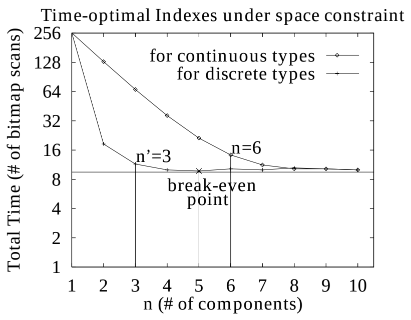


Figure 7: Global Time Optimal under Space Constraints

bitmap scans, seven bitmaps are accessed.

The encoding function shown in Figure 8 is a well-defined encoding subject to the selection predicate "$100 \leq A \leq 107$". There does not exist any other encoding function which can result in a reduction of more than three variables for logical disjunction of eight min-terms.

We can see that in an environment where selection patterns can be predefined, by a well-defined encoding we improve the time efficiency of an EBI in query processing without sacrifice of space efficiency. However, finding a well-defined encoding is an NP-problem [18]. The involvement of human experts in the index design phase is required. In [18], examples of well-defined encoding for some typical DW operations are given, including *hierarchical encoding* and *range based encoding*.

A quick and dirty, but efficient, variation of finding a well-defined encoding is to define extra bit vectors

Keys	Encoding	Keys	Encoding
void	$0000000000_{(2)}$	103	$0101001100_{(2)}$
NULL	$0000000001_{(2)}$	104	$0101010000_{(2)}$
100	$0101000000_{(2)}$	105	$0101010100_{(2)}$
101	$0101000100_{(2)}$	106	$0101011000_{(2)}$
102	$0101001000_{(2)}$	107	$0101011100_{(2)}$
...	...	...	...

(a) Mapping Table

$f_{void} = \bar{\mathbf{b}}_9\bar{\mathbf{b}}_8\bar{\mathbf{b}}_7\bar{\mathbf{b}}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_4\bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$ $\quad f_{103} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_4\mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$

$f_{\mathsf{NULL}} = \bar{\mathbf{b}}_9\bar{\mathbf{b}}_8\bar{\mathbf{b}}_7\bar{\mathbf{b}}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_4\bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\mathbf{b}_0$ $\quad f_{104} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\mathbf{b}_4\bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$

$f_{100} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_4\bar{\mathbf{b}}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$ $\quad f_{105} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\mathbf{b}_4\bar{\mathbf{b}}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$

$f_{101} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_4\bar{\mathbf{b}}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$ $\quad f_{106} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\mathbf{b}_4\mathbf{b}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$

$f_{102} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\bar{\mathbf{b}}_4\mathbf{b}_3\bar{\mathbf{b}}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$ $\quad f_{107} = \bar{\mathbf{b}}_9\mathbf{b}_8\bar{\mathbf{b}}_7\mathbf{b}_6\bar{\mathbf{b}}_5\mathbf{b}_4\mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0$

$\cdots \qquad \cdots$

(b) Retrieval Min-terms

Figure 8: A Well-Defined Encoding for $100 \leq A \leq 107$

for frequently asked selections. The resulting bitmap is precomputed and stored in the system. This approach increases the space requirement of EBIs, but since EBIs are space optimal bit-sliced indexes, they have more elbowroom for such space expansion.

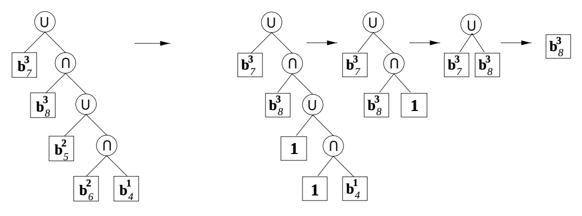
For the *worst cases*, the time function of an EBI (without considering the effect of well-defined encoding and logical reduction, discussed later) is the same as that of a binary bit-sliced index $time(EBI) = space(EBI) = n$, where $n = \lceil \log_2 |A| \rceil$ and A denotes the indexed attribute.

3.2 Algorithm design

In this subsection, we discuss static query optimization through better design of evaluation algorithms.

Tree reduction for `RangeEval-Opt` In [3], an algorithm — `RangeEval-Opt`, has been proposed to improve its former version proposed in [14]. Here we introduce a further improvement of `RangeEval-Opt` by the *execution tree reduction*.

