Optimizing Queries On Compressed Bitmaps

Sihem Amer-Yahia
AT&T Labs–Research
sihem@research.att.com

Theodore Johnson
AT&T Labs–Research
johnsont@research.att.com

Abstract

Bitmap indices are used by DBMS's to accelerate decision support queries. A significant advantage of bitmap indices is that complex logical selection operations can be performed very quickly, by performing bit-wise AND, OR, and NOT operators. Although they can be space inefficient for high cardinality attributes, the space use of compressed bitmaps compares well to other indexing methods. Oracle and Sybase IQ are two commercial products that make extensive use of compressed bitmap indices.

Our recent research showed that there are several fast algorithms for evaluating Boolean operators on compressed bitmaps. Depending on the nature of the operand bitmaps (their format, density and clusterdness) and the operation to be performed (AND, OR, NOT, ...), these algorithms can have different execution times. We present a linear time dynamic programming search strategy based on a cost model to optimize query expression evaluation plans. We also present rewriting heuristics that encourage better algorithms assignments. Our performance results show that the optimizer requires a negligible amount of time to execute, and that optimized complex queries can execute up to three times faster than unoptimized queries on real data.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

1 Introduction

A *bitmap index* is a bit string in which each bit is mapped to a record ID (RID) of a relation. A bit in the bitmap index is set (to 1) if the corresponding RID has property P (i.e., the RID represents a customer that lives in New York), and is reset (to 0) otherwise. In typical usage, the predicate P is true for a record if it has the value a for attribute A. One such predicate is associated to one bitmap index for each unique value of the attribute A. The predicates can be more complex, for example bitslice indices [18] and precomputed complex selection predicates [21].

One advantage of bitmap indices is that complex selection predicates can be computed very quickly, by performing bit-wise AND, OR, and NOT operations on the bitmap indices. Furthermore, the indexable selection predicates can involve many attributes. Let's consider some examples, using a customer database with schema *Customer(Name, Lives_in, Works_in, Car, Number_of_children, Has_cable, Has_cellular)*

• Suppose that we want to select all customers who live in the New York city tri-state area and who drive a car that is frequently purchased in the state in which they live. Then the selection condition is, e.g. *(Lives_in="NJ"* AND *(Car="Ford Expedition"* OR *Car="GMC Suburban"))* OR *(Lives_in="NY"* AND *(Car="Honda Accord"* OR *Car="Ford Taurus"))* OR *(Lives_in="CT"* AND *(Car="Mercedes 500SL"* OR *Car="Cadillac Seville")).*

• Suppose that we want to select all customers who work in a state different than the one in which they live, have one or more children, subscribe to a cable service, but do not have a cellular phone. Then the selection condition is *((Lives_in="AL"* AND NOT *Works_in="AL")* OR ⋯ OR *(Lives_in="WY"* AND NOT *Works_in="WY"))* AND (NOT *Number_of_children = 0)* AND *Has_cable="Y"* AND *Has_cellular = "N"*

While conventional indices in general cannot handle these types of selections easily, bitmap indices can. These properties of bitmap indices have led to considerable interest in their use in Decision Support Systems (DSS). O'Neil and Quass [18] provide an excel-

lent discussion of the architecture and use of bitmap indices. O'Neil and Graefe [16] show that bitmap indices can be used as join indices for evaluating complex DSS queries on star schemas. O'Neil and Quass [18] point out that bitmap indices not only accelerate the evaluation of complex Boolean query expressions, but can also be used to answer some aggregate queries directly. Several database management system vendors have incorporated bitmap index technology into their products [24, 19]. See [11] for a discussion of the uses of bitmap indexing in Oracle.

A problem with using uncompressed (*Verbatim*) bitmap indices is their high storage costs and potentially high expression evaluation costs when the indexed attribute has a high cardinality. One method for dealing with the problem of using bitmap indices on high-cardinality attributes is to compress the bitmaps. For example, in a secondary index B-tree each unique key value might be shared by many records in a relation. Oracle uses compressed bitmaps to represent these sets of records [19]. A considerable body of work has been devoted to the study of bitmap index compression (see [15]). The use of bitmap compression has many potential performance advantages: less disk space is required to store the indices, the indices can be read from disk into memory faster, and more indices can be cached in memory. Some Boolean operation evaluation algorithms which operate on compressed bitmaps, without having to decompress them, might be faster than same operations on the Verbatim bitmaps. However, the use of bitmap compression can introduce some problems. The data-dependent nature of bitmap compression makes it difficult to apply the existing optimal (uncompressed) bitmap design theory [25, 26, 3, 4]. If the bitmap must be decompressed before performing Boolean operations, the decompression overhead might outweigh any savings in disk space or bitmap loading time.

In a recent research paper [13], we have made a detailed study of the performance of algorithms for compressed bitmap indices. In particular, we analyzed several algorithms for performing Boolean operations between (possibly compressed) bitmaps. We found that the performance of these algorithms varied widely depending on the Boolean operation to be performed and on the properties of the operand bitmaps. No single algorithm was always the best one, and in many occasions there were orders of magnitude difference in the operation execution times.

Bitmap compression presents a number of bitmap index design issues, including 1) optimal bitmap decomposition [3, 4], 2) choosing optimal compression algorithms for storage [13] and 3) optimal evaluation of query expressions (composed of several Boolean operators) over compressed bitmaps.

