Improved Query Performance with Variant Indexes

Patrick O'Neil UMass/Boston poneil@cs.umb.edu
Dallan Quass Stanford University quass@db.stanford.edu

Abstract: The read-mostly environment of data warehousing makes it possible to use more complex indexes to speed up queries than in situations where concurrent updates are present. The current paper presents a short review of current indexing technology, including row-set representation by Bitmaps, and then introduces two approaches we call Bit-Sliced indexing and Projection indexing. A Projection index basically materializes all values of a column in RID order, and a Bit-Sliced index essentially takes an orthogonal bit-by-bit view of the same data. While some of these concepts started with the MODEL 204 product, and both Bit-Sliced and Projection indexing are now fully realized in Sybase IQ, this is the first rigorous examination of such indexing capabilities in the literature. We compare algorithms that become feasible with these variant index types against algorithms using more conventional indexes. The analysis demonstrates important performance advantages for variant indexes in some types of SQL aggregation, predicate evaluation, and grouping. The paper concludes by introducing a new method whereby multi-dimensional Group By queries, reminiscent of OLAP or Datacube queries but with more flexibility, can be very efficiently performed.

1. Introduction

Data warehouses are large, special-purpose databases that contain data integrated from a number of independent sources, supporting clients who wish to analyze the data for trends and anomalies. The process of analysis is usually performed with queries that aggregate, filter, and group the data in a variety of ways. Because the queries are often complex and the warehouse database is often very large, processing the queries quickly is a critical issue in the data warehousing environment.

Data warehouses are typically updated only periodically, in a batch fashion, and during this process the warehouse is made unavailable for querying. This allows the batch update process to *reorganize* the indexes to a new optimal clustered form, in a manner that could not take place if the indexes needed to remain available. Since the problems associated with maintaining indexes in the presence of concurrent updates is not an issue, it becomes possible to utilize more complex access structures, such as specialized indexes and materialized aggregate views (called *summary tables* in data warehousing literature), to speed up query evaluation.

This paper reviews current indexing technology, including row-set representation by Bitmaps, for speeding up the evaluation of complex queries over base data. It then introduces two previously known but relatively obscure indexing structures, which we call Bit-Sliced indexes and Projection indexes. We show that Bit-Sliced indexes and Projection indexes each provide significant performance advantages over traditional indexing structures for certain classes of queries. In fact, it may even be desirable in a data warehousing environment to have more than one type of index available on the same columns, so that the best index can be chosen for the query at hand. Indeed, the Sybase IQ data warehousing product currently provides both of these variant index types [EDEL95, FREN95], and recommends more than one index for a column in some cases. This paper represents the first time of which the authors are aware that these index structures have been examined and their differences analyzed rigorously in the database literature.

Data warehouses are often built to support *on-line analytical processing (OLAP)*. OLAP query performance depends on creating a set of summary tables to efficiently evaluate an expected set of queries. This approach of materializing needed aggregates is possible only when the expected set of queries is known in advance. Specifically, the OLAP approach addresses queries that group by different combinations of columns, known as *dimensions*. Such queries are called *Datacube* queries in [GBLP96]. But when ad-hoc queries must be issued that filter the rows by selection criteria that are not part of the dimensional scheme, summary tables that do not foresee such filtering cannot be used. In these cases the queries must be evaluated by accessing other indexes on the base data.

Example 1.1. Assume that we are given a star-join schema, consisting of a central fact table Sales, containing sales data, and dimension tables known as Stores (where the sales are made), Time (when the sales are made), Product (involved in the sales), and Promotion (method of promotion being used). (See [KIMB96], Chapter 2, for a detailed explanation of this schema. A comparable Star schema is pictured in Figure 5.1.) Using precalculated summary tables, OLAP products are able to quickly answer questions such as the total dollar sales that were made for a brand of products in a store on the East coast during the past 4 weeks with a sales promotion based on price reduction. The dimensions by which the aggregates are "sliced and diced" result in a multi-dimensional crosstabs calculation (Datacube) in which some or all of the cells may be precalculated and stored as summary tables for efficiency. But if we wanted to perform some selection criterion that has not been precalculated, such as asking for the same quantity as the one just named except for sales that occurred on days where the temperature reached 90, the answer cannot be supplied quickly if summary tables with dimensions based upon temperature do not exist. And there is a limit to the number of dimensions that can be represented in precalculated summary tables, since all combinations of such dimensions must be precalculated in order to achieve good performance at runtime. This suggests that queries requiring rich selection criteria must be evaluated by accessing the base data, rather than precalculated summary tables. u

An algorithm for efficient evaluation of OLAP-style queries with rich selection criteria is presented in Section 5.

Paper outline: We define Value-List, Projection, and Bit-Sliced indexes and their use in query processing in Section 2. Section 3 presents algorithms for evaluating aggregate functions using the index types presented in Section 2. Algorithms for evaluating Where Clause conditions, specifically range predicates, are presented in Section 4. In Section 5 of this paper, we introduce a method whereby OLAP-style queries that permit non-dimensional selection criteria can be efficiently performed. The method combines Bitmap indexing and physical row clustering, two features which provide important advantage for OLAP-style queries. Our conclusions are given in Section 6.

2. Indexing Definitions

In this section we examine traditional Value-List indexes and show how a Bitmap representation for storing RID-lists can easily be incorporated. We then explain Projection and Bit-Sliced indexes.

2.1 Traditional Value-List Indexes

Database indexes provided today by most database systems use B^+-tree[1] structures to retrieve rows of a table with specified values involving one or more indexed columns (see [COMER]). The leaf level of the B-tree index consists of a sequence of entries for index keyvalues. Each keyvalue reflects the value of the indexed column or columns in one or more rows in the table, and each keyvalue entry references the set of rows with that value. Since in a relational database all rows of the indexed table are referenced exactly once in the B-tree, partitioned by keyvalue, this type of index can also be referred to as a Value-Partitioned index. However, object-relational databases allow multi-valued attributes, so that in the future the same row may appear under many keyvalues in the index. We therefore refer to this type of index simply as a Value-List index.

---

[1] B^+-trees are commonly referred to simply as B-trees in database documentation, and we will follow this convention.

In the case where a keyvalue represents multiple column values, the column values are concatenated in such a way that values of the individual columns involved can be retrieved from the concatenated form. Thus in the concatenation of columns: LNAME||FNAME, the pair LNAME = 'ABC' and FNAME = 'DEFG' can be distinguished from the pair LNAME = 'ABCD' and FNAME = 'EFG'.

Traditionally, Value-List (B-tree) indexes have referenced each row individually as a RID, a *Row ID*, specifying the disk position of the row. A sequence of RIDs, known as a RID-list, is held in each distinct keyvalue entry in the B-tree. In indexes with a relatively small number of keyvalues compared to the number of rows, most keyvalues will have a large number of associated RIDs and the potential for compression arises by listing a keyvalue once, at the head of what we call a *RID-list Fragment*, containing a long list of RIDs for rows with this keyvalue. For example, MVS DB2 provides this kind of compression, listing one keyvalue for a Fragment of up to 255 4-byte RIDs in sequence (see [O'NEI96], Figure 7.19). Keyvalues with more than 255 associated rows require multiple Fragments of this kind, and some RID-lists are too long to materialize in memory all at once. We assume in what follows that RID-lists (and Bitmaps, which follow) are read from disk in multiples of Fragments. With this amortization of the space for the keyvalue over multiple 4-byte RIDs, the length in bytes of the leaf level of the B-tree index can be approximated as 4 times the number of rows in the table, times the reciprocal of the average fullness of the leaf nodes. In what follows, we will assume that we are dealing with data that is only updated infrequently, so that B-tree leaf pages can be completely filled and will be reorganized during batch updates. Thus the length in bytes of the leaf level of the B-tree index with a small number of keyvalues can be approximated as 4 times the number of rows in the table.

2.1.1 Bitmap Indexes

Bitmap indexes were first developed for database use in the Model 204 product from Computer Corporation of America (this product has been in use since the 1960s, see [O'NEI87]). A Bitmap is an alternate method of representing RID-lists in a Value-List index. Bitmaps are more space-efficient than RID-lists when the number of keyvalues for the index is low. Furthermore, we will show that for many functions Bitmaps are more CPU-efficient as well, because of the simplicity of their representation. To create Bitmaps for the n rows of a table T = $\{r_1, r_2, \ldots r_n\}$, we start with a 1-1 mapping m from rows of T to Z[M], the first M positive integers. In what follows we avoid frequent reference to the mapping m from rows of T to X[M]. When we speak of the *ordinal number* of a row r of T, we will mean the value m(r).

Note that while there are n rows in T = $\{r_1, r_2, \ldots r_n\}$, it is not necessarily the case that the maximum ordinal number M is the same as n, since a method is commonly used to associate a fixed number of rows p with each disk page for fast lookup. Thus for a given row r with ordinal number j, the table page number that should be accessed to retrieve row r is j/p and the page slot is (in C terms) j%p. This also means that rows will be assigned ordinal numbers in clustered sequence, a valuable property. Since the rows might have variable size and we may not always be able to accommodate an equal number of rows on each disk page, the value p must be a chosen as a maximum. Thus some integers in Z[M] might be wasted, in the sense that they will not be in the range of the mapping m for any row r in T; instead, they will correspond to non-existent slots on pages that cannot accommodate the full set of p rows. (And we may find that $m^{-1}(j)$ for some ordinal numbers j in Z[M] is undefined.)

A "Bitmap" B is defined on T as a sequence of M bits. If the Bitmap B is meant to list rows in T with a given property P, then for each row r with ordinal number j that has the property P, we set bit j in B; all other bits are set to zero. A Bitmap index for a column C with values $v_1, v_2, \ldots, v_k$, is a B-tree with entries having these keyvalues and associated data portions that contain Bitmaps for the properties C = v_1, C = v_2, . . ., C = v_k. Thus Bitmaps in this index are merely a different way of specifying lists of RIDs with specific column values. See Figure 2.1 for an Example. Note that a series of successive Bitmap Fragments make up the entry for "department = 'sports'".