Taking the example in [3], a 3-component range ($\leq$) bit-encoded bit-sliced index with decimal base on attribute A and the predicate "$A \leq 864$" are given. Using Algorithm `RangeEval-Opt` to evaluate the predicate results in the execution tree as shown in Figure 9(a).

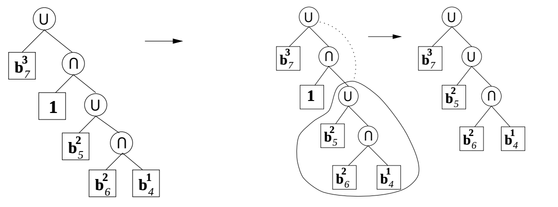


(a) the original tree (b) the reduced tree

Figure 9: Transformation of Execution Tree for $A \leq 864$, $\cup$ denotes OR and $\cap$ denotes AND

Suppose that for a certain running state of a database, the second digit of all the values of A is no larger than 5, *i.e.*, in the component-2 of the index, the bit vectors $\mathbf{b}_5^2$, $\mathbf{b}_6^2$, $\mathbf{b}_7^2$ and $\mathbf{b}_8^2$ are all set to "1". By replacing the corresponding bit vectors with "**1**" vectors, we have the first tree in Figure 9(b). By applying $x \cdot 1 = x$ (identity law) and $x + 1 = 1$ (dominance law) of Boolean algebra, the execution tree is reduced down to one node, *i.e.*, instead of 5 bitmap scans plus 4 logical operations (the tree in Figure 9(a)), only 1 bitmap (the bit vector $\mathbf{b}_8^3$ of component-3) is read.

The reduction of a partial sub-tree is depicted next. Following the above example, if the most significant digit of A is no larger than 8, then the original execution tree can be replaced by the tree in Figure 10(a). Applying the rule $x \cdot 1 = x$, we can connect the right sibling subtree of the **1**-node directly to the root, as shown in Figure 10(b). We reduce the computation complexity from 5 bitmap scans plus 4 logical operators to 4 bitmap scans and 3 logical operators. For equality predicates $(=, \neq)$, the idea also works. Because of the space limit, we do not provide a worked out example.



(a) execution tree – $A \leq 864$ (b) reduction of the execution tree

Figure 10: Transformation of Execution Tree for $A \leq 864$, ∪ denotes OR and ∩ denotes AND

Obviously, in order to be able to apply such a reduction, the information about the percentage of population of each bit vector is needed. Without much extra cost, this information can be computed at the time of index creation and can be synchronized every time the data are uploaded in batch mode to the data warehouse. The revised version of `RangeEval-Opt`, which suppresses unnecessary bitmap scans as described above, is defined in Algorithm 3.

Algorithm 3 (Range evaluation with tree-reduction)

Input: A bit-sliced index with the base, $< b_n, \ldots, b_1 >$, where n is the number of components and $\mathbf{b}_j^i$ denotes the j-th bit vector of i-th component. For a bit vector, $\mathbf{b}_j^i$, $\Theta(\mathbf{b}_j^i)$ denotes the percentage of "1"s in $\mathbf{b}_j^i$.
Selection predicate $A\ op\ v$, where $op \in \{<, >, \leq, \geq, =, \neq\}$.

Output: A bitmap vector representing the set of tuples which satisfy the selection predicate, $A\ op\ v$.

```
Begin
    $\mathbb{B} = 1$
    if ($op \in \{<, \geq\}$) then $v = v - 1$
    $v = v_n v_{n-1} \cdots v_1$
    if ($op \in \{<, >, \leq, \geq\}$) then
        if ($v_1 < b_1 - 1$) and ($\Theta(\mathbf{b}_{v_1}^1) \neq 1$) then $\mathbb{B} = \mathbf{b}_{v_1}^1$
        for $i = 2$ to $n$
            if ($v_i \neq b_i - 1$) and ($\Theta(\mathbf{b}_{v_i}^i) \neq 1$) then $\mathbb{B} = \mathbb{B} \cdot \mathbf{b}_{v_i}^i$
            if ($v_i \neq 0$) and ($\Theta(\mathbf{b}_{v_i-1}^i) \neq 1$) then $\mathbb{B} = \mathbb{B} + \mathbf{b}_{v_i-1}^i$
    else
        for $i = 1$ to $n$
            switch ($v_i$)
            case $v_i = 0$:
                if ($\Theta(\mathbf{b}_0^i) \neq 1$) then $\mathbb{B} = \mathbb{B} \cdot \mathbf{b}_0^i$
            case $v_i = b_i - 1$:
                if ($\Theta(\mathbf{b}_{v_i-1}^i) \neq 1$) then $\mathbb{B} = \mathbb{B} \cdot \overline{\mathbf{b}_{v_i-1}^i}$
                else return ($\mathbb{B} = \emptyset$)
            case $0 < v_i < b_i - 1$:
                if ($\Theta(\mathbf{b}_{v_i}^i) \neq 1$) and ($\Theta(\mathbf{b}_{v_i-1}^i) \neq 1$) then
                    $\mathbb{B} = \mathbb{B} \cdot (\mathbf{b}_{v_i}^i \oplus \mathbf{b}_{v_i-1}^i)$
                else if ($\Theta(\mathbf{b}_{v_i-1}^i) = 1$) then return ($\mathbb{B} = \emptyset$)
                else $\mathbb{B} = \mathbb{B} \cdot \overline{\mathbf{b}_{v_i-1}^i}$
            end switch
    /* filter out non-existing tuples before return */
    if ($op \in \{>, \geq, \neq\}$) then return $\overline{\mathbb{B}}$ else return $\mathbb{B}$
End.
```