In this paper, we solve the third problem, of optimizing Boolean query expression evaluation. This problem is the most pressing, because the first two problems can be partially solved with heuristics [4, 13]. Furthermore, choosing an optimal storage method requires an understanding of the Boolean predicate evaluation workload. If we know how to better evaluate a query expression involving several bitmaps, it can help making better decisions about the formats under which these bitmaps should be stored. Therefore, understanding how to optimally evaluate complex Boolean predicates is a necessary prerequisite for solving the other two problems.

Because the bitmaps are used as indices, any query expression optimizer must execute very quickly. We make the following three contributions:
(i) We present an $O(n)$ algorithm to make a globally optimal assignment of operation evaluation algorithms to a fixed query expression parse tree, where n is the number of operations in the tree.
(ii) We create an empirical cost model of Boolean operation evaluation and bitmap format conversion.
(iii) We present fast query expression parse tree rewriting heuristics that work with the global optimizer to further reduce the expression evaluation time.

We implemented a compressed bitmap index and incorporated the expression optimizer. We ran a suite of experiments to show the value of having an optimizer. For example, in the case of one experiment with real data using BBC encoded bitmaps [2, 1], optimizing the evaluation plan results in a two- to five-fold speed improvement, and the improvement increases with increasing attribute cardinality. Our optimizer achieves comparable speedups on complex expressions through the use of multiple operation evaluation algorithms and some query rewrite rules.

The paper is organized as follows. We present in Section 2 previous results about a performance study of compressed bitmaps and explain what influences the evaluation of query expressions in this context. We then present our optimization strategy and cost model (Section 3). We also describe new rewriting rules that we incorporated as heuristics in our optimizer to speed up query expression evaluation. Section 4 briefly presents the main implementation modules. Finally, we give our performance results (Section 5).

2 Bitmap Operation Performance

In a previous paper [13], we measured and analyzed the performance of compressed bitmaps used for data warehousing. In this section, we review these measurements, with a particular focus on factors that affect the performance of evaluating a query expression.

A bitmap is a representation of a set, where each bit represents an element of some common domain. If the bit is set (to one), the element is a member of the set; else the bit is reset (to zero). When used as a bitmap index, the bits represent records in a relation

and a bitmap is created for each distinct value of the indexed attributes in the relation. In an index where the bitmap can be compressed and operators can use compressed bitmaps as input, a bitmap (i.e., a set) may have several possible representations or formats. It is always possible to translate a bitmap from one representation to another without loss of information.

Boolean operations on bitmaps can be performed using several algorithms. Each of the algorithms requires the inputs to be in a certain format and produces an output in a given format. The costs of an algorithm depend on the properties of the input bitmaps and the Boolean operation (AND, NOT, ...) to be performed.

The set represented by a bitmap can therefore be stored and manipulated while it is in one of several different formats, some of which bear little resemblance to bitstrings. We might be more precise to say that we work with special set representations rather than bitmaps. However "bitmap" is the established term and we continue to use it.

We first present the different bitmap formats and the Boolean operation evaluation algorithms we are using in this paper. We finish this section by introducing the problem of evaluating a Boolean query expression on bitmaps.

Verbatim: The bitmap is represented as bit string.

Run Length Encoding (RLE): The bitmap is represented as a list of differences in bit positions of successive set bits. These differences are stored as four byte integers (by restricting the size of the bitmap, they can also be stored in two byte integers). In this paper, we use only one-sided RLE codes (i.e., we represent only runs of zeros, not runs of ones).

Gzip: A Verbatim bitmap that has been compressed using zlib [5].

ExpGol: A RLE bitmap that has been compressed using the variable bit length encoding described in [15]. We use only the one-sided ExpGol encoding.

BBC: A Verbatim bitmap that has been compressed using the variable byte length encoding described in [2, 1]. We use only the one-sided BBC encoding.

A wide variety of other bitmap representations have been proposed. For example, one could use two-sided codes, list of set bit positions instead of run length encodings, or variable byte length representations of the run lengths [17]. However, this collection of formats is representative of the best bitmap formats and is sufficient for our optimization study.

There are several algorithms for evaluating a Boolean operation between two operands, each using specific formats for their inputs and having a different cost. The algorithms we use in this paper are the following(for more details, see [13]).

Basic: The two input operands are in the Verbatim format, and the output is also in the Verbatim format. The output is computed by taking the word-size *Boolean_op* between the two inputs. The following code fragment implements the Basic algorithm to compute the OR of bitmaps rbm and lbm, and store the result in lbm (which should be arrays of the largest possible integer type):

```
BitmapLength=MaxLength(lbm, rmb);
for (i=0;i<BitmapLength;i++) lbm[i] |= rmb[i];
```

Inplace: One of the operands is in the Verbatim format, the other can be in RLE, ExpGol or BBC. The second bitmap is applied to Verbatim bitmap in an operation-specific manner. In the case of an OR operation the bits indicated by the second bitmap are set in the Verbatim bitmap. The output is a Verbatim bitmap. The following code fragment implements the OR operation, where lbm is a character array and rbm is an integer array:

```
currpos = -1;
for (i=0;i<NumBitsRhs;i++)
   currpos+=rbm[i];
   bytepos=currpos/8; bitpos=currpos%8;
   lbm[bytepos] |=1 << bitpos;
```