We say that Bitmaps are *dense* if the number of one-bits as a proportion of all bits in the Bitmap is large. For example, a Bitmap index for a column with 32 values will have Bitmaps with average density of 3.125%, which means that about one bit in 32 is set to one. In this case the disk space to contain the Bitmaps of a Bitmap column index will be comparable to the disk space needed for a RID-list index (which requires about 32 bits for each RID present). While the uncompressed Bitmap index size is proportional to the number of column values, a RID-list index is about the same size for any number of values (as long as the number of values is much less than the number of rows, so we can continue to amortize the keysize with a long block of RIDs). For columns with a very small number of values, the Bitmaps of a Bitmap index will have high densities (such as 50% for predicates such as GENDER = 'M' or GENDER = 'F"), and the disk savings is enormous. On the other hand, when average Bitmap density for a Bitmap index becomes too low, methods exist for compressing a Bitmap. The simplest of these is to translate the Bitmap back to a RID list, and we shall assume this in what follows.

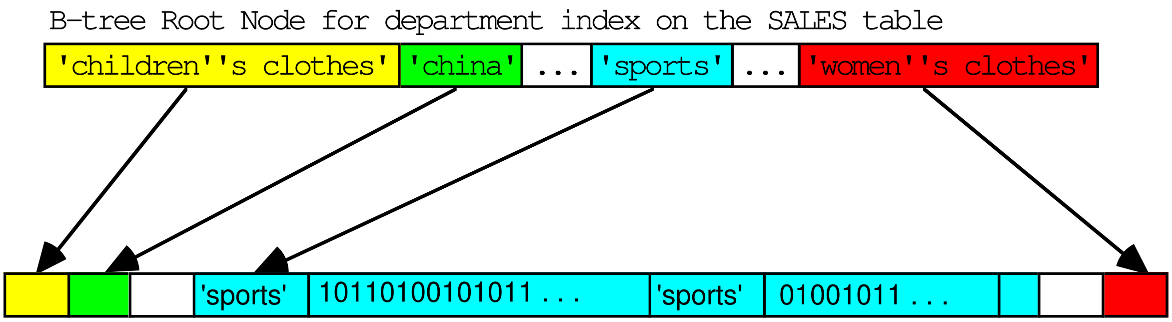

Figure 2.1. Example of a Bitmap Index on `department`, a column of the `SALES` table

2.1.2 Bitmap Index Performance

An important consideration for database query performance is the fact that Boolean operations, such as AND, OR, and NOT are extremely fast for Bitmaps. For given Bitmaps B1 and B2 we can calculate a new Bitmap B3, where B3 = B1 AND B2, by treating all bitmaps as arrays of long ints and looping through them, using the & operation of C:

```
for (i = 0; i < len(B1); i++)      /* Note: len(B1) = len(B2) = len(B3)     */
    B3[i] = B1[i] & B2[i];         /* B3 = B1 AND B2                        */
```

We would not normally expect the entire Bitmap to be memory resident, but would perform a loop to operate on Bitmaps by reading them in from disk in long Fragments. We ignore this loop in what follows. Using a similar approach, we can calculate B3 = B1 OR B2. But calculating B3 = NOT(B1) requires an extra step. Since some bit positions can correspond to non-existent rows, we postulate an *Existence Bitmap* (designated *EBM*) which has exactly those 1 bits corresponding to existing rows. Now when we perform a NOT on a Bitmap B, we loop through a long int array performing the ~ operation of C, and AND the result with the corresponding long int from EBM.

```
for (i = 0; i < len(B1); i++)      /* Note: len(B1) = len(B2) = len(B3)     */
    B3[i] = ~B1[i] & EBM[i];       /* B3 = NOT(B1)                          */
```

Typical Select statements may have a number of predicates in their Where Clause that must be combined in a Boolean manner. The resulting set of rows, which is retrieved or aggregated in the Select target-list, is called a *Foundset* in what follows. Sometimes, the rows filtered by the Where Clause must be further grouped, due to a Group By clause, and we refer to the set of rows restricted to a single group as a *Groupset*.

Finally, we show how the COUNT function for a Bitmap of a Foundset can be efficiently performed. First we create a short int array `shcount[ ]`, with entries initialized to contain *the number of bits in the entry subscript*.

```
  #define SHNUM 65536         /* the number of distinct short ints           */
  short int shcount[SHNUM];   /* one for each short int subscript            */

  /* Assume now that the function with prototype:  int  fcount(short int)
    returns a count of the number of 1-bits in its short int argument        */

  for (i = 0; i < SHNUM; i++)
    shcount[i] = fcount(i);   /* shcount[i] is count of bits in subscript    */
```

With this array created, we can loop through a Bitmap as an array of short int values, to get the count of the total Bitmap as shown in Algorithm 2.1. Clearly the `shcount[ ]` array is used to provide efficiency of multi-bit parallelism in calculating the COUNT. An array with a long int subscript would take too much memory. The `shcount[ ]` array could be declared of type char, since the count of bits in a short int subscript will fit in the significance of a char; this might be preferable, notwithstanding the extra looping, if the char `shcount[ ]` array can be cached.

Algorithm 2.1. Performing COUNT with a Bitmap

```
/* Assume B1[ ] is a short int array overlaying a Foundset Bitmap            */
  count = 0;
  for (i = 0; i < shnum; i++)
    count += shcount[B1[i]];  /* add count of bits for next short int        */
u
```

These loops to calculate Bitmap AND, OR, NOT, or COUNT are extremely fast compared to the same operations on RID lists where several operations are required for each RID, so long as the Bitmaps involved have reasonably high density (perhaps down to 1% or less).

Example 2.1. In the Set Query benchmark of [O'NEI91], the measurement results from one of the SQL statements in Query Suite Q5 give a good illustration of Bitmap performance. For a table named BENCH of 1,000,000 rows, two columns named K10 and K25 have cardinalities 10 and 25, respectively, with all rows in the table equally likely to take on any valid value for either column. Thus the Bitmap densities for indexes on this column are 10% and 4% respectively. One SQL statement from the Q5 Suite is:

```
[2.1]   SELECT K10, K25, COUNT(*)
           FROM BENCH
           GROUP BY K10, K25;
```

A 1995 benchmark on a 66 MHz Power PC of the Praxis Omni Warehouse, a C language version of MODEL 204, demonstrated an elapsed time of only 19.25 seconds to perform this query. The method employed was to read in the Bitmaps from the indexes for all values of K10 and K25, perform a double loop through all 250 pairs of values, AND all pairs of Bitmaps, and COUNT the results. Thus 250 ANDs and 250 COUNTs of 1,000,000 bit Bitmaps are performed in only 19.25 seconds on a relatively weak processor. By comparison, MVS DB2 Version 2.3, running on an IBM 9221 model 170 mainframe used an algorithm that extracted and wrote out all pairs of (K10, K25) values from the rows, sorted by value pair, and counted the result in groups, taking 248 seconds of elapsed time and 223 seconds of CPU. (See [O'NEI96] for more details.)u

2.1.3 Segmentation

To optimize Bitmap index access, Bitmaps are usually broken into Fragments of equal sizes which fit on single fixed-size disk pages. Corresponding to these Fragments, the rows of a table are partitioned into *Segments*, with an equal number of ordinal numbered rows for each segment, corresponding to a Bitmap Fragment. For example, in MODEL 204 (see [M204], [O'NEI87]), a Bitmap Fragment is expected to fit on a 6 KByte page, and contains about 48K bits. The table is therefore broken into segments of about 48K rows each. This segmentation has two important implications.

The first implication involves RID-lists. When Bitmaps are sufficiently sparse that they need to be converted to RID-lists, the RID-list for a segment is guaranteed to fit on a disk page (1/32 of 48K is about 1.5K; MODEL 240 actually allows sparser Bitmaps than that, so several RID lists for different segments or even different index values might fit on a disk page). Furthermore, RIDs need only be two bytes in length, because they only specify the row position within the segment (the 48K rows of a segment can be counted in a short int). At the beginning of each RID-list, the segment number will specify the higher order bits of a longer RID (4 byte or more), and within the segment, the segment relative RIDs only require two bytes each. This is an important form of prefix RID compression, which greatly speeds up index range search.

The second implication involves saving I/Os in combining predicates. The B-tree index entry for a particular value in MODEL 204 is made up of a number of pointers by segment to Bitmap or RID-list Fragments, but only when the segments have representative rows. In the case of a clustered index, for example, each particular index value entry will have pointers to only a small set of segments. Now if several predicates involving different column indexes are ANDed, we can think of the evaluation as taking place segment-by-segment. If one of the predicate indexes has no pointers to Bitmap or RID-list Fragments, then the index Fragments for the other predicates can be ignored as well. Queries such as this can turn out to be very common in a workload, and the I/O work saved by being able to ignore a lot of index Fragments can have significant effect on performance.

In some sense, Bitmap representations and RID-list representations are interchangeable: both provide a way to list all rows with a given index value or range of values. It is simply the case that Bitmaps are much more efficient than RID-lists, both in storage use and efficiency of Boolean operations, when the Bitmap representations involved are relatively dense. Indeed a Bitmap index can contain RID-lists for some entry values or even for some Segments within a value entry, whenever the number of rows with a given keyvalue would be too sparse in some segment for a Bitmap to be efficiently used. In what follows, we will assume that a Bitmapped index combines Bitmap and RID-list representations where appropriate, and continue to refer to the hybrid form as a *Value-List Index*. When we refer to the *Bitmap* for a given value v in the index, this is understood to be a generic name: it may be a Bitmap or it may be a RID-list. We intend to compare the use of this type of index to two new non-traditional forms of index, which we will call *Projection indexes* and *Bit-Sliced indexes*.

2.2 Projection Indexes

A Projection index is a simple structure. Assume that C is a column of a table T; then the Projection index on C consists of a stored sequence of column values from C, in order by the ordinal row number in T from which the values are extracted. If the column C is fixed length, 4 bytes in length, then we can fit 1000 values from C on each 4 KByte disk page, and continue to do this for successive column values until we have constructed the Projection index. Now for a given ordinal row number $n = m(r)$ in the table, we can access the proper disk page, p, and slot, s, to retrieve the appropriate C value with a simple calculation: $p = n/1000$ and $s = n\%1000$. Furthermore, given a C value in a given position of the Projection index, we can calculate the ordinal row number easily: $n = 1000*p + s$.