Logical reduction for EBI In the last section, we introduced well-defined encoding for EBI, which improves the query performance of a predefined set of selection patterns. However, ad-hoc queries dominate in the DW environment. Given any query, EBIs transform the selection predicates into a retrieval Boolean function. Before directly evaluating the Boolean expression on bitmaps, two potential reductions should be tested: *reducing the number of Boolean variables* and *reducing the number of redundant logical operations*. Both of them can be done by performing a *logical reduction* on the retrieval Boolean functions.

Each Boolean variable corresponds to a bit vector, if one variable in the retrieval expression is reduced, the corresponding bit vector needs not be read, *i.e.*, reducing the number of bitmap scans by 1. In addition, by reducing the Boolean expression to its minimal form, redundant operations are avoided.

For example, given is a selection on attribute B of Example 1 "$B \in \{e, f, t, u, v, w\}$". Suppose that an EBI is built on B, as Figure 4 shows. The retrieval Boolean function for the above selection will be $f_e + f_f + f_t + f_u + f_v + f_w$. By applying logical reduction on it,

$$
\begin{aligned}
& f_e + f_f + f_t + f_u + f_v + f_w \\
= \quad & \bar{\mathbf{b}}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0 + \bar{\mathbf{b}}_3\mathbf{b}_2\bar{\mathbf{b}}_1\mathbf{b}_0 + \mathbf{b}_3\mathbf{b}_2\mathbf{b}_1\mathbf{b}_0 + \mathbf{b}_3\mathbf{b}_2\mathbf{b}_1\bar{\mathbf{b}}_0 \\
& + \mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1\mathbf{b}_0 + \mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1\bar{\mathbf{b}}_0 \\
= \quad & \mathbf{b}_3\mathbf{b}_2 + \bar{\mathbf{b}}_1\mathbf{b}_2 = \mathbf{b}_2(\bar{\mathbf{b}}_1 + \mathbf{b}_3)
\end{aligned}
$$

the expression is reduced to $\mathbf{b}_2(\bar{\mathbf{b}}_1 + \mathbf{b}_3)$. That is, the time complexity is reduced from 4 bitmap scans plus 32 bitwise logical operations down to 3 bitmap scans plus 3 bitwise logical operations.

Some logical reduction algorithms have been introduced in the literature, such as *ordered binary decision diagram* (OBDD), *binary decision diagram* (BDD), *tabular method* [1, 2, 8, 11, 15]. The details of how these algorithms work are beyond the scope of this paper.

4 Dynamic query optimization

So far, we have discussed optimization that can be done at design time. In this section, we introduce query optimization strategies for run time. Cost models are also defined for the cost-efficiency analysis.

4.1 Principle of inclusion and exclusion

Selection evaluation consists of two phases — *index scanning phase* and *data fetching phase*. In the index scanning phase, selection predicates are evaluated using index structures, and the results of this phase are used as access plan in the data fetching phase. The idea of the *principle of inclusion and exclusion*[8] is as follows: by using a "coarse" filtering (instead of exact matching), the time of the index scanning phase is reduced. However, the desired data as well as some undesired data is *included* in the second phase. Therefore extra tests must be performed to *exclude* the unqualified data. If the extra cost for eliminating superfluous tuples in the second phase is smaller than the time-saving in the first phase, query performance improves.