Merge: This algorithm takes input operands in the RLE format and produces an RLE bitmap. The output is created by merging the inputs, and producing an output bit as required by the operation being evaluated. The following code fragment implements the AND operation, where lbm and rbm are the operands, stored as integer arrays, and obm is an integer array that is large enough to store the result:

```
obit = -1; opos = 0; lbit = lbm[0]-1;
lpos = 0; rbit = rbm[0]-1; rpos = 0;
while((lpos<NumBitsLhs)&&(rpos<NumBitsRhs))
   if(lbit == rbit)
      obm[opos++]=lbit-obit; obit=lbit;
      lbit+=lbm[++lpos]; rbit+=rbm[++rpos];
   else if (lbit<rbit) lbit+=lbm[++lpos];
        else rbit+=rbm[++rpos];
```

Direct: The input and output bitmaps are in a compressed format, and the operation is specialized to execute directly on the compressed bitmaps. Antoshenkov [2, 1] presents algorithms for Direct BBC operations (which we use in this paper), and Shoshani et al. [22] present Direct operations for their hybrid bitmap encoding. Because these algorithms are very complex and depend on the format that is used, we refer the reader to [2, 1] and [22] for more details.

Note that in order to use a particular evaluation algorithm, the input bitmaps must be provided in the

required format. If the inputs are not in the necessary format, they must be converted before the algorithm can be applied.

It is possible to develop variants of these four evaluation algorithms that use as input or produce as output different formats. For example, the hybrid nature of the BBC encoding allows an especially fast version of the Inplace algorithm that uses as input a Verbatim and a BBC bitmap (which we will refer to as Inplace_BBC). We also developed an Inplace algorithm that uses a bitmap in the ExpGol format instead of the RLE format (which we refer to as Inplace_ExpGol), to save on a transformation step (from ExpGol to RLE).

Boolean Operation Evaluation

Depending on the properties of the operand bitmaps, Boolean operation evaluation algorithms can have orders of magnitude differences in performance. There are two bitmap properties that can have a significant effect on format conversion and algorithm performance : their *density* and their *clusteredness*. The density of a bitmap is defined as the fraction of bits set in the bitmap. The clusteredness, or non-uniformity, or bias, of a bitmap is a measure of the departure from independence of the value of a neighboring bit. Highly clustered bitmaps tend to have all of their set bits in a few small regions of the bitmap. Due to space constraints, all of the experiments in this paper use non-clustered bitmaps (each bit is independent). See [13] for a detailed discussion of the performance effect of bitmap properties on Boolean operation evaluation.

If the output of an operation evaluated with, for e.g., Merge is used as input to an operation evaluated with Basic, the RLE output of the Merge algorithm must be converted to a Verbatim format suitable as input to the Basic algorithm. Similarly, a bitmap stored in BBC format must be converted to RLE format for use in a Merge algorithm and so on. The cost of converting bitmaps can be a significant and determining factor in choosing which algorithms to use to evaluate each operation in a query expression. See [13] for the effect of format conversion.

3 Optimizing Query Expressions

Given the wide variety of bitmap formats and evaluation algorithms, we cannot pick any single format or evaluation algorithm which is always the best one. Instead, we can optimize the evaluation plan and make use of the best bitmap format and evaluation algorithm for every operation. This optimization cannot be performed locally for each operation because the output of one operation (usually) becomes the input of another. If the format of the output is not what the next operation expects as its input, a potentially expensive format conversion is required. Therefore, optimal algorithm assignment (physical optimization) must be global. We present a fast $O(n)$ dynamic programming physical optimization algorithm.

We observed that it is possible to rewrite query expressions to obtain a faster equivalent expression. A wide variety of expression rewritings are possible. For example, one can try to combine common subexpressions. Another option is to use algebraic transformations to reduce the number of operations [25]. One can even use semantic information to obtain more efficient expressions [26]. Because we are optimizing an index structure, a very limited optimization time is available to us. The approach that we choose is to use a small collection of simple heuristic query rewritings that are robust to incorrect assumptions about the properties of the bitmaps involved in the rewritten expressions. These rewritings are inspired from our previous performance evaluation [13] and are designed to let the physical optimizer assign a better collection of algorithms to a portion of a query expression. We present the physical optimization and the cost model first, and the rewrite rules second.

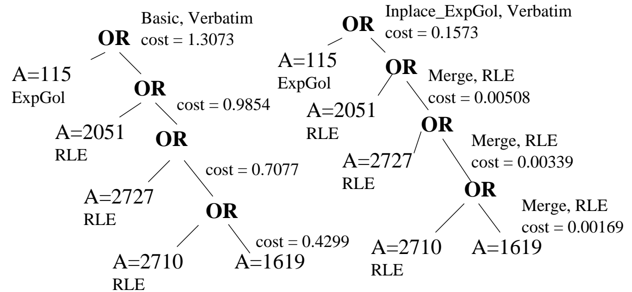

Figure 1 : A simple example

We consider a very simple example where we use a range query on a single attribute A in which each attribute value has a bitmap with density 1/3000. The query expression is the following: (A=115) OR (A=2051) OR (A=2727) OR (A=2710) OR (A=1619). This expression is as a parse tree where leaf nodes are bitmaps and interior nodes are Boolean operations. Each interior node is connected to one (case of NOT) or two subtrees (case of AND or OR). At each leaf, we know the format, density and clusteredness of the corresponding disk-resident bitmap for each value of attribute A. We can propagate this information up to the root node by computing an estimate of these properties at each interior node. For each interior node, we must assign an evaluation algorithm, and make any required format conversions.

Figure 1 shows two different algorithms assignments for the example. In one case, all the nodes are assigned the Basic algorithm and the estimated cost of the query expression is 1.3 seconds and the one. In the other case, the cost is 0.15 seconds. As can be readily seen, the optimization can have a significant effect on performance.