If the column values for C are variable length instead of fixed length, we have two alternatives. We can either fix on a maximum size and place a fixed number of column value on each page, as before, or we can use a B-tree structure to access the column value C by a lookup of the ordinal row number n. The case of variable-length values is obviously somewhat less efficient than fixed-length, and we shall assume fixed-length C values in what follows.

The Projection index turns out to be quite useful in cases where column values must be retrieved for all rows of a Foundset, where the Foundset is dense enough such that several column values would probably be found on the same disk page of the Projection index, but the rows themselves, being larger, would appear on several different pages. This would happen, for example, if the density of the Foundset is 1/50 (no clustering, so the density is uniform across all table segments), and the column values are 4 bytes in length, as above. Then 1000 values will fit on a 4 KByte page, and we expect to pick up 20 values per Projection index page. In contrast, if the rows of the table were retrieved, then assuming 200-byte rows only 20 rows will fit on a 4 KByte page, and we expect to pick up only 1 row per page. Thus reading the values from a Projection index requires only 1/20 the number of page access as reading the values from the rows. The Sybase IQ product is the first one to have utilized the Projection index heavily, under the name of "Fast Projection Index" [EDEL95, FREN95].

Note that the definition of the Projection index is reminiscent of the approach of vertically partitioning the columns of a table. Vertical partitioning is a good strategy for workloads where small numbers of columns are retrieved by most Select statements. But it is a bad idea, for example, in situations where most of the columns of each row are retrieved and densities of retrieval are relatively small, so that even having 1000 column values on a disk page is insufficient to achieve repeated access to column-value disk pages. Vertical partitioning is actually forbidden by the TPC-D benchmark, presumably on the theory that the queries chosen have not been sufficiently tuned to penalize this strategy. But Projection indexes are not the same as vertical partitioning. We are assuming that rows of the table are still stored in contiguous form (the TPC-D requirement) and the Projection indexes are auxiliary aids to retrieval efficiency. Of course this means that column values will be stored more than once, but in fact all traditional indexes materialize the values of columns in this same sense.

2.3. Bit-Sliced Indexes

A Bit-Sliced index stores a set of "Bitmap slices" which are "orthogonal" to the data held in a Projection index. As we will see, they provide an efficient means of calculating aggregates of Foundsets. We begin our definition of Bit-Sliced indexes with an example.

Example 2.2. Consider a table named SALES which contains rows for all sales that have been made during the past month by individual stores belonging to some large chain. The SALES table has a column named dollar_sales, which represents for each row the dollar amount received for the sale.

Now interpret the dollar_sales column as an integer number of pennies represented as a binary number with N+1 bits. For row with ordinal number n in SALES with a non-null value in the dollar_sales column, we define a function D(n, i), i = 0, . . . , N, as follows:

D(n, 0) = 1 if the ones bit for dollar_sales in row number n is on, else D(n, 0) = 0
D(n, 1) = 1 if the twos bit for dollar_sales in row number n is on, else D(n, 1) = 0
. . .
D(n, i) = 1 if the 2^i bit for dollar_sales in row number n is on, else D(n, i) = 0

For a row numbered n with a null value in the dollar_sales column, we define D(n, i) = 0, for all i. Now for each value i, i = 0 to N, such that D(n, i) > 0 for *some* row in SALES, we define a Bitmap B_i on the SALES table so that bit n of Bitmap B_i is set to D(n, i). Note that by requiring that D(n, i) > 0 for some row in SALES, we have guaranteed that we do not have to represent any

Bitmaps of all zeros. For a real table such as SALES, the appropriate set of Bitmaps with non-zero bits can easily be determined at Create Index time. u

The definitions of Example 2.1 generalize to any column C (or conceivably even a concatenation of columns, though we will not consider this) in a table T, where the column C is interpreted as a sequence of bits, from least significant (i = 0) to most significant (i = N).

Definition 2.1: Bit-Sliced Index. The Bit-Sliced index on the C column of table T is the set of all Bitmaps B_i as defined in Example 2.2. It should be clear, since a null value in the C column will not have any bits set to 1, that only rows with non-null values appear as 1-bits in any of these Bitmaps. Each individual Bitmap B_i is called a *Bit-Slice* of the column. We also define the Bit-Sliced index to have a Bitmap B_{nn} representing the set of rows with non-null values in column C, and a Bitmap B_n representing the set of rows with null values. Clearly B_n can be derived from B_{nn} and the Existence Bitmap EBM by the formula B_n = EBM AND NOT(B_{nn}), but we wish to save this effort in algorithms that follow. In fact, the Bitmaps B_{nn} and B_n turn out to be so useful that we assume from now on that B_{nn} exists for Value-List Bitmap indexes (clearly B_n already exists, since null is a particular value for such an index).u

In the algorithms that follow, we will normally be assuming that the column C is numeric, either an integer or a floating point value. In using Bitmap indexes, it is always necessary that different values have matching decimal points in their Bitmap representation. Depending on the variation in size of the floating point numbers, this could lead to an exceptionally large number of slices when values differ by numerous orders of magnitude. Such an eventuality is considered unlikely in business applications, however.

A user-defined method to bit-slice aggregate quantities was used by some MODEL 204 users and is defined on page 48 of [O'NEI87]. Sybase IQ currently provides a fully realized Bit-Sliced index, which is known to the query optimizer and transparent to SQL users. Usually, a Bit-Sliced index for a quantity of the kind in Example 2.2 will involve a relatively small number of Bitmaps (less than the maximum significance), although there is no real limit imposed by the definition. Note that 20 Bitmaps, 0 . . .19, for the dollar_sales column will suffice to represent quantities up to 2^{20} - 1 pennies, or $10,485.75, an extremely large sale by most standards. If we consider normal sales ranging up to $100.00, it is very likely that nearly all values on the range $1.00 to $100.00 will occur for some row in a large SALES table. Thus, a Value-List index would have nearly 10,000 different values, and row-sets with the given values in a Value-List index would almost certainly be represented by RID-lists rather than Bitmaps. The efficiency of performing Boolean Bitmap operations would be lost with a Value-List index, but not with a Bit-Sliced index, where all values are represented with about 20 Bitmaps.

There is an important fact about basic equivalence of the different index types.

Theorem 2.1. For a given column C on a table T, the information in a Bit-sliced index, Value-List index, or Projection index can each be derived from either of the others.

Proof. With all three types of indexes, we are able to determine the values of columns C for all rows in T, and this information is sufficient to create any other index. u

Although the three index types contain the same information, they provide significantly different performance advantages for different operations. In the next few sections of the paper we demonstrate this.

3. Comparison of Index Types For Aggregate Function Evaluation

In this section we give algorithms showing how Value-List indexes, Projection indexes, and Bit-Sliced indexes can be used speed up the evaluation of aggregate functions in SQL queries. We begin with a detailed analysis of evaluating SUM over a single column. Evaluating other aggregate functions is considered later, followed by a treatment of aggregate functions over multiple columns.

3.1 Evaluating Single-Column Sum Aggregates

We begin with an example to illustrate how a SUM aggregate function can be evaluated using any of four approaches: by reading the rows directly, by using a Projection index, by using a Value-List index, or by using a Bit-Sliced index.

Example 3.1. Assume that the SALES table of Example 2.2 has 100 million rows which are 200 bytes in length, stored 20 to a 4 KByte disk page, and that the following Select statement has been submitted:

```
[3.1]   SELECT SUM(dollar_sales) FROM SALES
            WHERE condition;
```

The condition in the Where clause that restricts rows of the SALES table will result in a Foundset of rows. We will assume in what follows that the Foundset determined in Select statement [3.1] contains 2 million rows and that the rows are not clustered in a small range of disk pages, but are spread out evenly across the entire table. We vary these assumptions later. We will also assume that the Foundset has already been determined, and it is represented by a Bitmap B_f. The most likely eventuality is that determining the Foundset was easily accomplished using a few indexes, so the resources used were relatively insignificant compared to the aggregate evaluation to follow.

Query Plan 1: Direct access to the rows to calculate the SUM. Each disk page contains only 20 rows, meaning that there must be a total of 5,000,000 disk pages occupied by the SALES table. Since 2,000,000 rows in the Foundset B_f represent only 1/50 of all rows in the SALES table, the number of disk pages that the Foundset occupies can be estimated (see [O'NEI96], Formula [7.6.4]) as:

$$5{,}000{,}000(1 - e^{-2{,}000{,}000/5{,}000{,}000}) = 1{,}648{,}400 \text{ disk pages}$$

The time to perform such a sequence of I/Os, assuming one disk arm retrieves 100 disk pages per second in relatively close sequence on disk, is 16,484 seconds, or more than 4 hours of disk arm use. We estimate 25 instructions needed to retrieve the proper row and column value from each buffer resident page (and this occurs 2,000,000 times), but in fact the CPU utilization associated with reading the proper page into buffer is much more significant. Each disk page I/O is generally assumed to require several thousand instructions to perform (see, for example, [PH96], the second example in Section 6.7, where 10,000 instructions are assumed).

Query Plan 2: Calculating the SUM through a Projection index. We can use the Projection index to calculate the sum by accessing each dollar_sales value in the index corresponding to an ordinal row number in the Foundset; these ordinal row numbers will be provided in increasing order. We assume as in Example 2.2 that the dollar_sales Projection index will contain 1000 values per 4 KByte disk page. Thus the Projection index will require 100,000 disk pages, and we can expect all of these pages to be accessed in sequence when the values for the 2,000,000 row Foundset are retrieved. This implies we will have 100,000 disk page I/Os, with elapsed time 1000 seconds (roughly 17 minutes), given the same I/O assumptions as in Query Plan 1. In addition to the I/O, we will need to have perhaps 10 instructions to convert the Bitmap ordinal number into a disk page offset, access the appropriate value, and add this to the SUM.

Query Plan 3: Calculating the SUM through a Value-List index. Assuming we have a Value-List index on dollar_sales, we can calculate SUM(dollar_sales) for our Foundset by ranging through all possible values in the index and determining the rows with each value, then determining how many rows with each value are in the Foundset, and finally multiplying that count by the value and adding to the SUM. In pseudo code, we have Algorithm 3.1 below. (Note that the first condition of Algorithm 3.1 tests whether all C values for rows in the Foundset are null, in which case the algorithm should return null, rather than 0.)