Dynamic optimization for EBI The basic idea of the inclusion/exclusion approach is to identify the conditions under which an approximate access plan in the index scanning phase will still provide a globally optimized query processing time.

We give an example to illustrate how the idea works. Given is a selection predicate "$B \in \{u, v, w\}$", and an EBI on B is defined as Figure 4 shows. The retrieval Boolean function for the predicate, $f_u + f_v + f_w$, is reduced to $\mathbf{b}_3\mathbf{b}_2(\bar{\mathbf{b}}_1 + \bar{\mathbf{b}}_0)$. (The function is illustrated using a Karnaugh Graph in Figure 11(a).) That means, 4 bitmap scans plus 5 logical operators are required in the index scanning phase.

However, if we include the value t in the selection predicate and make it "$B \in \{t, u, v, w\}$", as shown in Figure 11(b), then the corresponding retrieval function, $f_t + f_u + f_v + f_w$, is reduced to $\mathbf{b}_3\mathbf{b}_2$. The time complexity is reduced from 4 bitmap scans plus 5 logical operators down to 2 bitmap scans plus 1 logical operator.

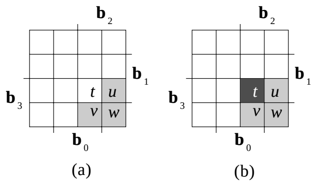


Figure 11: Karnaugh Graph of Retrieval Boolean Functions

However, the deliberate inclusion of t in the index scanning phase causes extra cost in the second phase. For the above example, if we use the resulting bitmap of $\mathbf{b}_3\mathbf{b}_2$ to fetch the data, some undesired tuples are also read, which might cost extra I/O time, and in order to filter those undesired tuples out of the final result, extra CPU time is involved. In spite of the extra cost, the inclusion/exclusion approach might still result in better query performance, due to the characteristics of *block*-I/Os and different distributions of underlying data.

In order to determine which of the two query execution plans (exact match or coarse filtering) is better, we propose the following cost model.

Cost model for inclusion-and-exclusion method We define the cost of query processing by the I/O time, *i.e.*, the total number of pages read in both index scanning and data fetching phases. Let us first define the terms and the extra data structure used in the cost model.

The *page-level* bitmap of a bitmap contains information about the distribution of the "1" bits into the logical page space of the indexed table. That means, the "1" bits in the page-level bitmap indicate those logical pages which contain the tuples represented by the original bitmap, (named *tuple-level bitmap*). Figure 12 depicts the construction of a page-level bitmap from its tuple-level bitmap. The tuple-level bitmap is divided into m p-bit segments, except the last one. Each p-bit segment of the tuple-level bitmap corresponds to one bit in the page-level bitmap. The bits in the page-level bitmap are set, if any bit in their corresponding p-bit segments is set. Other terms are defined in Table 1.

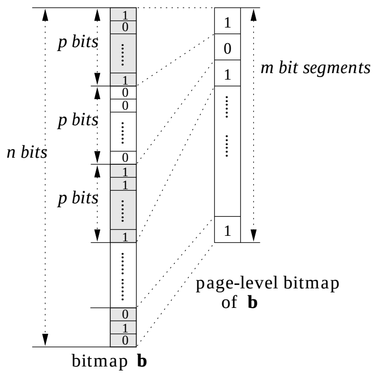


Figure 12: Transforming Tuple-ids to Data-Block-ids

Notation	Description
$\mathbf{b}$	a bit vector
$\lvert\mathbf{b}\rvert$	the number of bits in $\mathbf{b}$
$\mathfrak{T}$	a table
$\lvert\mathfrak{T}\rvert$	the cardinality of $\mathfrak{T}$, *i.e.*, $\lvert\mathbf{b}\rvert = \lvert\mathfrak{T}\rvert$
$w(\mathfrak{T})$	width of table $\mathfrak{T}$ in bytes
π	logical page size in bytes
$p = \lfloor \frac{\pi}{w(\mathfrak{T})} \rfloor$	blocking factor (the number of tuples per page for $\mathfrak{T}$)
$\mathbf{b}^p$	a *page-level* bitmap of $\mathbf{b}$ with respect to p
$\Sigma(\mathbf{v})$	the number of "1" bits in the bit vector $\mathbf{v}$