Optimization Algorithm

We have developed an optimization algorithm that is a variant of the well-known dynamic programming algorithm, first presented in [20]. This algorithm uses an *enumerative* search strategy but avoids enumerating all query plans by dynamically pruning suboptimal parts of the space as partial plans are generated.

Given a Boolean operation, an evaluation algorithm and the properties of the operands (format, bit density and clusteredness), we can estimate the time to perform the operation, and make any necessary format conversions (e.g., convert an RLE output to the desired Verbatim output). Therefore, the time to evaluate the whole query expression is the time to evaluate the right subtree (if any) plus the time to evaluate the left subtree (if any), plus evaluation and transformation costs at the root node. Computing the time to evaluate the subtrees is a procedure identical to that for computing the time to evaluate the whole tree. If the subtree is a leaf, the only cost is to transform the bitmap representation into the format expected by the algorithm that will be applied at its parent. Because the only interaction between the evaluation of the root and the evaluation of the subtrees is through the conversion costs, we can develop an efficient dynamic programming algorithm.

Our optimization algorithm decides, in two traversals of the parse tree, which is the best evaluation algorithm at each operation node. The algorithm starts by computing, at each operation node, the lowest cost to evaluate the expression represented by the subtree rooted at the node, for each possible output format. The optimal evaluation algorithm for each output format is also stored. By making a second pass through the tree, we make the algorithm assignments to each node.

In order to give a detailed description of the underlying dynamic programming algorithm, we need to make the following definitions:

n: A node in the query expression tree.
$p(n)$: A number in $[0, 1]$ indicating the fraction of bits set (bit density) in the bitmap output by n.
$c(n)$: A number in $[.5, 1]$ indicating the clusteredness of the bitmap output by n.
$op(n)$: Operation that node n performs. If n is a leaf, the operation is to fetch a bitmap (we set this cost to zero, as it can not be optimized). If n is an interior node, the operation is a Boolean operation.
$lc(n)$: left child of node n. $rc(n)$: right child of node n.
$\mathcal{A}$: Algorithms used to evaluate the Boolean operations. $\mathcal{A}(op)$ a subset of $\mathcal{A}$, contains the set of algorithms that can be used to evaluate op.
$\mathcal{F}$: Collection of bitmap formats.
$F_o(A)$: Output format of algorithm $A \in \mathcal{A}$.
$F_l(A)$: Input format of the left operand of algorithm $A \in \mathcal{A}$.
$F_r(A)$: Input format of the right operand of algorithm $A \in \mathcal{A}$.
$F_i(n)$: Format in which leaf node n is stored.
$conv(F_1, F_2, p, c)$: Time to convert a bitmap with bit density p and clusteredness c from format F_1 to format F_2.
$ev(op, A, p_l, c_l, p_r, c_r)$ is the time to evaluate operation op using algorithm A, where the left operand has bit density p_l and clusteredness c_l, and the right operand has bit density p_r and clusteredness c_r.
$cost(n, F)$: The optimal total time to evaluate the query expression represented by the tree rooted at n and producing output in format F.

We can derive the following equations for $cost(n, F)$ as follows:

- $cost(n, F) = conv(F_i(n), F(n), p(n), c(n))$ if n is a leaf
- $cost(n, F) = min_{A \in \mathcal{A}(op(n))}\{conv(F_o(A), F, p(n), c(n)) + ev(op(n), A, p(lc(n)), c(lc(n))) + cost(F_l(A), lc(n))\}$ if n is unary
- $cost(n, F) = min_{A \in \mathcal{A}(op(n))}\{conv(F_o(A), F, p(n), c(n)) + ev(op(n), A, p(lc(n)), c(lc(n)), p(rc(n)), c(rc(n))) + cost(F_l(A), lc(n)) + cost(F_r(A), rc(n))\}$ if n is binary .

If we precompute $cost(lc(n), F)$ and $cost(rc(n), F)$ for each $F \in \mathcal{F}$, we can evaluate $cost(n, F')$ in $O(|\mathcal{A}|)$ time. We need to evaluate this function for every node n in the expression tree T, and for every format $F \in \mathcal{F}$. Therefore, computing the minimum time to evaluate a query expression requires $O(|T| * |\mathcal{A}| * |\mathcal{F}|) = O(|T|)$ time.

Cost Model

Our optimization algorithm requires a cost model for each Boolean operation evaluation $ev(op, A, p_l, c_l, p_r, c_r)$, a cost model for the transformation costs $conv(F_1, F_2, p, c)$, and a way to estimate the properties of the bitmap produced by each node in the tree.

The execution times of the algorithms are difficult to model analytically. In addition, analytical models can be fragile. Instead, we developed an empirical performance model. We measured the time to perform each operation using each applicable algorithm for a range of the bitmap operand parameters (see [13] for a discussion of the relevant performance parameters). Given a particular function $ev(op, A, p_l, c_l, p_r, c_r)$ to evaluate, we use linear interpolation through the closest measured data points. In a similar manner, we create an empirical cost model to estimate $conv(F_1, F_2, p, c)$. While constructing these models is expensive (several hours of execution time), it only needs to be constructed once per installation site, and is robust to changes in implementation and local conditions.