Algorithm 3.1. Evaluating SUM(C) with a Value-List Index

```
   If (COUNT(Bf AND Bnn) == 0)
      Return null;
   SUM = 0.00;
   For each non-null value v in the index for C      {
      Designate the set of rows with the value v as Bv
      SUM += v * COUNT(Bf AND Bv);
   }
   Return SUM;
u
```

Our earlier analysis indicates that we have perhaps 10,000 distinct values in this index, and thus the Value-List index method requires approximately 10,000 Bitmap ANDs and 10,000 COUNTs. If we make the assumption that the Bitmap B_f is held in memory all at once (100,000,000 bits, or 12,500,000 bytes) while we loop through the values, and that the sets B_v for each value v are actually RID-lists, this will entail 3125 I/Os to read in B_f, 100,000 I/Os to read in the index RID-lists for all values (100,000,000 RIDs of 4 bytes each, assuming all pages are completely full), and a loop of several instructions to translate 100,000,000 RIDs to ordinal bit positions and test if they are on in B_f.

Note that in this algorithm, we receive an enormous advantage from assuming B_f is a Bitmap (rather than a RID-list) and that it can be held in memory, so that RIDs read from the index can be looked up directly. If B_f were held as a RID-list instead, the lookup would be a good deal less efficient, and would probably entail a sort by RID value of structs containing RIDs and values from the index, followed with a merge-intersect with the RID-list B_f. Even with the assumption that B_f is a Bitmap, the loop through 100,000,000 RIDs can be extremely CPU intensive, especially if the translation from RID to bit ordinal entails a complex lookup in a memory-resident tree to determine the extent containing the disk page of the RID and the corresponding RID number within the extent. With optimal assumptions, Plan 2 seems to require 103,125 I/Os and a loop of length 100,000,000, with a loop body of perhaps 10 instructions. Even with all the disadvantages, this Query Plan is probably superior to Query Plan 1, which requires I/O for 1,340,640 disk pages.

Query Plan 4: calculating the SUM through a Bit-Sliced index. Assuming we have a Bit-Sliced index on dollar_sales as defined in Example 2.2, we can calculate SUM(dollar_sales) with the pseudo code of Algorithm 3.2.

Algorithm 3.2. Evaluating SUM(C) with a Bit-Sliced Index

```
/* We are given a Bit-Sliced index for C, containing bitmaps Bi, i = 0 to N (N = 19),
   Bn and Bnn, as in Example 2.2 and Definition 2.1.                           * /
   If (COUNT(Bf AND Bnn) == 0)
      Return null;
   SUM = 0.00
   For i = 0 to N
      SUM += 2^i * COUNT(Bi AND Bf);
   Return SUM;
u
```

Thus, we are able to perform the calculation by performing 21 ANDs and 21 COUNTs of 100,000,000 bit Bitmaps. Each Bitmap is 12.5 MBytes in length, requiring 3125 I/Os, but we assume that B_f can remain in memory after the first time it is read. Therefore, we need to read a total of 22 Bitmaps from disk, using 22*3125 = 68,750 I/Os, a bit over half the number needed in Query Plan 2. For CPU, we need to AND 21 pairs of Bitmaps, which can be done by looping through the Bitmaps in long int chunks, a total number of loop passes on a 32-bit machine equal to: 21*(100,000,000/32) = 65,625,000. Then we need to perform 21 COUNTs, which are done in half-word single instruction loops will require 131,250,000 passes. However, all these 196,875,000 passes to perform ANDs and COUNTs are single instruction loops, and thus presumably take a good deal less time than the 100,000,000 multi-instruction loops of Plan 2.u

3.1.1 Comparing Algorithm Performance

Table 3.1 compares the above four Query Plans to calculate SUM, in terms of I/O and factors contributing to CPU.

Method	I/O	CPU contributions
Add from Rows	1,341K	I/O + 2M*(25 ins)
Projection index	100K	I/O + 2M *(10 ins)
Value-List index	103K	I/O + 100M *(10 ins)
Bit-Sliced index	69K	I/O + 197M *(1 ins)

Table 3.1. Costs of four plans, I/O and factors contributing to CPU

We can compare the four query plans in terms of total dollar cost by converting I/O and CPU costs to dollar amounts, as in [GP87]. In 1996, a 2 GB hard disk with a 10 ms access time costs roughly $600. With the I/O rate we have been assuming, this is approximately $6.00 per I/O per second. A 200 MHz Pentium computer, which processes approximately 150 MIPS (million instructions per second), costs roughly $1800, or approximately $12.00 per MIPS. If we assume that each of the plans above is submitted at a rate of once each 1,000 seconds, the most expensive plan, "Add from rows", will keep 13.41 disks busy at a cost of $8046 purchase. We calculate the number of CPU instructions needed for I/O for the various plans, with the varying assumptions in Table 3.2 of how many instructions are needed to perform an I/O. Adding the CPU cost for algorithmic loops, we determine the dollar purchase cost ($Cost) for a CPU to support the method. For example, for the "Add from Rows" plan, assuming one submission each 1000 seconds, if an I/O uses (2K, 5K, 10K) instructions, the CPU cost is ($32.78, $81.06, $161.52). The cost for disk access ($8046) clearly swamps the cost of CPU in this case, and in fact the relative importance of I/O holds for all methods. Figure 3.3 shows that the Bit-sliced index is the most efficient for this problem, with the Projection index and Value-List index a close second and third. The Projection index is so much better than the fourth ranked plan of accessing the rows that one would prefer it even if thirteen different columns were to be summed, notwithstanding the savings to be achieved by summing all the different columns from the same memory-resident row.

Method	$Cost for 2K ins per I/O	$Cost for 5K ins per I/O	$Cost for 10K ins per I/O
Add from Rows	$8079	$8127	$8207
Projection index	$603	$606	$612
Value-List index	$632	$636	$642
Bit-Sliced index	$418	$421	$425

Table 3.2. Costs of the four plans in dollars under current hardware cost assumptions

3.1.2 Varying Foundset Density and Clustering

Changing the number of rows in the Foundset has little effect on the Value-List index or Bit-Sliced index algorithms, because the entire index must still be read in both cases. However, the algorithms for accessing the rows and using a Projection index entail work proportional to the number of rows in the foundset. We do not consider the plan of accessing from the rows in what follows.

Suppose that the Foundset contains kM rows, but now clustered on a fraction f of the disk space. Both the Projection and Bit-Sliced index algorithms can take advantage of the clustering. The table below shows the comparison between the three index algorithms.

Method	**I/O**	**CPU contributions**
Projection index	$f \cdot 100K$	$I/O + kM \cdot (10\ ins)$
Value-List index	103K	$I/O + 100M \cdot (10\ ins)$
Bit-Sliced index	$f \cdot 69K$	$I/O + f \cdot 197M \cdot (1\ ins)$

Table 3.3. Costs of four plans, I/O and factors of CPU, kM rows, clustering fraction f

Clearly there is a relationship between k and f in Table 3.3, since for k = 100, 100M rows sit on a fraction f = 1.0 of the table, we must have that $k \le f \cdot 100$. Also, if f becomes very small compared to k/100, we will no longer pick up every page in the Projection or Bit-Sliced index. In what follows, we shall assume that f is sufficiently large that the I/O approximations given in Table 3.3 are valid.

The dollar cost of I/O continues to dominate the total dollar cost of the plans under the assumption of Section 3.1.1 that each plan is submitted once every 1000 seconds. For the Projection index, the I/O cost is $f \cdot \$600$. The CPU cost under the assumption that I/O requires 10K instructions is: $((f \cdot 100 \cdot 10{,}000 + k \cdot 1000 \cdot 10)/1{,}000{,}000) \cdot \$12$. Since $k \le f \cdot 100$, the formula $f \cdot 100 \cdot 10{,}000 + k \cdot 1000 \cdot 10 \le f \cdot 100 \cdot 10{,}000 + f \cdot 100 \cdot 1000 \cdot 10 = f \cdot 2{,}000{,}000$. Thus, the total CPU cost is bounded above by $f \cdot \$24$, which is still cheap compared to an I/O cost of $f \cdot \$600$. Yet this is the highest cost we assume for CPU due to I/O, which is the dominant CPU term. In Table 3.4, we give the maximum dollar cost for each index approach.

Method	**$Cost for 10K ins per I/O**
Projection index	$f \cdot \$624$
Value-List index	$642
Bit-Sliced index	$f \cdot \$425$

Table 3.4. Costs of the four plans in dollars, kM rows, clustering fraction f

The clustered case clearly affects the plans by making the Projection and Bit-Sliced indexes more efficient compared to the Value-List index.

3.2 Evaluating Other Single-Column Aggregate Functions

We consider aggregate functions of the following form, where T is a table, C is a column for which we might create a Projection index, Value-List index, or Bit-Sliced index, and AGG is an aggregate function, such as COUNT, MAX, MIN, etc.

```
[3.2]   SELECT AGG(C) FROM T
            WHERE condition;
```

In what follows, we usually consider only aggregate functions that are "Distributive" or "Algebraic" [GBLP96]. A *Distributive* function is a function that can be applied to subsets of the input and the results combined to compute the answer for the entire set. The aggregate functions

COUNT, SUM, MIN, and MAX are distributive aggregate functions. *Algebraic* functions are evaluable using other distributive functions. The function AVG (= SUM/COUNT) is an algebraic aggregate function. The MEDIAN function is neither distributive nor algebraic because it requires all values in the input simultaneously, and it is called *Holistic* in [GBLP96]. As we will see, the three index types have different capabilities in evaluating the different aggregate functions.

Table 3.5 lists a group of aggregate functions and the index types with which we can evaluate these functions. We enter the value "Best" in a cell if the given index type is the most efficient one to have for this aggregation, and "Slow" if the index type works but not most efficiently. Note that Table 3.5 demonstrates how different index types are optimal for different aggregate situations.

Aggregate	**Value-List Index**	**Projection Index**	**Bit-Sliced Index**
COUNT	Not needed	Not needed	Not needed
SUM	Not bad	Good	**Best**
AVG (= SUM/COUNT)	Not bad	Good	**Best**
MIN and MAX	**Best**	Slow	Slow
MEDIAN, N-TILE	Usually Best	Not Useful	Sometimes Best[2]
Column-Product	Very Slow	**Best**	Very Slow

Table 3.5. Tabulation of Performance Evaluating Aggregate Functions by Index Type

The COUNT aggregate on a Foundset B_f can be performed with Algorithm 2.1 and requires no index. The SUM aggregate was covered in Section 3.1. The AVG aggregate can be evaluated as SUM/COUNT, and SUM is what determines the performance under the various Index types.