Table 1: Notations Used in the Cost Analysis

For the above example, the cost-efficiency analysis is performed as follows. Using the exact-match approach, in the index scanning phase the bitmap $\mathbb{B} = \mathbf{b}_3\mathbf{b}_2(\bar{\mathbf{b}}_1 + \bar{\mathbf{b}}_0)$ is evaluated, and it is used to access the desired

[8]The name, *the principle of inclusion and exclusion*, has been borrowed from the set theory, and it describes a different scenario here in this paper from that in the set theory.

tuples in the data fetching phase. The total I/O cost of this approach will be

$$\frac{4 \cdot |\mathbf{b}|}{8\pi} + \Sigma(\mathbb{B}^{p}) \text{ pages,} \tag{7}$$

where the first term denotes the cost of index scanning phase, and the second one denotes the cost of data fetching phase. On the other hand, using the approach of inclusion/exclusion, the bitmap $\mathbf{B} = \mathbf{b}_3\mathbf{b}_2$ is evaluated in the index scanning phase, and $\mathbf{B}$ is used to fetch data in the second phase. The total I/O cost is

$$\frac{2 \cdot |\mathbf{b}|}{8\pi} + \Sigma(\mathbf{B}^{p}) \text{ pages.} \tag{8}$$

Obviously, if Equation (8) is less than Equation (7), it is beneficial to perform the inclusion/exclusion approach.

$$\frac{2 \cdot |\mathbf{b}|}{8\pi} + \Sigma(\mathbf{B}^{p}) < \frac{4 \cdot |\mathbf{b}|}{8\pi} + \Sigma(\mathbb{B}^{p}) \implies \Sigma(\mathbf{B}^{p}) - \Sigma(\mathbb{B}^{p}) < \frac{|\mathbf{b}|}{4\pi}$$

A general form for the cost-efficiency analysis can be derived as follows.

$$\Sigma(\mathbf{b}_{\phi}^{p}) - \Sigma(\mathbf{b}^{p}) < \frac{\lambda \cdot |\mathbf{b}|}{8\pi}, \tag{9}$$

where $\mathbf{b}$ is the resulting bitmap of the exact-match approach, $\mathbf{b}_\phi$ is the resulting bitmap of the inclusion/exclusion approach, $\mathbf{b}^{p}$ and $\mathbf{b}_{\phi}^{p}$ are the page-level bitmaps of $\mathbf{b}$ and $\mathbf{b}_\phi$, respectively, and λ is the number of variables (bitmaps) which are reduced from the retrieval Boolean function after the inclusion of additional values into the selection predicate. Simply speaking, the right hand side of Inequality (9) denotes the total number of I/O-saving in pages gained through the principle of inclusion and exclusion in the index scanning phase, and the left hand side denotes the additional I/O cost of reading the extra data pages arising from the approach of inclusion/exclusion in the data fetching phase. If Inequality (9) is true, then the inclusion/exclusion approach provides a better query performance.

In practice, the calculation of the term $\Sigma(\mathbf{b}_{\phi}^{p}) - \Sigma(\mathbf{b}^{p})$ (in Inequality (9)) could be expensive and I/O intensive, since both page-level bitmaps $\mathbf{b}^{p}$ and $\mathbf{b}_{\phi}^{p}$ have to be read. In order to reduce the overhead of optimization, the term above can be replaced with an approximation using a statistic model. It can be estimated by the *expected* number of page accesses. The expected number of page accesses of a query is defined as a function of the selectivity of the query and can be computed by the following probability model.

Let n, p and k denote the total number of data pages of a selected table, the blocking factor of a page and the number of selected tuples of a query (estimated by selectivity of a query), respectively. The term — $prob(x|_{k}^{n,p})$ denotes the probability that exactly x pages are accessed subject to n, p and k. Then, the expected number of data page accesses in processing a query Q, denoted by $E(Q)$, is calculated by

$$\begin{aligned} E(Q) &= \sum_{i=1}^{n} i \cdot prob(i|_{k}^{n,p}), \text{ where} \qquad (10) \\ prob(i|_{k}^{n,p}) &= \frac{C_i^n}{C_k^{pn}}(C_k^{ip} - C_1^i \cdot C_k^{(i-1)p} + C_2^i \cdot C_k^{(i-2)p} + \\ &\quad \cdots + (-1)^i C_i^i \cdot C_k^{(i-i)p}) \end{aligned}$$

The inclusion/exclusion approach increases the selectivity of the query by including additional values into the range selection. Say, the number of selected tuples is changed to k' and the revised query is denoted by Q'. Then, Inequality (9) can be estimated by

$$E(Q') - E(Q) < \frac{\lambda \cdot |\mathbf{b}|}{8\pi} \tag{11}$$

The probability, $prob(i|_{k}^{n,p})$, is proven in [19].

