A Mapping Mechanism to Support Bitmap Index and Other Auxiliary Structures on Tables Stored as Primary B$^+$-trees

Eugene Inseok Chong, Jagannathan Srinivasan, Souripriya Das
Chuck Freiwald, Aravind Yalamanchi, Mahesh Jagannath
Anh-Tuan Tran, Ramkumar Krishnan, Richard Jiang

Oracle Corporation
One Oracle Drive, Nashua, NH 03062, USA

Abstract

Any auxiliary structure, such as a bitmap or a B$^+$-tree index, that refers to rows of a table stored as a primary B$^+$-tree (e.g., *tables with clustered index* in Microsoft SQL Server, or *index-organized tables* in Oracle) by their physical addresses would require updates due to inherent volatility of those addresses. To address this problem, we propose a mapping mechanism that 1) introduces a single *mapping table*, with each row holding one key value from the primary B$^+$-tree, as an intermediate structure between the primary B$^+$-tree and the associated auxiliary structures, and 2) augments the primary B$^+$-tree structure to include in each row the physical address of the corresponding mapping table row. The mapping table row addresses can then be used in the auxiliary structures to indirectly refer to the primary B$^+$-tree rows. The two key benefits are: 1) the mapping table shields the auxiliary structures from the volatility of the primary B$^+$-tree row addresses, and 2) the method allows reuse of existing conventional table mechanisms for supporting auxiliary structures on primary B$^+$-trees.

This paper presents the mapping mechanism, its possible application in supporting various auxiliary structures on primary B$^+$-trees, and a case study describing its use for supporting bitmap indexes on index-organized tables in Oracle9*i*. The case study demonstrates that the proposed mapping mechanism allows us to reuse existing bitmap index technologies with minimal changes. It also includes a comparison between bitmap and non-bitmap (B$^+$-tree) index performance on index-organized tables for both single-table queries and star queries. The analytical and experimental studies show that the method is storage efficient, and (despite the mapping table overhead) provides performance benefits that are similar to those provided by bitmap indexes implemented on conventional tables.

1 Introduction

Primary B$^+$-tree [Helm94] like structures (e.g., *tables with clustered index* in Sybase [SYB95] and Microsoft SQL Server [MS98], *index-organized tables* in Oracle [SDFC+00], *key-sequenced tables* in Compaq Non-Stop SQL [Tand87]) are increasingly being used in emerging domains [SDFC+00] such as internet (e.g., search engines and portals), e-commerce (e.g., online store), and data warehousing. These structures provide fast primary key based access for both exact and range searches and are storage efficient at the same time.

Although these structures provide fast primary key based access, other access methods may still be needed depending on application workloads. For example, if the workload contains queries that involve predicates on non-primary key columns, creating a B$^+$-tree index [Com79] on those columns may be useful. However, if the target columns have a large number of duplicates then creating a bitmap index [OQ97] may be more appropriate. Similarly, if the workload contains queries involving aggregates, then creating a materialized view [SI84] that pre-computes the frequently needed aggregates would be useful. Also, for queries involving user-defined predicates, there may be a need to create indexes that are supported via extensible indexing mechanisms (such as [SMSAD00]). In general, depending on workload characteristics, auxiliary structures such as B$^+$-tree indexes, bitmap indexes, user-defined indexes, and materialized views may be required for tables stored as primary B$^+$-trees. (In the rest of the paper, for brevity, we use 'primary B$^+$-tree' instead of 'table stored as primary B$^+$-tree'.)

Auxiliary structures built on conventional tables typically rely on the properties of physical row identifiers, namely, known format and relative stability. For example, a bitmap index, frequently used in data warehousing applications, provides compact storage as well as efficient AND and OR operations by building bitmaps that are based on the physical row addresses [OQ97, CI98].

Although such a physical address based mechanism works well for conventional tables where physical addresses of table rows rarely change, it is not well suited for primary B$^+$-trees where physical addresses of table rows change because table rows must move to maintain the sorted order. For example, consider a DML operation on a primary B$^+$-tree that does not affect any existing bitmap index key values. Such an operation can still cause physical addresses of multiple table rows to change (due to movement of rows inside a block or due to block splits) requiring expensive bitmap index maintenance.