The cost model needs an estimate of the density and clusteredness of each operand bitmap. The den-

sity and clusterdness of the leaf node bitmaps can be computed at compression time. The bitmaps output by the interior nodes are computed dynamically, so we need to estimate their density and clusteredness. One estimator is to assume that bitmaps are uncorrelated, except for bitmaps of values of the same attribute. Let p_l and p_r be the density of the right and left operand bitmaps, and let p_o be the density of the output bitmap. Similarly, let c_l and c_r be the clusteredness of the right and left operand bitmaps, and let c_o be the clusteredness of the output bitmap. Then:
$p_o = p_l + p_r - p_l * p_r$, OR, different attribute
$p_o = p_l + p_r$, OR, same attribute
$p_o = p_l * p_r$, AND, different attribute
$p_o = 0$, AND, same attribute
$c_o = (c_l + c_r)/2$

For a NOT operation, $c_o = c_l$ and $p_o = 1 - p_l$. The bit density and clusteredness can be computed for every node in the expression tree using a simple recursive procedure. In our case, we do it when building the tree.

Rewriting NOT-Free Expressions

Suppose that we have an expression involving only OR operations. If the operands are sparse, then Inplace is a fast evaluation algorithm. The left operand must be Verbatim, the right RLE (or BBC, or ExpGol), and the output is Verbatim. We can minimize the number of conversions into Verbatim by reorganizing the query expression to ensure that the result of an OR (that is in Verbatim) is given as the left operand of the next OR operation. To encourage the physical optimizer to use the Inplace algorithm, we need to convert the tree of OR operations from a bushy tree to a left-deep tree (to minimize the number of conversions to Verbatim), and put the densest bitmap at the leftmost leaf, since making this operand a right operand gives the least performance improvement when using Inplace. This transformation, illustrated in Figure 2, can always be applied because of the commutativity and associativity properties of the OR operation.

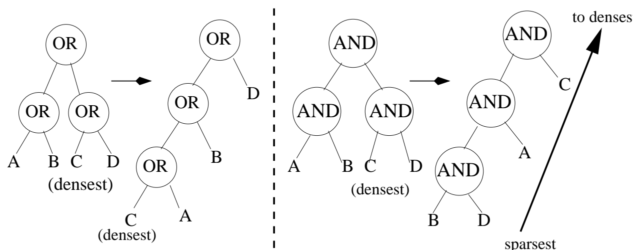

Figure 2 : Transformation of ANDs and ORs

Let us now suppose that we have a query expression involving only AND operations. We know, from our previous experiments, that the Merge and Direct algorithms are the fastest ways to perform the AND operation when the operands are sparse. Merge requires both input operands to be in a RLE format and Direct requires them to be in a BBC format. To encourage the use of these algorithms, we convert a bushy tree of AND operations into a left-deep tree, with the operands sorted from the sparsest on the bottom to the densest at the top. The sparse operands are clustered together and are likely to be evaluated with Merge or Direct. When the density reaches a certain limit, we expect the optimizer to assign a Reverse Inplace algorithm (an Inplace algorithm where the left operand has to be in RLE/BBC/ExpGol and the right one in Verbatim). The format of the result is then Verbatim and the optimizer assigns to the remaining operations an Inplace algorithm. This transformation is also illustrated in Figure 2.

A significant advantage of these transformations is that they tend to be resilient to mistakes in the assumed formats of the operands. Let's consider the case of an OR operation. The Inplace algorithm can take RLE, ExpGol, or BBC as the right operand format. If the right operand is instead supplied as Verbatim, then the reasonably efficient Basic algorithm can be used. With the AND operation, we expect that the format of sparse operands deep in the tree to be in RLE or BBC format. If one or more is instead Verbatim, then the evaluation algorithm switches to Inplace and/or Basic.

Suppose that we have a two level expression, e.g. an AND of OR clauses, or an OR of AND clauses. The OR and AND transformations presented above can still be applied within their own subtrees. These transformations can be applied iteratively because the transformations are not likely to change the format of the output. The output format of an OR subtree that has a sparse result is likely to be RLE or BBC (because the Merge and Direct algorithms are fast for sparse bitmaps), which is what our AND subtree rewriting expects. The output of a sparse AND might be in a Verbatim format, but this does not affect the validity of the OR subtree rewriting.

For three or more levels, the assumptions behind the transformations becomes questionable. However, there will be relatively few operations so far from the leaf level, so most of the expression will benefit from the rewrite.

Creating AND_NOT and OR_NOT

These rewritings aim to reduce the number of operations in a query expression by absorbing the NOT operation into an AND or an OR when it is possible.
a AND NOT *b* is equivalent to *a* AND_NOT *b*.
a OR NOT *b* is equivalent to *a* OR_NOT *b*.
AND_NOT and OR_NOT are two new operations for which we have fast algorithms to evaluate (Oracle uses the AND_NOT operation, calling it MINUS [11]). The

benefit of using these operators is not only to reduce the number of operations in the expression, but to allow the use of faster evaluation algorithms that use and accept a wider variety of bitmap formats.

The AND_NOT operator can be evaluated by Inplace, having performance similar to that of the OR operation evaluated by Inplace. The AND_NOT operator can also be evaluated by Merge and Direct, with performance between that of AND and OR evaluated by Merge and Direct. The OR_NOT operator can be evaluated by Inplace, with performance similar to that of the AND operation evaluated by Inplace. However, there is no good implementation of OR_NOT using the Merge or Direct algorithm (assuming one-sided codes).