<u>What to include</u> Another issue concerning the inclusion/exclusion approach is what to include. It itself is an optimization problem. Let us formally define the cardinality of a selection range first. Given an attribute A and a range selection on A, "A *op* V", the cardinality of the selection range is the cardinality of the set S, such that $S = \{v | v \in A, \text{ and } v \text{ } op \text{ } V\}$. If $V \subseteq A$, then $|S| = |V|$. Following the above example, for the selection, "$B \in \{u, v, w\}$", since $\{u, v, w\} \subset A$, the cardinality of the selection range is $|\{u, v, w\}| = 3$. For later discussion, we assume that $V \subseteq A$.

To find out *what to include*, One must answer the following questions, to avoid unnecessary attempts: *how many additional values must be included at least in the operand set, V, in order to make a further reduction?* And, *how much reduction can be achieved?*

To answer the first question, a simple test on the operand set, V, is performed. If the cardinality of the selection range, denoted by $|V|$, is even, then at least $2^i - |V|$ values must be included in V to make a further reduction possible, where i is the smallest integer such that $2^i - |V| \geq 0$. If $|V|$ is odd, then a further reduction is possible by including one additional value into V. The reason why the minimum number of additional inclusion is 1 or $2^i - |V|$ is behind the idea of making 2^j $(j \in \mathbb{Z}^+)$ neighboring cells in a Karnaugh graph, as Figure 11(b) shows. Note that this is only the necessary condition. Satisfying this condition does not imply the existence of a reduction.

To answer the second question, we should explore the relationship between the cardinalities of selection ranges and the probably minimal numbers of bitmap scans in the index scanning phase. Assuming the existence of a well-defined encoding, Table 2 lists the minimal number of bitmap scans with respect to different cardinalities of selection ranges for attribute A and $|A| = 8$. For example, if we extend the cardinality of the selection range from 3 up to 4, *for the best case*, the number of bitmap scans could drop from 3 down to 1.

The computation of the table is based on Property 3.1 in [18], which describes the following scenario: for best cases, the number of bitmap scans in processing a selection on A is $(\lceil \log_2 |A| \rceil - i) + \theta(\delta, i)$, where δ is the cardinality of the selection range and i is the largest integer, such that $\frac{\delta}{2^i} \geq 1$. The function, $\theta(\delta, i)$ is defined as follows.

$$\theta(\delta, i) = \begin{cases} 0, & \text{if } (\delta \bmod 2^i) = 0, \\ i - \gamma, & \text{if } ((\delta \bmod 2^i) \bmod 2^{\gamma+1}) = 2^\gamma, \ \gamma = 0, \ldots, i-1 \end{cases}$$

Cardinality of selection (δ)	Optimized # of bitmap scans
1	3
2	2
3	3
4	*1*
5	3
6	2
7	3
8	**0**

Table 2: Optimized Number of Bitmap Scans with respect to Cardinality of Selection Range

Algorithm 4 (Finding common variables)

```
Input:  Selection predicate $A \in V$, $V = \{v_1, \ldots, v_k\}$
        An EBI on $A$ with the mapping function $\mathbb{M} : A \rightarrow \{\langle b_{n-1} \cdots b_0 \rangle | b_i \in \{0,1\},\ 0 \leq i < n\}$
        A set of Boolean variables used in the retrieval min-terms, $\{\mathbf{b}_{n-1}, \ldots, \mathbf{b}_0\}$
Output: A set of common variables, $\mathbb{C}$, of the retrieval function for $A \in V$
Begin
    set $\mathbb{C} = \emptyset$
    initialize an array of bits, $\mathbf{B}[n]$
    set $\mathbf{B} = 1$
    for $i = 2$ to $k$
        $\mathbf{B} = \mathbf{B} \cdot (\mathbb{M}(v_{i-1}) \odot \mathbb{M}(v_i))$
        /* $\odot$ denotes exclusive-NOR, $\cdot$ denotes AND,
           and $\mathbb{M}(v_i)$ denotes the encoded value of $v_i$ */
    for $i = 0$ to $n-1$
        if $\mathbf{B}[i] = 1$ then $\mathbb{C} = \mathbb{C} \cup \{\mathbf{b}_i\}$
    return $\mathbb{C}$
End.
```

Note that there does not exist an unconditional implication from the cardinality of selection range to the number of bitmap scans in Table 2. Nevertheless, the numbers provide for a quick estimation of the saving in the index scanning phase and the cost arising in the data fetching phase. For an extreme example, if the cardinality of selection is expanded to 8 in Table 2, although the cost of the index scanning phase is reduced to 0, a table scan is required to access all the data, since all attribute values are included in the predicate.