The other option of using logical (primary key-based) row identifiers directly in place of physical addresses of table rows may not be always feasible since logical row identifiers do not have the properties of physical row identifiers. For example, constructing a bitmap from a set of logical row identifiers would require significant changes to existing bitmap index mechanism owing to the continuous distribution of primary key values.

The focus of this paper is on providing a mechanism to support auxiliary structures such as bitmap indexes on primary B$^+$-trees in an environment where primary B$^+$-trees are used in place of conventional tables for better performance and storage characteristics [SDFC+00]. To support bitmap indexes or other auxiliary structures on primary B$^+$-tree (referred to as *base table*), we propose:

- creating a set of physical addresses that uniquely identify the primary B^+-tree rows and at the same time do not change when the base table rows move, and
- a *mapping mechanism* that provides efficient bi-directional (one-to-one) mapping between the set of logical row identifiers for the primary B^+-tree rows and the set of physical addresses introduced above.

This is achieved in Oracle9*i* as follows:

- A new single-column conventional table (referred to as *mapping table*) is introduced with the same number of rows as the base table. This in effect creates a new set of physical addresses for the base table.
- Each mapping table row holds the logical row identifier for a primary B^+-tree row. This provides a mapping from the physical address of a mapping table row to the logical row identifier for the corresponding primary B^+-tree row.
- The primary B^+-tree structure is augmented to include in each row the physical address of the corresponding mapping table row. This provides an efficient reverse mapping, namely, from the logical row identifier to the physical addresses of the corresponding mapping table row.

The mapping table row identifiers can then be used in bitmap indexes or other auxiliary structures. Maintenance requirements for the auxiliary structures will be exactly the same as those in the case of conventional tables. Specifically, row movements in the underlying primary B^+-tree structure do not affect the correctness of either the auxiliary structures or the mapping table. Even in the rare case of primary key update, only the mapping table row needs to be updated because such an update does not cause the mapping table row to move thereby avoiding any auxiliary structure maintenance. Thus the mapping table minimizes auxiliary structure maintenance while permitting free row movement in the underlying primary B^+-tree structure. Furthermore, this allows the existing auxiliary structure mechanisms to be used without any changes.

As far as performance of primary B^+-tree query is concerned, the mapping table may introduce a level of indirection. However, for queries accessing auxiliary structure only or accessing mapping table via auxiliary structure, there is no performance degradation when compared to similar queries on conventional tables (for details see Section 6.1). It does incur an extra level of indirection for any query plan that uses auxiliary structure based access to the primary B^+-tree (see Figure 1). For example, in a bitmap index based primary B^+-tree access, the physical row identifier returned by bitmap is used to directly access the corresponding mapping table row. The logical row identifier contained in the mapping table row is then used, usually with a single read by using the guess-DBA hint [CDFS+01, SDFC+00] stored in the logical row identifier (for details see Section 2), to access the target primary B^+-tree row. Although the mapping mechanism has the potential overhead of one additional I/O for each primary B^+-tree row that needs to be accessed via the mapping table, the compact nature of the mapping table may amortize the actual overhead over multiple row fetches. Furthermore, as shown in the case study presented in Section 7, use of bitmap indexes, enabled by the mapping mechanism, results in significant performance improvement for queries accessing the primary B^+-tree rows when compared to the alternative of using B^+-tree index.

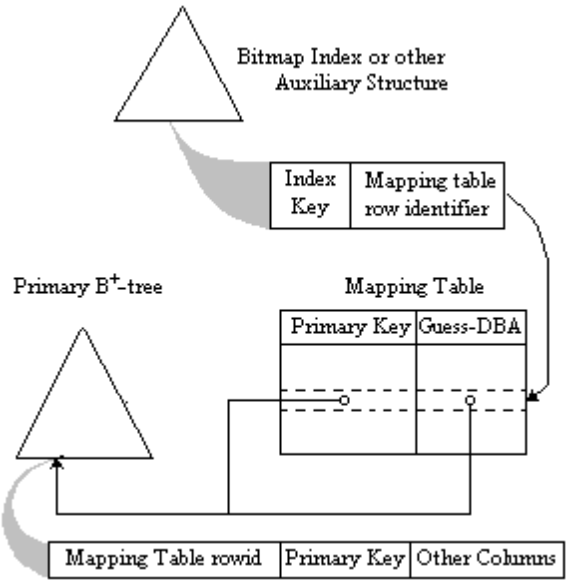

Figure 1: The primary B^+-tree Structure with Mapping Table

Using this approach, we have implemented bitmap index support on index-organized tables in Oracle9*i*. The analysis and experiments in Sections 6 and 7 show that despite the mapping table overhead, this approach provides performance benefits that are similar to those provided by bitmap indexes on conventional tables.