The MIN and MAX aggregate functions are most efficiently evaluated using a Value-List index. For example, given a Foundset B_f, we loop from the largest value in the Value-List index down to the smallest, until we find one with a row in B_f. The value of the index entry containing that row is the MAX. We can perform the aggregates using a Projection index simply by looping through all values stored. The algorithm to evaluate MAX or MIN using a Bit-Sliced index is rather surprising, and it is given in Appendix A, Algorithm A.1.

MEDIAN and N-TILE Aggregates. The MEDIAN value of a column for a set of rows B_f is the value M such that at least half of the rows have values greater than or equal to M and at least half of the rows have values less than or equal to M. Projection indexes are not useful for evaluating MEDIAN, unless the number of rows in the Foundset is very small indeed: all values in such an index would have to be sorted. To calculate MEDIAN(C) when C is the keyvalue in a Value-List index, one loops through the non-null values of C in order, keeping a count of rows encountered that fall in B_f AND B_{nn}, until for the first time with some value v the number of rows encountered so far is greater than the count of all rows in (B_f AND B_{nn}) divided by 2. Then M = v. A Bit-Sliced index can also be used to determine the MEDIAN, in about the same amount of time as it takes to determine SUM. (See Appendix A, Algorithm A.2, for the method.) The algorithm using the Value-List index, on the other hand, requires about half the effort of SUM (assuming that the MEDIAN value is located about half-way through the Value-List index loop). From Table 3.4, we see the Bit-Sliced index approach can be the more efficient, but only if the rows of B are clustered in a local region, a fraction f of the pages, and $f \cdot 425 \leq 642/2$, or $f \leq 0.755$. Note that the algorithms for evaluating MEDIAN using Value-List or Bit-Sliced indexes can both provide not only the median value but also the set of rows that is equal to the median (or by simple modification, greater than or equal to, strictly greater than, less than or equal to, or strictly less than).

[2]Best only if there is a clustering of rows in B in a local region, a fraction f of the pages, $f \leq 0.755$.

The N-TILE aggregate function determines values $v_1, v_2, \ldots, v_{N-1}$, which partition the rows in B_f into N sets of (approximately) equal size based on the interval in which their C value falls: $C \le v_1$, $v_1 < C \le v_2$, . . ., $v_{N-1} < C$. MEDIAN in this nomenclature is a 2-TILE. Generalization of the algorithms for MEDIAN are immediate, and the set of rows in each interval is easily returned by the algorithms. The effort required by the Bit-Sliced algorithm varies in proportion to N, whereas the Value-List index only becomes about twice as expensive (since (N-1)/N values probably need to be examined).

Column-Product Aggregates. An example of a COLUMN-PRODUCT aggregate function is VARIANCE, where SUM(C*C) must be calculated for all rows in the foundset B_f. An aggregation might also involved the product of different columns. For example, in the TPC-D benchmark, the LINEITEM table has columns L_EXTENDEDPRICE (the price for the ordered quantity of some product listed in the line-item for a particular order) and L_DISCOUNT (the discount on this price extended to the customer). A large number of queries in TPC-D (Q1, Q3, Q5, Q6, Q10) retrieve the aggregate: SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)), usually given the column alias "REVENUE", or in Q1, "SUM_DISC_PRICE".

The most efficient method for calculating Column-Product Aggregates uses Projection indexes for the columns involved. It is possible to calculate products of columns using Value-List or Bit-Sliced indexes, using the same sort of algorithm that was used for SUM. In Appendix A, Algorithm A.3 shows how such a product calculation can be performed with a Value-List index. For both Value-List and Bit-Sliced indexes, Foundsets of all possible cross-terms of values must be formed and counted, so the algorithm are terribly inefficient.

An important possibility for this sort of Aggregate is to create a *Virtual Column* index on the table T. A Virtual Column index is created as if on a "virtual column" of a table, with values that are calculated as some formula f( ) on columns in each single row of T. For example, we could imagine creating an index on the formula f(T.A, T.B) = T.A * T.B. The virtual column is not actually placed the table, so the logical and physical design of the table is unaffected, but the new index allows the SQL query optimizer to perform one of the algorithms described in Section 3.1 or 3.2, depending upon whether the index created was Value-List, Bit-Sliced, or Projection. (Virtual Value-List indexes can be defined now using Informix DSS Indexes [INF96].) Naturally, we cannot foresee all possible functional aggregate needs, but there are often several functions which occur often enough in a query workload to justify a Virtual Column index. The need in TPC-D to calculate SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) is such a case. Note that any change in the logical or physical design of database is disallowed by the TPC-D Specification, but a number of products are already permitting index-only query evaluation in some cases.

4. Evaluating Range Predicates

Consider a Select statement of the following form:

```
[4.1]   SELECT target-list FROM  T
            WHERE C-range AND <condition>;
```

Here, C is a column of T, and <condition> is a general Where clause condition resulting in a Foundset B_f. C-range represents a range predicate, {C > c1, C >= c1, C = c1, C >= c1, C > c1, or C between c1 and c2}, where c1 and c2 are constant values. We will demonstrate in what follows how to further restrict the Foundset B_f, creating a new Foundset B_F, so that the compound predicate "C-range AND <condition>" holds for exactly those rows contained in B_F. We do this with varying assumptions regarding index types on the column C.

Evaluating the Range using a Projection Index. If there is a Projection index on C, we can create B_F in a straightforward way by accessing each C value in the index corresponding to an ordinal row number in B_f and testing whether it lies within the specified range.

Evaluating the Range using a Value-List Index. In the case of a Value-List index, the C-range restriction of [4.1] uses an algorithm common in most database system products. Since we have a Bitmap capability, we will make a slight variation by accumulating a Bitmap B_r as an OR of all row sets in the index for values that lie in the specified range, then AND the result with B_f.

Algorithm 4.1. Range Predicate with a Value-List Index

B_r = the empty set
For each entry v in the index for C that satisfies the range specified
  Designate the set of rows with the value v as B_v
  $B_r = B_r$ OR B_v
$B_F = B_f$ AND B_r
u

Note that for Algorithm 4.1 to be efficiently performed, we must find some way to guarantee that the Bitmap B_r remains in memory at all times as we loop through the values v in the range. This requires some forethought in the Query Optimizer if the table T being queried is large: 100 million rows will mean that a Bitmap B_r of 12.5 MBytes must be kept resident.

Evaluating the Range using a Bit-Sliced Index. Rather surprisingly, it is possible to evaluate range predicates efficiently using a Bit-Sliced index. Given a Foundset B_f, we will demonstrate how to evaluate the set of rows B_{GT} such that C > c1, B_{GE} such that C >= c1, B_{EQ} such that C = c1, B_{LE} such that C <= c1, B_{LT} such that C < c1.

Algorithm 4.2. Range Predicate with a Bit-Sliced Index

$B_{GT} = B_{LT}$ = the empty set; $B_{EQ} = B_{nn}$
For each Bit-Slice B_i for C from most to least significant
  If bit i is on in constant c1
    $B_{LT} = B_{LT}$ OR (B_{EQ} AND NOT(B_i))
    $B_{EQ} = B_{EQ}$ AND B_i
  else
    $B_{GT} = B_{GT}$ OR (B_{EQ} AND B_i)
    $B_{EQ} = B_{EQ}$ AND NOT(B_i)
$B_{EQ} = B_{EQ}$ AND B_f; $B_{GT} = B_{GT}$ AND B_f; $B_{LT} = B_{LT}$ AND B_f
$B_{LE} = B_{LT}$ OR B_{EQ}; $B_{GE} = B_{GT}$ OR B_{EQ}
u

Naturally, we can drop Bitmap calculations in Algorithm 4.2 that do not evaluate the condition we seek. For example, if we only need to evaluate C > c1 or C >= c1, we do not need any steps that evaluate B_{LE} or B_{LT}.

Proof that B_{EQ} B_{GT} and B_{GE} are properly evaluated. The method to evaluate B_{EQ} clearly determines all rows with C = c1, since it requires that all 1-bits on in c1 be on and all 0-bits 0 in c1 be off for all rows in B_{EQ}. Next, we note that B_{GT} is created as the OR of a group of Bitmaps with certain conditions, which we now describe.

Assume that the bit representation of c1 is $b_N b_{N-1} \ldots b_1 b_0$, and that the bit representation of C for some row r in the database is $r_N r_{N-1} \ldots r_1 r_0$. For each bit position i from 0 to N with bit b_i off in c1, a row r will be in B_{GT} if bit r_i is on and bits $r_N r_{N-1} \ldots r_1 r_{i+1}$ are all equal to bits $b_N b_{N-1} \ldots b_{i+1}$. It is clear that C > c1 for any such row r in B_{GT}. Furthermore for any value of C > c1, there must be some bit position i such that the i-th bit position in c1 is off, the i-th bit position of C is on, and all more-significant bits in the two values are identical. Therefore, Algorithm 4.2 properly evaluates B_{GT}.u (The proof that B_{LT} and B_{LE} are properly evaluated is similar.)

4.1 Comparing Algorithm Performance

Now let us compare the performance of the three algorithms to evaluate a range predicate, such as "C between c1 and c2". We assume that C is not a clustering value for the table. The cost of evaluating a range predicate using a Projection index is similar to that of evaluating SUM using a Projection index and a given Foundset. We need the I/O to access each of the index pages with C values plus the CPU cost to test each value and, if the row passes the range test, to turn on the appropriate bit in a Foundset.

As we have just seen, it is possible to determine the Foundset of rows in a range using Bit-Sliced indexes: note that we can calculate the range predicate c2 >= C >= c1 using a Bit-Sliced index by calculating B_{GE} for c1 and B_{LE} for c2, then ANDing the two. Once again the calculation is generally comparable in cost to calculating a SUM aggregate, Algorithm 3.2 of Example 3.1.