In addition, Table 2 is used not only to check how much reduction an inclusion might lead to, but also to determine how *effective* an inclusion is. An effective inclusion is one that can lead to further reduction of the retrieval function. For example, to expand the cardinality of a selection from 4 to 6 can never be an effective inclusion.

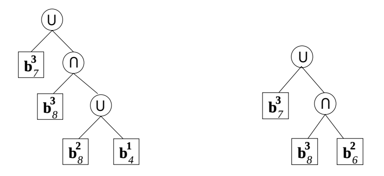
For some situations, a less optimal solution might perform even better than an optimal solution. For example, instead of finding the best inclusion, we could coarsely expand the cardinality of selection range by finding the common variables of the retrieval min-terms. For the example in the beginning of this section, the common variables of $\mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_1$ and $\mathbf{b}_3\mathbf{b}_2\bar{\mathbf{b}}_0$ are $\mathbf{b}_3\mathbf{b}_2$, *i.e.*, expanding $\mathbf{b}_3\mathbf{b}_2(\bar{\mathbf{b}}_1 + \bar{\mathbf{b}}_0)$ to $\mathbf{b}_3\mathbf{b}_2$. In this example, it is also the best expansion. Although there might exist another better way of inclusion, the cost of finding it might not be compensated by its benefit. Algorithm 4 finds the set of common variables of the retrieval Boolean function at a complexity of $O(|V|)$.

So far, we have discussed the application of the inclusion/exclusion approach on EBIs for the selection operator "$\in$". For finite domains, other operators, such as $\notin, =, \neq, <$ and $>$, can all be rewritten using $\in$. Therefore, the above discussion does not lose its generality. In addition, if the encoding function of an EBI is *total-order preserving*[9], another way of applying the inclusion/exclusion approach is discussed next.

Dynamic optimization for bit-slices For both bit-sliced indexes and EBIs with total-order preserving encoding, there exists a total ordering in the bitmaps that is the same as that in the attribute domain. With this characteristic, the principle of inclusion/exclusion can be applied in the following way.

Let us first quickly review selection evaluation using bit-sliced indexes. Following the example in section 3.2, to evaluate the predicate "$A \leq 864$", the execution tree is shown in Figure 9(a).

The idea of the principle of inclusion/exclusion is to expand the range of the selection such that tree-reduction on the execution tree occurs, *e.g.*, by enlarging the range of the selection from "$A \leq 864$" to "$A \leq 894$", the execution tree is reduced as Figure 13(a) shows.



(a) $A \leq 894$ (b) $A \leq 869$

Figure 13: Execution Trees for $A \leq 894$ and $A \leq 869$

In addition, if we change the predicate to "$A \leq 869$", the execution tree is reduced further as shown in Figure 13(b). We can see that the scale of enlargement in selection ranges does not imply the scale of reduction in bitmap scans. The latter case of the last example reduces the number of bitmap scans down to 3, while the former case reduces the number of bitmap scans to 4, in spite of larger expansion of the selection range.

Inequality (11) is also used as the cost model to determine whether an expansion in selection range improves the query performance or not. However, instead of using page-level bitmaps to compute the extra cost arising in the data fetching phase, the distribution of the underlying data is used to estimate the change in query selectivity. For the above example, assuming the attribute values of A are evenly distributed within $100 \leq A \leq 900$, changing the predicate from "$A \leq 864$" to "$A \leq 869$" increases the query selectivity by $\frac{5}{801}$,

[9]An encoding function is called total-order preserving, if there exists a total order in the domain of the indexed attribute, and the same total order still exists in the encoded attribute domain.

while changing the predicate to "$A \leq 894$" increases the query selectivity by $\frac{30}{801}$.

For numeric data types with even distribution, it is preferable to expand the selection range of the least significant digit first, since the higher the query selectivity increases, the higher the extra cost in the data fetching phase. Because of space limitation, we do not give another example for EBIs.