The contributions of this paper are:

- a mapping mechanism that provides the desirable properties of physical row identifiers while still using primary key-based logical row identifiers. Although the paper focuses on use of the mapping table for building bitmap indexes, such an intermediate structure is applicable for other scenarios where physical row identifier properties are desired (possible uses of the mapping table mechanism are discussed in Section 3).
- enabling technology for bitmap index support for primary B^+-tree like structures. To the best of our knowledge, our paper is the first to address bitmap index support on primary B^+-tree like structures. This greatly increases the applicability of primary B^+-tree structures to data warehousing environments because we can efficiently execute star queries involving primary B^+-trees by using bitmap indexes.

Section 2 describes, in detail, the mapping mechanism. Section 3 discusses various application areas of the mapping mechanism. Section 4 presents changes needed for the bitmap index support in Oracle9*i*. Section 5 gives an overview of bitmap index performance using the mapping mechanism. Section 6 compares bitmap index performance on index-organized tables with that on conventional tables. Section 7 shows the performance comparison between a bitmap index and a B^+-tree index on index-organized tables to validate the usefulness of bitmap index on primary B^+-tree structures. Our conclusions are given in Section 8.

2 Key Concepts

In this section, we describe the mapping mechanism that provides such properties and at the same time retains the logical nature of the primary B^+-tree structures.

2.1Mapping Table

We build a conventional table, referred to as *mapping table*, that contains one row for each row in the primary B^+-tree like structure. Specifically,

- The mapping table row contains the primary key values of the base table. The mapping table provides a 1-1 mapping between the primary keys of the primary B^+-tree like structure and the physical row identifiers of the mapping table. Given n rows of the primary B^+-tree structure, their primary keys $\{K_1, K_2, ..., K_n\}$, and their physical row identifiers in the mapping table, $\{p_1,p_2,\ldots,p_n\}$, the mapping table provides the mapping, m: $p_i = m(K_i)$ and $K_i = m^{-1}(p_i)$ for $i = 1,\ldots,n$. The important aspects of the mapping table are:
  1) Even if a row moves in the primary B^+-tree, the corresponding row in the mapping table does not move.
  2) The row identifiers of the mapping table have the physical row identifier properties.
  3) If a primary key value in the primary B^+-tree structure changes, the corresponding mapping table row is updated in-place. This ensures that the physical row identifier of the mapping table row does not change.
- The mapping table also contains, along with the primary key, the disk block address of the leaf block where the primary B^+-tree row is *likely* to be found. This disk block address, referred to as guess-DBA [SDFC+00], facilitates direct lookup from the mapping table to the primary B^+-tree row. If the primary B^+-tree row moves due to a leaf block split, the guess-DBA stored in the corresponding mapping table row is not updated immediately. If the guess-DBA is incorrect, accessing the primary B^+-tree row from the mapping table row requires a primary key-based traversal. The incorrect guess-DBAs (if any) can be fixed online by a background process.
- One mapping table is shared by all bitmap indexes or other auxiliary structures on the primary B^+-tree amortizing the storage overhead across all auxiliary structures.

2.2 Augmented Primary B^+-tree

In conventional tables, physical row identifiers stored in auxiliary structures provide direct access to the base table, while the base table row identifiers in conjunction with key values allow DML operations to efficiently access the auxiliary structures. The mapping table provides the same mechanism efficiently:

- The mapping table row identifiers stored in auxiliary structures are used to directly access the mapping table. The logical row identifiers stored in the mapping table are then used to access the primary B^+-tree.
- The primary B^+-tree is augmented to include the physical row identifiers of the mapping table rows. Each row includes the corresponding mapping table row identifier. The translation from the primary key to the mapping table row identifier is efficiently done by primary key lookup on the primary B^+-tree structure, which identifies the row identifier for the corresponding mapping table row. Note that the primary B^+-tree behaves like a primary key index on the mapping table. These row identifiers are used during DML operations to directly access the auxiliary structures.