With a Value-List index, the effort is proportional to the width of the range, but for a wide range, it is also comparable to the effort needed to perform SUM for a large Foundset. Therefore for wide ranges the Projection and Bit-Sliced indexes have a performance advantage. For short ranges the work to perform the Projection and Bit-Sliced algorithms remain nearly the same (assuming the range variable is not a clustering value) while the work to perform the Value-List algorithm is approximately proportional to the number of rows found in the range. Eventually as the width of the range decreases the Value-List algorithm is the better choice. These considerations are summarized in Table 4.1.

Range Evaluation	**Value-List Index**	**Projection Index**	**Bit-Sliced Index**
Narrow Range	**Best**	Good	Good
Wide Range	Not bad	Good	**Best**

Table 4.1. Performance of Range Evaluation by Index Type

4.2 Range Predicate Using a Bit-Sliced Index with a Non-Binary Base

For many years, MODEL 204 has used a form of indexing to evaluate range predicates known as "Numeric Range" [M204]. Numeric Range performs range evaluation similar to the Bit-Sliced Algorithm 4.2, except that numeric quantities were expressed in a larger base (base 10). It turns out that the effort of performing a range retrieval can be reduced if we are willing to store a larger number of Bitmaps. We give an example below that generalizes the Bit-Sliced algorithm to an octal representation. Sybase IQ was the first product to demonstrate in practice that the same Bit-Sliced index, called the "High NonGroup Index" [EDEL95], could be used both for evaluating range predicates and performing Aggregates.

Definition 4.1. Bit-Sliced Indexes with a non-binary base. We denote the representation of the column value C or the constant c1 by $O_K O_{K-1} \ldots O_1 O_0$, where each O_i is an octal digit: 0, 1, . . ., 7. Thus the value represented can be rewritten as: $O_K \cdot 8^K + O_{K-1} \cdot 8^{K-1} + \ldots O_1 \cdot 8^1 + O_0 \cdot 8^0$. We assume also that we have in existence Bitmaps of the following descriptions:

[4.2] B_i^d is the Foundset of all rows such that the octal representation for C has $O_i >= d$

Thus, for example, B_5^3 will contain rows whose C value has $O_5 \geq 3$. We designate the set of all Bitmaps of this form, $(B_0^1, B_0^2, \ldots, B_0^7, B_1^1, B_1^2, \ldots, B_1^7, \ldots, B_K^1, B_K^2, \ldots, B_K^7)$ together with B_{nn} and B_n, to be a *Bit-Sliced index with base 8*. Note that for d=0, it is not necessary to provide a Bitmap of the form B_i^0, since all non-null rows have a C value greater than or equal to 0, and this means that is identical to B_{nn}, which is already present. u

Note that a Bit-Sliced index with base 2 has the same form of definition as the one in [4.2]. Consider the binary representation of C, $b_N b_{N-1} \ldots b_1 b_0$. We can define the binary bit-slice B_i:

B_i is the Foundset of all rows such that the binary representation for C has $b_i >= 1$

If we consider the column C to have a binary representation, $b_N b_{N-1} \ldots b_1 b_0$, as well as the octal one of Definition 4.1, then the octal digits in the octal representation can be represented in binary terms:

$O_0 = b_2 b_1 b_0$; $O_1 = b_5 b_4 b_3$; . . .; $O_i = b_{3i+2} b_{3i+1} b_{3i}$

Note that with this equivalence in mind, we can represent all of the Bit-Slices of the binary Bit-Sliced index as Bit-Slices of the octal Bit-Sliced index.

$B_{3i} = B_i^1$; $B_{3i+1} = B_i^2$; $B_{3i+2} = B_i^4$

For example, the set of rows whose octal digit O_1 is greater than or equal to 4 is the same as the set of rows whose binary digit b_5 is on. This representation shows that we can create a Bit-Sliced index of base 8 from a Bit-Sliced binary index simply by adding (approximately) 5 new Bitmaps for every three that exist in the Bit-Sliced binary index.

The value of having a higher-base Bit-Sliced index is that we can perform ranges more efficiently. Given a Foundset Bitmap B_f, Algorithm 4.3 will evaluate, using a Bit-Sliced index of base 8, the set of rows B_{GT} such that C > c1, B_{GE} such that C >= c1, B_{EQ} such that C = c1, B_{LE} such that C <= c1, B_{LT} such that C < c1. Assume c1 is represented as $O_K O_{K-1} \ldots O_1 O_0$.

Algorithm 4.3. Range Predicate with a Bit-Sliced Index of Base 8

```
B_GT = B_LT = the empty set; B_EQ = B_nn
For each octal digit O_i in c1 from most to least significant
   If O_i > 0
      B_LT = B_LT OR (B_EQ AND NOT(B_i^{O_i}))
      If O_i < 7
         B_GT = B_GT OR B_i^{O_i+1}
         B_EQ = B_EQ AND B_i^{O_i} AND NOT(B_i^{O_i+1})
      else                                  /* O_i = 7                  * /
         B_EQ = B_EQ AND B_i^{O_i}
   else                                     /* O_i = 0                  * /
      B_GT = B_GT OR (B_EQ AND B_i^1)
      B_EQ = B_EQ AND NOT(B_i^{1O})
B_EQ = B_EQ AND B_f; B_GT = B_GT AND B_f; B_LT = B_LT AND B_f
B_LE = B_LT OR B_EQ; B_GE = B_GT OR B_EQ
```
u

As we can see, the amount of work to calculate B_{EQ} is somewhat less with a base 8 Bit-Sliced index than it is with a binary Bit-Sliced index (Algorithm 4.2). Instead of performing a Bitmap assignment for each bit in a binary representation, assignments are only needed for every octal number (although for 1 <= O_i <= 6, two Bitmap operations instead of 1 are actually needed in the assignment: $B_{EQ} = B_{EQ}$ AND B_i^{Oi} AND NOT(B_i^{Oi+1}). Calculating B_{GT} also is performed with less work, although the analysis is somewhat more complex and we omit it. If we were to use base 16 or base 32 Bit-Sliced indexes, the amount of CPU resources would continue to decrease. While higher bases will entail more disk utilization and greater load times, data warehouses that are infrequently refreshed and make heavy use of long range predicates can make good use of higher-base Bit-Sliced indexes.

5. Evaluating OLAP-style Queries

Figure 5.1 pictures a star-join schema with a central fact table, Sales, containing sales data, together with dimension tables known as TIME (when the sales are made), PRODUCT (product sold), and CUSTOMER (purchaser in the sale). Most OLAP products do not express their queries in SQL, but much of the work of typical queries could be represented in SQL [GBLP96] (although more than one query might be needed).

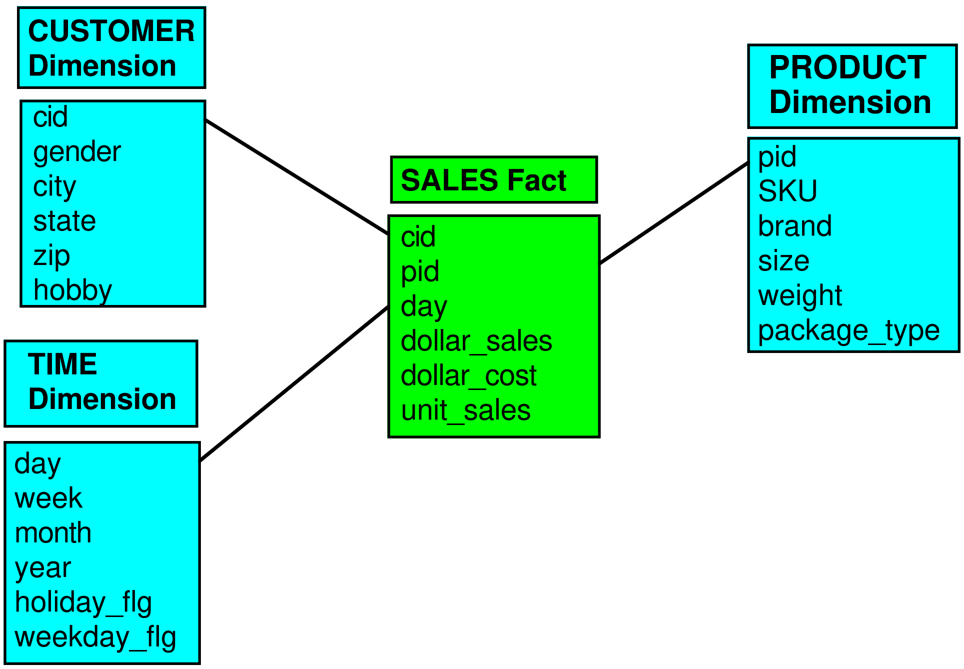

Figure 5.1. Star Join Schema of `SALES`, `CUSTOMER`, `PRODUCT`, and `TIME`

Query 5.1 retrieve total dollar sales that were made for a brand of products during the past 4 weeks to customers in New England.

```
[5.1] SELECT P.brand, T.week, C.city, SUM(S.dollars)
      FROM SALES S, PRODUCT P, CUSTOMER C, TIME T
      WHERE S.day = T.day and S.cid = C.cid and S.pid = P.pid
         and P.brand = :brandvar and T.week >= :datevar and
         C.state in ('Maine', 'New Hampshire', 'Vermont', 'Massachusetts', 'Connecticut',
            'Rhode Island')
      GROUP BY P.brand, T.week, C.city;
```

An important selling point of OLAP products is that they can evaluate queries of this type quickly, even though the fact tables are usually very large. The OLAP approach to the problem is to pre-calculate results of some Grouped queries and store them in what we have referred to as *summary tables*. For example, we might create a summary table where sums of Sales.dollars and sums of Sales.quantity_sold are precalculated for all combination of values at the lowest level of divisibility in the dimensions, e.g., for C.cid values, T.day values, and P.pid values. Within each dimension there are also hierarchies sitting above the lowest level of divisibility. A week has 7 days and a year has 52 weeks, and so on. Similarly, a customer might be an individual or a retail store that orders from your Wholesale business (with a gender of NA), having an address with a geographic hierarchy, city and state. Although we have only precalculated a summary table at the lowest dimensional level, there could still be many rows of data associated with a particular cid, day, and pid (a store that reorders frequently), or there might also be none. A summary table on the lowest level of divisibility will normally allow us to provide quicker responses for queries that group by attributes at higher levels of the dimensional hierarchy, such as city (of customers), week, and brand. And we are also allowed to create other summary tables with some of the dimensions representing higher-level dimensional attributes. The higher the dimensional levels, the fewer elements that will be in the summary table, but there are a lot of possible combinations of hierarchies. Luckily, we don't need to create all possible summary tables in order to speed up the queries a great deal. For more details, see [STG95, HRU96].