The AND_NOT and OR_NOT transformations can be integrated with the local AND and OR subtree rewritings. After creating the (AND/OR) left-deep subtree and performing reordering, the NOT operations are absorbed into the parent operations (except for any NOT operation which is the left child of the left-most leaf). There are some considerations for operand reordering when NOT operations are absorbed – the density of a negated operand should be computed as though the NOT operation is not applied (as it will be absorbed). After rewriting the expression, the properties of the nodes in the subtree might have to be recomputed (because they depend upon the properties of their children, which might have changed).

4 Implementation

We built a prototype implementation of a compressed bitmap index that incorporates expression optimization. The index has three major components: a compressed bitmap object, a storage manager, and the optimized plan generator. The compressed bitmap object is a convenient abstraction for handling a bitmap and performing the necessary actions, such as format conversions and operation evaluation. Each bitmap object represents a bitmap of a particular length. Multiple bitmap objects can exist simultaneously.

The storage manager permits the convenient retrieval of bitmaps (i.e., in their disk-resident format), given the attribute name and the attribute value. Because we have assumed that only low to moderate cardinality attributes are indexed, the attribute-value-to-bitmap index is very simple (Oracle uses a more sophisticated index, see [19, 11]). The storage manager also records bitmap density and clusteredness statistics for use by the optimizer. We store all bitmap indices for all attributes in a single file for convenient access during the optimization and evaluation stage. The layout of the index is determined at index creation time. However we have not addressed optimal bitmap layout.

The storage manager breaks each bitmap into fixed size verbatim *bitmap blocks* before compression and disk storage. This horizontal partitioning significantly reduces operation evaluation time by increasing the likelyhood that a bitmap is in the CPU cache when it is used as an operand (e.g., an operand is usually the result of a recent operation, or was recently loaded from disk). For example, we found that performing an operation between two 8 Mbyte verbatim operands takes .91 seconds when an 8Mbyte block size is used, but .088 seconds when a 64 Kbyte block size is used.

The optimizer uses the algorithms we have described to rewrite an expression tree and make an evaluation algorithm assignment at each interior node. We walk this tree inorder traversing the left branch first to generate an evaluation plan. Finally, a simple program uses the bitmap objects and the storage manager to execute the plan.

5 Experiments

To test the beneficial effect of our optimizer, we built bitmap indices on a synthetic data set and a real data set using the Gzip, BBC, and ExpGol compression algorithms and with varying bitmap block sizes. As was suggested in [13], we compressed bitmaps that had a bit density of .05 or less, and stored the denser bitmaps Verbatim.

Data Sets: The synthetic data set has seven attributes, with attribute cardinalities 3, 10, 30, 100, 300, 1000, 3000. Each attribute value is an integer chosen independently and uniformly randomly from the attribute range. This data set is intended to aid in exploring performance trends rather than to model a real data set. Therefore, in addition, we extracted data from an actual data set, which contains information about subscribers to an AT&T service. The real data has seventeen attributes with cardinalities 3, 3, 3, 4, 7, 50, 53, 59, 209, 241, 251, 383, 792, 793, 856, 995, 1079. The attribute values are strings, and the distribution of attribute values of each attribute is highly skewed. Every attribute is indexed in both data sets.

The compressed bitmap indices are fairly space efficient. Their size is not much larger than the size of the data set when compressed. In the case of the synthetic data (16 million tuples, total size of 356 MB, compressed size of 144 MB), the size of the BBC index is 182 MB, the size of the ExpGol index is 144 MB and the size of the Gzip index is 226 MB. For the real data (6 million tuples, total size of 427 MB, compressed size of 86 MB), the size of the BBC index is 110 MB, the size of the ExpGol index is 94 MB and the size of the Gzip index is 127 MB. In the case of the real data, all 17 attributes are indexed with 7.3 bits per tuple per attribute (using the ExpGol compressed index).

We built indices with bitmap block sizes ranging from 8 Kbytes to 64 Kbytes to determine the effect of block size on performance. We found that performance improved as the block size increases (faster queries and smaller indices), but that the improvement is minor

after a block size of 32 Kbytes.

Queries: A significant advantage of bitmap indices is their ability to handle complex ad-hoc queries. However, one cannot show trends with ad-hoc queries. Instead, we used the following types of parameterizable queries:

Range: Parameterized by attribute A and range k, a range query is $(A = v_1)$ OR $\cdots$ OR $(A = v_k)$ where v_i's are randomly chosen in the range of A.

Inequality: Parameterized by attributes A, B and range k,an inequality query is $(A = v_1$ AND NOT $B = w_1)$ OR $\cdots$ OR $(A = v_k$ AND NOT $B = w_k)$ where v_i's and w_i's are randomly chosen

Although both of these query types are essentially ranges, the inequality queries are complex conditions difficult to evaluate using conventional indices and difficult to optimize using ad-hoc techniques. Inequality queries will also show the benefit of the NOT rewriting strategy.

Measured Times: In our experiments, we measure only the CPU time to evaluate the Boolean expression, not the time to fetch the compressed bitmaps from disk. This measurement strategy greatly simplified the experiments as we did not need to flush the disk cache between trials. Furthermore, the index loading time depends on the disk-resident bitmap format and the index layout, neither of which we optimize in this paper. Each data point represents the average of 11 trials. To ensure consistent measurements, for each data point we make a first evaluation and throw away the measurement, to ensure that the cache is warm for the remaining trials.

One component of the query evaluation time is the optimization time. However, even for very complex queries (i.e., inequality queries over 200 values) the optimization time was negligible (1/10 of the total time for some complex queries). In the rest of the discussion, we focus on the query evaluation time.