5 Conclusions

We discussed issues of static and dynamic query optimization for bit-sliced indexes and encoded bitmap indexes. The main contributions of this paper are:

For *static query optimization*,

- we divide selections into continuous and discrete ones. We have proposed two algorithms for evaluation of discrete selections using bit-sliced indexes. Time complexities of these two algorithms are also derived.
- We have defined a global time function of bit-sliced indexes for both types of selections, and the "break-even" point is defined as the minimum point of the function. The "break-even" point serves as a new design criterion for global time-optimal bit-sliced indexes, with respect to both types of selections.
- The effect of space constraints on finding a global time-optimal index is studied.
- In [3], an algorithm `RangeEval-Opt` was proposed to improve its former version proposed in [14]. In this paper, we have proposed further improvements on `RangeEval-Opt` using the "tree-reduction" technique.
- To optimize the processing time of EBIs, we propose to use known methods, such as *ordered binary decision diagram* (OBDD), *binary decision diagram* (BDD), or *tabular method* to minimize the retrieval Boolean function.

For *dynamic optimization*,

- The principle of inclusion and exclusion is introduced, and its application to both EBIs and bit-sliced indexes is discussed.
- Cost models, both analytical and probabilistic, have been defined to select better execution plans.
- An algorithm is defined to quickly find a set of common variables in the retrieval Boolean function. This set of common variables serves as the retrieval function for the "coarse" filtering in the approach of inclusion and exclusion.

Future work includes:

- Defining heuristics or tools that assist human experts to define well-defined encodings.
- To apply the inclusion/exclusion principle in dynamic query optimization, finding *what to include* or *how to expand* is itself an optimization problem. For EBIs, we have proposed an algorithm to find the set of common variables in the retrieval Boolean function. However, for bit-sliced indexes, guidelines for quickly finding how to expand the selection range are still required.

Acknowledgment I am grateful to Alex Buchmann for his detailed comments that enabled to improve this paper and also to Ron Bourret for his reviewing regarding the presentation of this paper.

References

[1] R.E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Trans. on Computers, 35(6), 1986.

[2] R.E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*, ACM Computing Surveys, 24(3), 1992.

[3] C.-Y. Chan, Y.E. Ioannidis, *Bitmap Index Design and Evaluation*, SIGMOD Conf., Seattle, 1998.

[4] C.-Y. Chan, Y.E. Ioannidis, *Bitmap Index Design and Evaluation*, CS Dept., Univ. of Wisconsin-Madison, *http://www.cs.wisc.edu/~cychan/paper101.ps*, 1997.

[5] J.-H. Chu, G.D. Knott, *An Analysis of B-Trees and Their Variants*, Information Systems, 14(5), 1989.

[6] D. Comer, *The Ubiquitous B-Tree*, ACM Computing Surveys, 11(2), June 1979.

[7] D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood, *Implementation Techniques for Main Memory database Systems*, SIGMOD, New York, 1984.

[8] J. Feigenbaum, S. Kannan, M.Y. Vardi, M. Viswanathan, *Complexity of Problems on Graphs Represented as OBDDs*, AT&T Technical Report: 97.1.1, 1996.

[9] G. Graefe, *Query Evaluation Techniques for Large Databases*, ACM Computing Surveys, 25(2), 1993.

[10] K. Küspert, *Storage Utilization in B^*-Trees with a Generalized Overflow Technique*, Acta Informatica, 19, 1983.

[11] E.J. McCluskey, *Minimisation of Boolean functions*, Bell System Technical Journal, 35(6), 1956.

[12] P. O'Neil, *Model 204 Architecture and Performance*, Springer-Verlag LNCS 359, 2nd Intl. Workshop on High Performance Transactions Systems, Asilomar, CA, 1987.

[13] P. O'Neil, G. Graefe, *Multi-Table Joins Through Bitmapped Join Indices*, SIGMOD Record, 24(3), 1995.

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

[15] W.V. Quine, *The Problem of Simplifying Truth Functions*, American Mathematical Monthly, 59(8), 1952.

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

[17] L.D. Shapiro, *Join Processing in Database Systems with Large Main Memories*, ACM TODS, 11(3), 1986.

[18] M.C. Wu, A. Buchmann, *Encoded Bitmap Indexing for Data Warehouses*, 14th ICDE, Orlando, 1998.

[19] M.C. Wu, *Query Optimization for Selections using Bitmaps*, Tech. Report, DVS98-2, DVS1, CS Dept, Technische Universität Darmstadt, 1998.

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