2.3 Mapping Table Creation and Maintenance

The mapping table of a primary B^+-tree is created along with the base table and is maintained through all the DML operations. During an initial bulk-load of the primary B^+-tree, for each row, the corresponding mapping table entry is inserted and its physical address is used to create the primary B^+-tree row. By ensuring that the mapping table rows are well clustered with respect to the primary B^+-tree index rows, a bitmap index-based (or auxiliary structure-based) scan can minimize random I/Os incurred during the primary B^+-tree row access. Specifically, the bitmap index operation returns the mapping table row identifiers in physical address order, ensuring that mapping table blocks are never revisited. Similarly, due to the clustering characteristic of the mapping table, primary B^+-tree blocks are not revisited either.

For incremental insert operations, a mapping table row is created and inserted into the mapping table and its physical address is stored as part of primary B^+-tree row. Similarly for delete operations the mapping table row is deleted using the physical row identifier obtained from the primary B^+-tree row. For update operations on the primary B^+-tree, mapping table maintenance is required only if the primary key columns are updated. The corresponding mapping table row is accessed using the physical mapping table row identifier stored in the primary B^+-tree row. An in-place update is performed and if required (when mapping table row expands) a forward pointer is created to retain the same physical row identifier. Note that the primary key column updates are rare and hence forward row chaining is not a major performance issue.

3 Applications of the Mapping Mechanism

3.1 Supporting Bitmap Indexes on Primary B^+-trees

Bitmap indexes are useful for indexing columns with a small number of distinct values. They are widely used in data warehousing applications [CD97] especially for star queries. They are space-efficient and can efficiently filter out rows by performing operations directly on bitmaps. A bitmap represents a 1-1 mapping from a table row to its corresponding bit position, and is based on the properties of physical row identifiers, namely, known format/distribution, and relative stability. Therefore, given a bit position, the location of its corresponding table row can be computed by simple arithmetic operations. Since the physical row identifiers of primary B^+-tree structures are volatile, we propose the following mechanism for supporting bitmap indexes on these structures:

- Use the physical row identifiers of the mapping table rows corresponding to the primary B^+-tree structure rows to construct bitmaps. This will enable existing bitmap index techniques to be applied without change.
- When a bitmap index column is updated, use the physical row identifier of the mapping table row, stored in the primary B^+-tree structure, to perform corresponding update directly on bitmap index. The mapping table need not be accessed for the update unless the primary key is updated which is typically rare.

3.2 Supporting Secondary B$^+$-tree Indexes on Primary B$^+$-trees with Large Keys

Due to frequent row movements of the primary B$^+$-tree, secondary indexes on the primary B$^+$-tree usually contain the primary key instead of physical row identifiers of the primary B$^+$-tree [CDFS+01]. While this approach has lesser index maintenance overhead, the storage overhead for large sized primary keys can be significant as secondary indexes may occupy more space than the primary B$^+$-tree. The mapping table mechanism can be used to resolve this problem by storing all the primary keys in one mapping table, and using the physical row identifiers of the mapping table rows in all the secondary indexes [CDFS+01]. This approach has several advantages:

- Significant storage savings can be achieved as all the secondary indexes now contain physical row identifiers of the mapping table rows, not large primary keys. The large primary keys are stored in the mapping table, and shared among all the secondary indexes.
- Additionally, by storing a guess-DBA per index row, such indexes can have index-based scan performance comparable to that of their counterparts on conventional tables [SDFC+00, CDFS+01]. Index-based scans can verify the existence of the row in the primary B$^+$-tree block identified by the guess-DBA by comparing the mapping table row identifier with mapping table row identifiers stored in that block. If no match is found, implying the guess-DBA is stale, then it can fall back to mapping table based access to primary B$^+$-tree.
- Index-only scan would be faster when no primary keys are fetched since fewer index blocks need to be accessed.

A drawback of this mechanism is the extra level of indirection to fetch a primary key from the mapping table when the guess-DBA is obsolete. This can be alleviated by fixing stale guess-DBAs using a background process. This approach is attractive when there are multiple secondary indexes and storage efficiency becomes a motivating factor.