Range Queries: In Figure 3, we show the time to perform a range query over 20 values of an attribute of the synthetic data as the attribute cardinality is varied. For the BBC and the ExpGol compressed bitmaps, we show the two extreme evaluation plans: *Basic*, which performs no rewriting and assigns the Basic evaluation algorithm at each operator node (the default evaluation plan), and *All_opt*, which uses all of the optimization techniques that we have discussed. In the case of Gzip bitmaps, *Basic* and *All_opt* are equivalent: the optimizer always assigns the Basic algorithm. The bitmap block size is 64 Kbytes.

We note that Figure 3 is a log-log chart to better show performance trends over a wide range of attribute cardinalities. The optimizer significantly reduces expression evaluation time for the BBC encoded bitmap indices, and for the ExpGol bitmap indices when the attribute cardinality is moderately large. This difference is due to the availability or unavailability of fast evaluation algorithms. Note also that Gzip encoded bitmaps can evaluate ranges faster than ExpGol encoded bitmaps for attributes with small to moderate cardinalities, indicating that using multiple compression schemes at index creation time can improve performance.

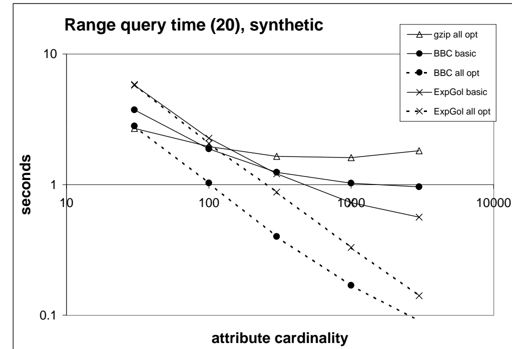

Figure 3 : Time to evaluate a range query over 20 values vs. attribute cardinality (synthetic data)

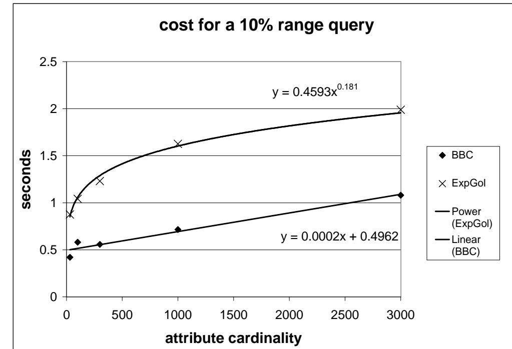

Figure 4 : Range expression evaluation time trends (synthetic data)

We find that the time to evaluate a range query expression is very well fit by a linear regression on the size of the range. By interpolating, we created Figure 4 which shows the time to evaluate a range query over 10% of the attribute values as the attribute cardinality increases. This chart shows that by optimizing the evaluation of the range query, evaluating a range over 300 values of an attribute with cardinality 3000 takes only twice as long as evaluating a range over 3 values of 30. In Figure 4 we fit trend lines to the points, a

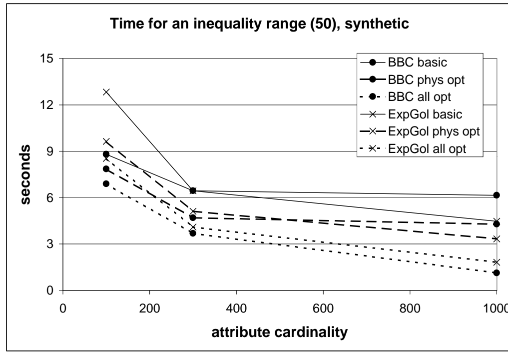

Figure 5 : Time to evaluate an inequality range over 50 values vs. attribute cardinality (synthetic data)

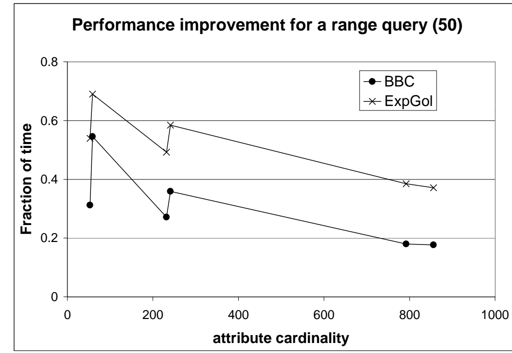

Figure 6 : Time to execute an optimized range query relative to the unoptimized query (50 values, real data)

power regression for the ExpGol encoded bitmap, and a linear regression for the BBC encoded bitmaps.

Inequality Queries: Figure 5 shows the time to evaluate an inequality query over the synthetic data. In addition to showing the time to evaluate the *Basic* plan and the *All_opt* plan, we show the time to evaluate the plan generated using the physical optimizer but with no rewriting (*Phys_only*). The performance improvement is due to absorbing the NOT operators into AND_NOT operators.

Time Ratio for Queries on Real Data: In Figure 6, we show the time to evaluate a range query using the *all opt* plan divided by the evaluation time using the *basic* plan (we change our presentation method to unclutter the charts). Note that the degree of improvement varies considerably as the attribute cardinality increases. The performance of the evaluation algorithms, and thus the optimizer, depends on the characteristics of the bitmaps. The data for these experiments is real data, and thus we cannot precisely tune the bitmap characteristics. However two trends are clear: For the BBC encoded bitmaps, optimizing the evaluation plan results in a two- to five-fold speed improvement. Furthermore, the improvement increases with increasing attribute cardinality. Figure 7 shows similar results for inequality queries over the real data.