By doing the aggregation work beforehand, summary tables provide quick response to queries, so long as any selection conditions are restrictions on the dimensions that have been foreseen in advance. But if the dimensions should have a large number of (non-hierarchical) attributes that could figure in restrictions (such as days with temperature above 90 degrees) then summary tables that do not permit such restrictions independently of all other dimensions will be useless. To put it another way, the size of data in the summary tables grows as the product of the number of independent restrictions, and very soon it becomes impossible to include all of them. The goal of this paper is thus to describe and analyze variant indexing structures that are useful for evaluating OLAP-style queries quickly, even when the queries are completely ad-hoc. Before continuing we need to explain Join indexes.

5.1 Join Indexes

Definition 5.1. Join Index. A Join index is an index on one table for a quantity that involves a column value of a different table through a commonly encountered join ([VALD87], [O'NGG95]).u

For example, the Star Join index, which was invented a number of years ago, concatenates column values (or rather ordinal encodings of those values) from different dimension tables of a Star schema, and lists RIDs in the central fact table for each concatenated value. The Star Join index was the best approach known in its day, but there is a problem with it that is comparable to the problem with summary tables. If there are numerous columns used for restrictions in each dimension table, then the number of Star Join indexes needed to be able to combine any single column choice from each dimension table is a product of the number of columns in each dimension. There will be a "combinatorial explosion" of Join Indexes in terms of the number of useful columns.

The Bitmap join index, defined in [O'NGG95], addresses this problem. In its simplest form, this is an index on a table T based on a single column of a table S, where S commonly joins with T in a specified way. For example, in the TPC-D benchmark database, the O_ORDERDATE column is a column of the ORDER table, but in two queries (Q5 and Q8) we need to join ORDER with LINEITEM to restrict LINEITEM rows to a range of O_ORDERDATE. This can easily be accomplished by creating an index for the value ORDERDATE on the LINEITEM table. Note that this does not change the design of the LINEITEM table, since the index on ORDERDATE is for a virtual column through a join. The join between rows is never going to vary: LINEITEM is a weak entity dependent on ORDER. Obviously the number of indexes of this kind increases linearly with the number of useful columns in the dimension tables. We depend on the speed of combining Bitmapped indexes to create ad-hoc combinations, and thus the explosion of Star Join indexes because of different combinations of dimension columns is not a problem. Another way of looking at this is that Bitmap join indexes are *Recombinant*, whereas Star join indexes are not.

Join indexes can be of any type: Projection, Value-List, or Bit-Sliced. To speed up Query [5.1], it would be possible to create Join indexes on the SALES fact table for columns in the dimensions. If join indexes have been created on all dimension table columns mentioned in the query, then explicit joins with dimension tables may no longer be necessary at all. Using Value-List or Bit-Sliced join indexes we can evaluate the selection conditions in the Where Clause, and using Bit-Sliced or Projection indexes we can then retrieve the dimensional values into the answer set.

5.2 Calculating Groupset Aggregates

We assume that in star-join queries like [5.1], the aggregation is performed on columns of the central Fact table, F. There is a Foundset of rows on the Fact table, and the Group By columns will be columns in the Dimension tables D1, D2, . . . (they might be primary keys of the Dimension tables, in which case they will also exist as foreign keys on F). Once the Foundset has been computed from the Where Clause, the bits in the Foundset must be partitioned into groups, which we call *Groupsets*, again sets of rows from F. Any aggregate functions then need to be evaluated separately over these different Groupsets. In what follows, we describe how to compute Groupset aggregates using our different index types.

Computing Groupsets Using Projection Indexes. We assume Projection indexes exist on F for each of the group-by columns (these will be Join Indexes, since the group-by columns are on the Dimension tables), and also for all columns of F involved in aggregates. If the number of group cells is small enough so that all grouped aggregate values in the target list will fit into memory, then partitioning into groups and computing Algebraic or Distributive aggregate functions for each group can usually be done rather easily.

For each row of the Foundset returned by the Where clause, classify the row into a group-by cell by reading the appropriate Projection indexes on F. Then read the values of the columns to be aggregated from Projection indexes on these columns, and aggregate the result into the proper cell of the memory-resident array. (This approach can be used for Distributive functions with the aggregate calculated so far and the new value; for Algebraic functions, such as AVG(C), it can be done by accumulating a "handle" of Distributive results, such as SUM(C) and COUNT(C), to calculate the final aggregate.)

If the total set of cells in the Group by cannot be retained in a memory-resident array, then a variant algorithm can be devised, similar to Hash join, in which multiple passes are used to calculate each successive portion of the array that can be maintained in memory. Alternatively, the values to be aggregated can be tagged with their group cell values, and then values with identical group cell values brought together using a disk sort (this is a common method used today).

Computing Groups Using Value-List Indexes. The idea of using Value-List indexes to compute aggregate groups is not new. As mentioned in Example 2.1, Model 204 used them years ago. In this section we formally present the concepts hinted at in Example 2.1.

Algorithm 5.1. Grouping by D1.A, D2.B using a Value-List Index

```
For each entry v1 in the Value-List index for D1.A
    For each entry v2 in the Value-List index for D2.B
        Bg = Bv1 AND Bv2 AND Bf
        Evaluate AGG(F.C) on Bg (normally, we would do this with a Projection index)
```

u

Algorithm 5.1 presents an algorithm for computing aggregate groups that the Query Optimizer will perform for ad-hoc queries with two group-by columns (Bitmap Join Value-List indexes on Dimension tables D1 and D2). The generalization of Algorithm 5.1 to the case of n group-by attributes is straightforward. We assume the Where clause condition already performed resulted in the Foundset B_f on the Fact table F. The algorithm generates a set of Groupsets, B_g, one for each

(D1.A, D2.B) group. The aggregate function AGG(F.C) is evaluated for each group using B_g in place of B_f.

The Evaluation of AGG(F.C) using Algorithm 5.1 can be quite inefficient when there are a lot of Groupsets, and rows in each Groupset are spread out at random throughout the table F. The reason is that the aggregate function must be re-evaluated for each group, and we lose the ability to cluster work on Projection index disk pages, an important issue if we assume that the Projection index for the column F.C is too large to be cached in memory. With many Groupsets, we would expect there to be few rows in each, and to evaluate AGG(F.C) might require an I/O for each individual row.

5.3 Improved Grouping Efficiency Using Segmentation and Clustering

In this section we show how segmentation and clustering can be used to speed up any query with one or more group-by attributes, using a generalization of Algorithm 5.1. The idea of segmentation was first introduced in Section 2.1. Rather than evaluating a query for all rows of a table at once, the rows are partitioned into Segments. Query evaluation is performed on one Segment at a time, and the results from evaluating each Segment are combined at the end to form the final query result. Segmentation is most effective when the number of rows per Segment is the number of bits that will fit on a disk page. Setting the Segment size this way allows us to read the bits in an index entry that correspond to a segment by performing a single disk I/O.

As pointed out earlier, if a Segment s_1 of the Foundset (or Groupset) is completely empty (i.e., all bits are 0), then ANDing s_1 with any other Segment s_2 will also result in an empty Segment, so for any algorithm evaluating the expression s_1 AND s_2, Segment s_2 need not be read and the AND need not be performed. Indeed, as explained in [O'NEI87], the entry in the B-tree leaf level for a column C that references an all-zeros Bitmap Segment can be flagged (or simply missing), and a reasonable algorithm to AND Bitmaps will test this before accessing any Segment Bitmap pages. Thus neither s_1 nor s_2 will need be read from disk after this phase of evaluation. This optimization becomes especially useful when rows are clustered on disk by nested dimensions used in grouping, as we shall see.

Consider a Star Join schema with a central fact table F and a set of dimension tables, $D_1, D_2, \ldots, D_k$. Each dimension D_m, $1 \le m \le k$, has a primary key, d_m, with a domain of values having an order assigned by the DBA. We represent the number of values in the domain of d_m by n_m, and list the values of d_m in increasing order, differentiated by superscript, as: $d_m^1, d_m^2, \ldots, d_m^{n_m}$. For example, the primary key of the TIME dimension of Figure 5.1 would have natural temporal order. The DBA would probably choose the order of values in the PRODUCT dimension so that the most commonly used hierarchies, such as brand or product_type, consist of contiguous values in the dimensional order.

In what follows, we will be considering a Workload of OLAP-type queries which have Group By clauses on some values in the dimension tables (not necessarily the primary key values). The fact table F contains foreign key columns that match the primary keys of the various dimensions. We will assume indexes on these foreign keys for table F and make no distinction between these and the Join Indexes on primary keys of the Dimensions. We intend to demonstrate how these indexes can be efficiently used to perform Group By queries using Algorithm 5.1.

We wish to create a clustering for the fact table F to improve performance of the most finely divided Group By possible (grouping by primary key values of the dimensions rather than by any hierarchy values above these). It will turn out that this clustering is also effective for arbitrary Group By queries on the dimensions. To evaluate the successive Groupsets by Algorithm 5.1, we consider performing the nested loop of Figure 5.2.

In the loop of Figure 5.2, we assume $(D_1, D_2, \ldots, D_k)$ is placed in some order determined by the DBA (this order has long-term significance; we give an example below). The loop on dimension values here produces conjoint cells $(v_1, v_2, \ldots, v_k)$, $v_j \in D_j$, of this most finely divided Group By. Each cell may contain a large number of rows from table F or none. The set of rows in a particular cell is what we have been referring to as a Groupset.

```
For each key-value v1 in order from D1
  . . .
      For each key-value vk in order from Dk
         <Here, we calculate aggregates for cell v1, v2, . . . , vk>
      End For vk
  . . .
End For v1
```

Figure 5.2. Nested Loop to Perform a Group By

It is our design to *cluster the rows of the fact table F* so that all the rows with foreign keys matching the dimension values in each cell $(v_1, v_2, \ldots, v_k)$ *are placed together on disk*, and furthermore that the successive cells fall in the same order as the nested loop above on $(D_1, D_2, \ldots, D_k)$. Naturally, the Bitmap ordinal numbers are in the same order as the rows on disk.