3.3 Supporting User-defined Indexes on Primary B$^+$-trees

Most database systems support extensibility mechanisms (such as [SMSAD00]) that allows integration of non-native indexing schemes into the server. Ideally, such user-defined indexing schemes should work transparently against both conventional tables and primary B$^+$-trees. If these indexing schemes rely on the properties of physical row addresses, then they cannot be reused for primary B$^+$-trees.

For such cases, the proposed mapping mechanism can be used. Namely, these indexing schemes can use the mapping table row addresses to refer to the rows of the primary B$^+$-trees. This will allow reuse of the user-defined indexing schemes developed for conventional tables. The extra level of indirection resulting from mapping table would increase the performance overhead by a factor proportional to final result set size. Thus, using a user-defined scheme via the mapping table will exhibit performance gains when the user-defined predicate is selective.

3.4 Supporting Materialized Views on Primary B$^+$-trees

Materialized views defined on conventional tables can be incrementally refreshed. This is accomplished by keeping track of either primary keys or physical addresses of rows that have changed since the last refresh operation. Physical row address based incremental refresh mechanism is more commonly used as it can capture the incremental bulk load sessions simply via physical row address ranges. Also, such a scheme is more efficient compared to a scheme that enumerates the row addresses as it avoids random I/Os to get the content of changed rows.

For tables stored as primary B$^+$-trees, materialized views with primary key-based incremental refresh mechanism must be used. However, in conjunction with the proposed mapping mechanism, one can create materialized views on tables stored as primary B$^+$-trees that support the incremental refresh by using the physical addresses of mapping table rows. Since the mapping table is a conventional table, we can reuse the existing physical row address based incremental refresh mechanism. Note that the mapping table is used only for materialized view maintenance and it does not affect materialized view query performance.

4 Bitmap Indexes on Index-Organized Tables in Oracle9*i*

The proposed mapping mechanism is used in Oracle9*i* to support bitmap indexes on index-organized tables. In addition to mapping table changes as outlined in Section 2, bitmap index support requires changes described below.

4.1 Creation and Maintenance of Bitmap Indexes

Creation of bitmap indexes on primary B$^+$-trees is similar to that for conventional tables. As explained earlier the mapping table row identifiers are used for creating bitmap indexes. However, since the mapping table row identifiers are stored as part of each row in the primary B$^+$-tree, bitmap index creation does *not* require accessing the mapping table. For all DML operations, bitmap index maintenance on primary B$^+$-trees is similar to corresponding operations on conventional tables.

4.2 Query Processing

In conventional tables with bitmap indexes there are two kinds of query plans, namely, bitmap index-only scan and bitmap index-based scan where the base table is accessed after the bitmap operation. However, a primary B$^+$-tree with bitmap indexes which uses mapping mechanism would have three query plans (See Figure 2), namely,

- *bitmap index-only scan*: When a query references bitmap index columns only, bitmap index scan is sufficient.
- *bitmap index-based mapping table scan*: When a query references bitmap index columns and primary key columns only, bitmap index scan followed by the mapping table scan is sufficient.
- *bitmap index-based base table scan* (via mapping table): When a query references bitmap index columns and columns that are neither part of primary key nor part of bitmap index columns, the base table is accessed via the mapping table.

Queries on conventional table that reference primary key columns and the bitmap index columns are executed using the bitmap index-based scan. However, the same query when executed against primary B$^+$-tree will be processed using a bitmap index-based mapping table scan. Since this eliminates the base table access, and the mapping table is smaller (it holds only the logical row identifier as opposed to the entire row) than the conventional table, faster query processing times are achieved.

The cost-based optimizer has been augmented to account for mapping table access cost. This is done by utilizing the same cost model as for the conventional table scan. The difference is that the mapping table statistics replace the base table statistics when the scan cost is computed. The cost of the base table scan using the mapping table depends on the quality of the guess-DBAs. The optimizer maintains a statistic called the *guess quality*, a ratio of correct guess-DBAs to the total number of rows, for the mapping table, and uses it in the generation of query plans. The guess quality is implicitly collected for the mapping table. Given the guess quality *P*, the number of rows fetched *N*, and the height of the tree *L*, the cost formula for accessing the primary B^+-tree from the mapping table is:

$$P * N + (1 - P) * N * (L + 1) = N + N * L * (1 - P)$$

The first term represents the access cost when guess-DBAs are correct. The second term represents the access cost when guess-DBAs are incorrect (i.e., first trying the guess-DBA and then falling back to the primary key traversal).