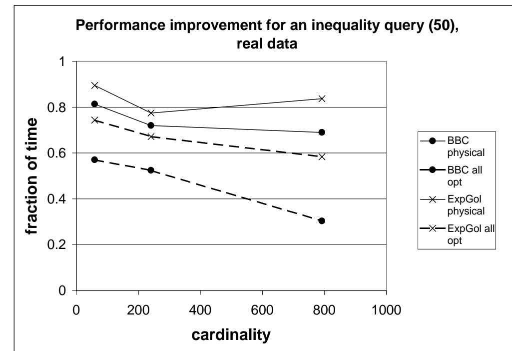

Figure 7 : Time to execute an optimized inequality query relative to the unoptimized query (50 values, real data)

On the synthetic data, the NOT-free rewriting has little effect because every attribute value has the same density. In Figure 8, we isolate the performance improvement due to NOT-free rewriting, comparing the performance obtained by the *phys opt* plan with a plan obtained by both rewriting and physical optimization. While not as dramatic as the benefit of the physical optimizer, rewriting does account for a further 4% to 10% performance improvement.

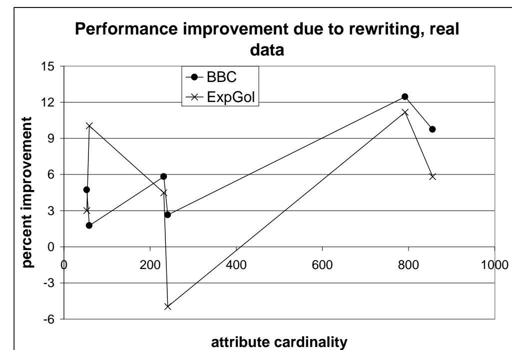

Figure 8 : Performance improvement due to NOT-free rewriting of a range query (50 values, real data)

6 Conclusion

In this paper, we have presented fast and efficient optimization techniques for the evaluation of Boolean query expressions on compressed bitmap indices. We have shown that our optimization algorithm, combined with some simple heuristics, significantly decreases the evaluation time of complex query expressions. We are currently integrating our prototype with the Daytona database system [6] to improve its performance in an OLAP environment. Our optimization algorithm is extensible and can support other Boolean operation evaluation and bitmap compression algorithms. Finally, we are working on extending our techniques to the evaluation of OLAP queries.

References

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

[2] G. Antoshenkov. Byte-aligned bitmap compression. Technical report, Oracle Corp., 1994.

[3] C-Y. Chan and Y.E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD '98.*

[4] C-Y. Chan and Y.E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. 1999 ACM SIGMOD Conf.*

[5] J-L. Gailly and M. Adler. Zlib home page. http://quest.jpl.nasa.gov/zlib/.

[6] R. Greer. Daytona and the fourth-generation language Cymbal. In *Proc. 1999 ACM SIGMOD Conf.*

[7] T. Ibaraki and T. Kameda. Optimal Nesting for Computing N-relational joins. In *ACM Transactions on Database Systems*, volume 9, September 1984.

[8] Y. E. Ioannidis and Y. Kang. Randomized algorithms for Optimizing Large Join Queries. In *Proc. ACM SIGMOD Conf.*, Atlantic city, NJ, 1990.

[9] Y.E. Ioannidis. Query Optimization. In *Encyclopedia of Computer Science*, 1997.

[10] Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM SIGMOD Conf.*, 1987.

[11] H. Jacobsson. Bitmap indexing in Oracle data warehousing. http://WWW-DB.Stanford.EDU/dbseminar/Archive/ http://FallY97/slides/oracle.

[12] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Comp. Surveys*, 16(2):111–152, June 1984.

[13] T. Johnson. Performance measurements of compressed bitmap indice. In *Proc. Conf. Very Large Data Bases*, 1999.

[14] M.V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Databse Systems. *ACM Comp. Surveys*, 20(3):192–221, September 1984.

[15] A. Moffat and J. Zobel. Parameterized compression of sparse bitmaps. In *Proc. SIGIR Conf. on Information Retrieval*, 1992.

[16] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24:8–11, 1995.

[17] P. O'Neil, 1998. Personal communication.

[18] P. O'Neil and D. Quass. Improved query performance with variant indices. In *SIGMOD '97*, 1997.

[19] Rdb7:
Performance enhancements for 32 and 64 bit systems. http://www.oracle.com/products/servers/rdb/html/fs_vlm.html.

[20] P.G. Selinger and al. Access path selection in a rdbms. In *Proc. ACM SIGMOD Conf.*, Boston, May 1979.

[21] M. Schaller. Reclustering of high energy physics data. In *Proc. Scientific and Statistical Database Management Conf.*, 1999.

[22] Shoshani and et al. Multidimensional indexing and query coordination for tertiary storage management. In *Proc. Scientific and Statistical Database Management Conf.*, 1999.

[23] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. ACM SIGMOD Conf.*, 1988.

[24] Sybase iq indexes. In *Sybase IQ Administration Guide*, Sybase IQ Release 11.2 Collection, Chapter 5., 1997. http://sybooks.sybase.com/cgi-bin/nph-dynaweb/siq11201/iq_admin/1.toc.

[25] M-C. Wu and A.P. Buchmann. Encoded bitmap indexing for data warehouses. In *Int. Conf. on Data Engineering*, 1998.

[26] M-C. Wu. Query optimization for selections using bitmaps. In *Proc. 1999 ACM SIGMOD Conf.*, 1999.