Given this clustering, the Bitmaps for each Groupset will have 1-bits in a limited contiguous range. Furthermore, as the loop is performed to calculate a Group By, successive cells will have rows in Groupset Bitmaps that are contiguous one to another and increase in ordinal row number. Figure 5.3 gives a schematic representation of the Bitmaps for index values of three dimensions. The Groupset Bitmaps, of course, are calculated by ANDing the appropriate index Bitmaps for the given values. Note that as successive Groupset Bitmaps in loop order are generated from ANDing, the 1-bits in each Groupset move from left to right.

```
D1 = d1^1   1111111111111111111111111111111111000000000000000000000000000000000...
   = d1^2   0000000000000000000000000000000000111111111111111111111111111111000...
   .  .  .
D2 = d2^1   1111111100000000000000000000000000111111110000000000000000000000111...
   = d2^2   0000000011111111000000000000000000000000001111111100000000000000000...
   .  .  .
D3 = d3^1   1100000011000000000000000000000000110000000000000000000000000000110...
   = d3^2   0011000000110000000000000000000000001100000000000000000000000000001...
   .  .  .
   = d3^n3  0000001100000011000000000000000000000000110000000000000000000000001...
```

Figure 5.3. Schematic Representation of Dimension index Bitmaps for Clustered F

In terms of Figure 5.3, the Groupset for the first cell $(d_1{}^1, d_2{}^1, d_3{}^1)$ will have a Bitmap AND of the three index Bitmaps $D_1 = d_1{}^1$, $D_2 = d_2{}^1$, and $D_3 = d_3{}^1$, as follows.

```
1100000000000000000000000000000000000000000000000000000000000000000...
```

The Groupset for the next few cells will have Bitmaps:

```
0011000000000000000000000000000000000000000000000000000000000000000...
0000110000000000000000000000000000000000000000000000000000000000000...
```

And so on, moving from left to right.

To repeat: as the loop to perform the most finely divided Group By is performed, and Groupset Bitmaps are generated, successive blocks of 1-bits by ordinal number will be created, and successive row values from the Projection index will be accessed to evaluate an aggregate. Because of Segmentation, no unnecessary I/Os are ever performed to AND the Bitmaps of the individual dimensions. Indeed, due to clustering, it is most likely that Groupset Bitmaps for successive cells will have 1-bits that move from left to right on each Segment Bitmap page of the Value index, and the column values to aggregate will move from left to right in each Projection index page, only occasionally jumping to the next page. This is tremendously efficient, since relevant pages from the Value-list dimension indexes and Projection indexes on the Fact table are read only once to perform the entire Group By.

If we consider Group By queries where the Groupsets are less finely divided than in the primary key loop given, grouping instead by higher hierarchical levels in the dimensions, this approach should still work. We materialize the Grouped Aggregates in memory, and continue to loop in nested loop order by the primary keys of the dimensions as we examine rows in F. Now for each cell, $(v_1, v_2, \ldots, v_k)$ in the loop of Figure 5.2, we determine the higher order hierarchy values of the Group By we are trying to compute. Corresponding to each dimension primary key value of the current cell, $v_i = d_i^m$, there is a value in the dimension hierarchy we are grouping by h_i^r; thus, as we loop through the finely divided cells, we aggregate the results for $(d_1^{m_1}, d_2^{m_2}, \ldots, d_k^{m_k})$ into the aggregate cell for $(h_1^{r_1}, h_2^{r_2}, \ldots, h_k^{r_k})$. We have lost none of the nested loop efficiency. In a similar manner, if we were to group by only a subset of dimensions, we would be able to treat all dimensions not named as the highest hierarchical level for that dimension, which we refer to as ALL, and so continue to use this nested loop approach.

5.4 Groupset Indexes

While Bitmap Segmentation permits us to use normal Value-List indexing, ANDing Bitmaps (or RID-lists) from individual indexes to find Groupsets, there is a certain amount of inefficiency associated with calculating which Segments have no 1-bits for a particular Cell when they have representation in some dimensions and not in others. In Figure 5.1, for example, the cell (d_1^1, d_2^1, d_3^1) has only the leftmost 2 bits on, but the Value-List index Bitmaps for these values have many other segments with bits on, as we see in Figure 5.3, and the Bitmaps for the individual index values might have 1-bits that span many Segments.

To reduce this overhead, we suggest creating a special *Groupset index*, whose keyvalues are a concatenation of the dimensional primary-key values. Since the Groupset Bitmaps in nested loop order are represented as successive blocks of 1-bits in ordinal number, the Groupset index value can be represented by a simple integer, which represents the starting position of the first 1-bit in the Groupset, and the ending position of that Bitmap can be determined as one less than the starting position for the following index entry. Some cells will have no representative rows, and this will be most efficiently represented in the Groupset index by the fact that there is no value representing a concatenation of the dimensional primary-key values.

We believe that the Groupset index makes the calculation of a multi-dimensional Group By as efficient as possibly, without summary tables where aggregates are precalculated. As we pointed out earlier, a precalculated aggregate loses flexibility when a Foundset of rows must be aggregated in the Group By, and the criteria that determined this Foundset has not been foreseen in specially created summary tables. Furthermore, because of combinatorial explosion of possibilities, not all Foundset criteria can be foreseen. The Groupset index adds back this flexibility.

6. Conclusion

The read-mostly environment of data warehousing has made it feasible to use more complex index structures to speed up the evaluation of queries. This paper has introduced two index structures: Bit-Sliced indexes and Projection indexes. The two index structures have appeared in commercial systems — a user-defined bit-sliced capability was possible in MODEL 204, and both Bit-Sliced and Projection indexes are provided by Sybase IQ. But this paper constitutes the first rigorous examination of these index structures in the literature. Their performance is examined for evaluating a variety of aggregate functions and (range) predicates, and in Table 3.5 we show circumstances in which type of index out-performs the others.

As a new contribution, we have examined how OLAP-style queries involving aggregation and grouping can be efficiently evaluated using indexing and clustering, and we have introduced a third index type, Groupset indexes, that are especially well-suited for evaluating this type of query. Using indexes, as opposed to precalculated summary tables, to evaluate OLAP-style queries allows more flexible queries, having ad-hoc selection conditions, to be evaluated efficiently.

References

[COMER] Comer, D. The Ubiquitous B-tree. Comput. Surv. 11 (1979), pp. 121-137.

[EDEL95] Herb Edelstein. Faster Data Warehouses. Information Week, Dec. 4, 1995, pp. 77-88. Give title and author on http://www.techweb.com/search/advsearch.html.

[FREN95] Clark D. French. "One Size Fits All" Database Architectures Do Not Work for DSS. Proceedings of the 1995 ACM SIGMOD Conference, pp. 449-450.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Proceedings of the 12th International Conference on Data Engineering, pp. 152-159, 1996.

[GP87] Jim Gray and Franco Putzolu. The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. Proceedings of the 1987 ACM SIGMOD Conference, pp. 395-398.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing Data Cubes Efficiently. Proceedings of the 1996 ACM SIGMOD Conference, pp. 205-216.

[INF96] INFORMIX-OnLine Extended Parallel Server: A New Generation of Decision Support Indexing. Informix White Paper, 1996.

[KIMB96] Ralph Kimball. The Data Warehouse Toolkit. John Wiley & Sons, 1996.

[M204] MODEL 204 File Manager's Guide, Version 2, Release 1.0, April 1989, Computer Corporation of America.

[O'NEI87] Patrick O'Neil. Model 204 Architecture and Performance. Springer-Verlag Lecture Notes in Computer Science 359, 2nd International Workshop on High Performance Transactions Systems (HPTS), Asilomar, CA, 1987, pp. 40-59.

[O'NEI91] Patrick O'Neil The Set Query Benchmark. The Benchmark Handbook for Database and Transaction Processing Systems, Jim Gray (Editor), Morgan Kaufmann, 2nd Edition, 1993, pp. 359-395.

[O'NEI96] Patrick O'Neil. Database: Principles, Programming, and Performance. Morgan Kaufmann, 3rd corrected printing, 1996.

[O'NGG95] Patrick O'Neil and Goetz Graefe. Multi-Table Joins Through Bitmapped Join Indices. SIGMOD Record, September, 1995, pp. 8-11.

[PH96] David A. Patterson and John L. Hennessy. Computer Architecture, A Quantitative Approach. Morgan Kaufmann, 2nd Edition, 1996.

[STG95] Stanford Technology Group, Inc., An INFORMIX Company. Designing the Data Warehouse on Relational Databases. Informix White Paper, 1995, http://www.informix.com.

[TPC] TPC Home Page. Descriptions and results of TPC benchmarks, including the TPC-C and TPC-D benchmarks. http://www.tpc.org.

[VALD87] Patrick Valduriez. Join Indices. ACM TODS, Vol. 12, No. 2, June 1987, Pages 218-246.

Appendix A. Some Algorithms for Evaluation

Algorithm A.1. Evaluating MAX(C) with a Bit-Sliced Index

```
B_current = B_f AND B_nn; MAX = 0
If (COUNT(B_current) == 0)
    Return null;
For each Bit-Slice B_i for C from most to least significant
    If COUNT(B_i AND B_current) > 0          {
        B_current = B_i AND B_current
        MAX += 2^i
    }
Return MAX
```

u

Algorithm A.2. Evaluating MEDIAN(C) with a Bit-Sliced Index

```
B_current = B_f AND B_nn;  totcount = COUNT(B_current); SoFar = 0; MEDIAN = 0
If (totcount == 0)
    Return null;
For each Bit-Slice B_i for C from most to least significant  {
    If SoFar + COUNT(B_current AND NOT(B_i)) <= totcount/2
        SoFar += COUNT(B_current AND NOT(B_i))
        B_current = B_current AND B_i
        MEDIAN += 2^i
    }
    Else
        B_current = B_current AND NOT(B_i)
Return MEDIAN
```

u

Algorithm A.3. Evaluating SUM(T.A * T.B) with a Value-List Index

```
SUM = 0
For each entry v1 in the Value-List index for T.A
    For each entry v2 in the Value-List index for T.B
        SUM += v1 * v2 * COUNT(B_f AND B_v1 AND B_v2)
```

u