If this cost is smaller than the primary key based traversal, the guess-DBA is used first to access the primary B^+-tree. Otherwise, the primary key based traversal is directly used for the base table lookup without using the guess-DBAs.

5 Bitmap Index Performance Evaluation: An Overview

Performance of bitmap indexes on primary B^+-trees differs from that on conventional tables [JL99] due to the presence of a mapping table. The analyses and experiments conducted fall in three categories described below:

5.1 Bitmap Index Performance on Primary B^+-tree vs. Conventional Table

These experiments focus on characterizing the performance degradation due to a level of indirection caused by the presence of mapping table. As shown in Section 6.1, two out of three possible primary B^+-tree query plans involving bitmap indexes, namely, bitmap index-only scan and bitmap index-based mapping table scan, suffer no performance degradation when compared to conventional tables.

For the queries requiring base table access, performance depends on the result-set size. We expect such queries on primary B^+-tree using bitmap indexes to potentially incur one additional I/O per row in the result set. However, the overall performance degradation depends on factors such as bitmap operation cost and mapping table access cost, which are discussed in Section 6.1. Section 6.2 discusses the effect of two of these factors, namely, cost of bitmap operations vs. cost of table row access and the effect of caching the mapping table. The experiment uses a constant result-set size thereby fixing the mapping table overhead. The performance degradation of a query on a primary B^+-tree with respect to a similar query on a conventional table decreases as the bitmap index processing cost increases. Namely, the fixed mapping table overhead becomes smaller part of overall query processing cost. Also, in this particular experiment when mapping table rows were cached the performance degradation is further reduced. In the case of star queries, the overhead from additional processing such as joins makes the overall cost larger thereby reducing the impact of the mapping table access overhead (see Section 6.3 for more details).

5.2 Bitmap Index vs. B^+-tree Index Performance on Primary B^+-trees

These experiments focus on characterizing the performance against best alternative in the absence of bitmap indexes, namely, logical row identifier based B^+-tree indexes that use guess-DBAs [CDFS+01]. The other alternative, namely B^+-tree index built with mapping table row identifier was not compared as it would incur additional overhead when compared to a logical row identifier based B^+-tree index.

Queries involving B^+-tree index typically choose the most selective index and filters the rest of the predicates after accessing the base table rows. The same query with bitmap index configuration uses all applicable bitmap indexes to filter the rows by performing bitmap operations and then accesses the base table for the rows in the result set. When the queries involve conjunctive predicates, the result set size is expected to be smaller than the number of rows returned by the most selective index. This behavior is analyzed in Section 7.1 and validated through experiment in Section 7.2. For conjunctive predicates, as expected, the bitmap index configuration outperforms the corresponding B^+-tree index configuration.

The same benefit is expected even for star queries which usually involve a combination of conjunctive and disjunctive predicates. Section 7.3 describes the experiment which compares the performance of star queries returning varying number of rows for bitmap and B^+-tree index configuration. Since the result set size is smaller than the rows returned by most selective B^+-tree index, the bitmap index configuration outperforms the B^+-tree index configuration.

The two classes of experiments describe the bitmap index performance for a primary B^+-tree populated via bulk-load. Below we discuss the performance characteristics when more data is added via DML or batch append operations using loader utility.

5.3 Bitmap Index Performance on Primary B^+-tree after DML Operations or Batch Appends

The impact of bulk-load vs. incremental load on bitmap index performance can be categorized into two classes:

- For queries resulting in bitmap index-only scan or bitmap index-based mapping table scan, there is no performance degradation as the base table is not accessed.
- For queries resulting in bitmap index-based base table scan, performance can potentially degrade as mapping table rows returned 1) may not be in primary key order and 2) may contain invalid guess-DBAs. The first factor is typically not the issue since the result set is usually much smaller than total number of blocks occupied by the base table. This is the most common case, that is, we expect that bitmap operation processing typically would return small fraction of rows from the base table. For such cases, performance degradation is mainly caused by the presence of invalid guess-DBAs in mapping table rows. However, despite this degradation, the query performance utilizing bitmap indexes is better than that utilizing B^+-tree indexes (See Section 7.4 for performance results). For cases, where result set is a significant fraction of (or larger than) the number of blocks occupied by the base table, the clustering factor may become important. For such cases, the performance degradation can be minimized by loading additional batches of data after sorting as this will ensure partial ordering, if not global ordering of mapping table rows in primary key order. Also,

the need for subsequent updates can be avoided by using a partitioned table where new batches of data are added as separate partitions. In general, we expect that bitmap index-based scan typically would return small result sets.

6 Bitmap Index Performance Evaluation: Index-Organized Table vs. Conventional Table

In this section, we compare performance of queries using bitmap indexes on index-organized tables with that on conventional tables.

6.1 Analysis

The three query plans explained in Section 4.2 are shown in Figure 2. The figure also includes, for comparison, corresponding query plans that will be used for conventional tables.

- *Bitmap index-only scan* (1(a) and 1(b) in Figure 2): A bitmap index on an index-organized table is built using mapping table row addresses whereas a bitmap index on a conventional table is built using base table row addresses. Since, unlike conventional table rows, mapping table rows hold only primary column values, the mapping table is smaller in size than the base table. This leads to fewer bitmap index entries since each bitmap now needs to cover a smaller range of blocks. Therefore, query performance using bitmap index-only scan on index-organized tables will be comparable or better than that on conventional tables.
- *Bitmap index-based mapping table scan* (2(a) and 2(b) in Figure 2): As explained in Section 4.2, for index-organized tables, queries that refer only to bitmap index columns and primary key columns can be processed without performing a base table scan. However, since processing similar queries for conventional tables would require bitmap index-based table scan, performance for such queries for index-organized tables will be comparable or better than that for conventional tables.

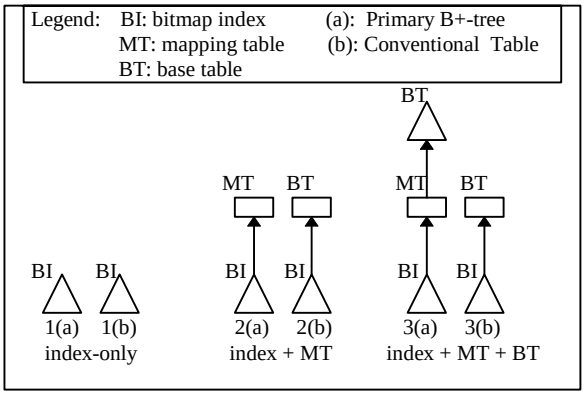

Figure 2: Query Plans (Primary B^+-tree vs. Conventional Table)

- *Bitmap index-based base table scan* (3(a) and 3(b) in Figure 2): For this query plan, performance for conventional tables will be better than that for index-organized tables because one additional structure (namely, the mapping table) needs to be scanned for an index-organized table. A detailed analysis follows.

Performance for Bitmap Index-based Base Table Scan

Using symbol definitions in the adjoining table, query costs for the two cases can be expressed as: $(B + R_{heap})$ and $(B + R_{map} + R_{iot})$, respectively. Thus, the performance degradation, as a fraction of the cost in the conventional table case, may be expressed as:

$$\rho = [(B + R_{map} + R_{iot}) - (B + R_{heap})] / (B + R_{heap})$$
$$= (R_{map} + R_{iot} - R_{heap}) / (B + R_{heap})$$
$$\approx R_{map} / (B + R_{heap}),$$ assuming $R_{heap} = R_{iot}$, which is true if all the guess-DBAs corresponding to result set are valid and base table row clustering for the rows in the result set is similar in both cases. Note that the second factor is less significant as discussed earlier in Section 5.

Table 1: Parameters used for Query Costs

Symbol	Definition
B	Cost for bitmap operations to determine the result-set
R_{heap}	Cost for accessing rows in the result-set from conventional table (stored as heap)
R_{map}	Cost for accessing rows in the result-set from mapping table
R_{iot}	Cost for accessing rows in the result-set from index-organized table

The difference in bitmap index-based base table scan cost for index-organized table and that for conventional table is influenced by one or more of the following factors:

Bitmap Operations Cost vs. Table Row Access Cost: As the relative cost of bitmap operations (B) increases with respect to the table row access costs (R_{map}, R_{iot}, and R_{heap}), value of ρ decreases. This is demonstrated in the experiments we conducted for single-table (Section 6.2) and star queries (Section 6.3). In star queries, in particular, bitmap operations cost can often be significant as is seen in the star query experiment.

Effect of Large Non-key Columns: If the size of the non-key columns is much larger compared to that of the key columns, then the size of the mapping table is accordingly much smaller than that of the index-organized table or conventional table. This results in R_{map} being much smaller than R_{heap}, thereby reducing the difference in performance.

Effect of Caching Mapping Table: Since the mapping table is crucial for bitmap index performance, caching it would yield significant performance gains. This is also made possible by the fact that mapping table can often be much smaller than the base table. Caching mapping table blocks can reduce the value of R_{map}, thereby reducing the value of ρ, as shown in Section 6.2.

6.2 Single-Table Query Performance

Although several factors (as identified in the analysis above) may affect single-table query performance, due to space constraints we show the effect of only two of the factors namely, bitmap operations cost vs. table row access cost and the effect of caching the mapping table for the case when bitmap index-based base table scan is employed. The experiments described in this section as well as in next section are performed on a Sun workstation Ultra 60 with 2 CPUs having 512MB memory using Oracle9*i*.

Experiment 1: Single-Table Query

The goal of this experiment was to measure the value of ρ for different costs of bitmap operations, B, while keeping the cost of table row access unchanged. To achieve this, we varied the total

number of bitmaps selected by the query while keeping the size of the result-set constant. The results (Figure 3) showed that, as the cost of B increased due to higher number of bitmaps selected, the performance difference gradually went down from 93% to 67%. Next, to see the effect of caching of mapping table, we carried out the same measurements but this time with the mapping table cached. The new results showed a significant reduction in performance difference (ranges from 29% to 18%).

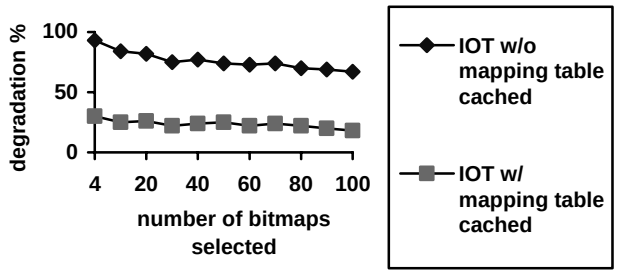

Figure 3: Performance Degradation of Index-Organized Table (w & w/o mapping table cached) w.r.t. Conventional Table

6.3 Star Query Performance

To study performance implications of this mapping mechanism in a data warehousing environment, we conducted experiments on a star schema (see Figure 4). Specifically, a Sales fact table containing 1 million rows was used for experimentation. The fact table is implemented as an index-organized table. Bitmap indexes are built on the fact table columns that correspond to keys for the dimension tables.

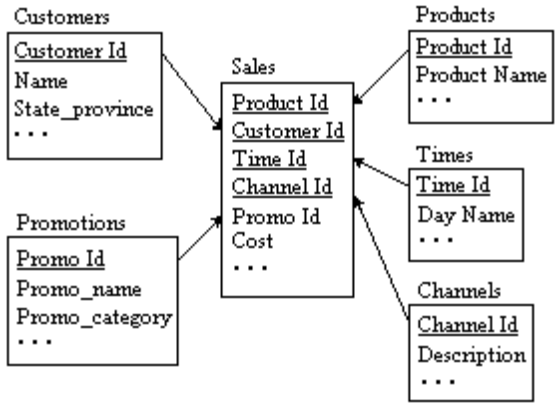

Figure 4: Star Schema Used in the Experiment

Since, based on the analysis above, the difference in performance between the two configurations is: $(B + R_{map} + R_{iot}) - (B + R_{heap}) = (R_{map} + R_{iot} - R_{heap}) \approx R_{map}$, assuming $R_{heap} = R_{iot}$, we measured the query costs for different values of the result-set size after bitmap index operations. It may be noted that R_{map} depends upon the result-set size. As the result-set size increases, while other factors remain constant, cost for star queries using bitmap indexes on an index-organized table would increase more than the corresponding increase in cost for a conventional table.

Experiment 2: Star Query

To study the effect of result-set size on the performance degradation factor (ρ), we chose four star queries (joining Promotions, Times, and Customer with Sales) which returned 8, 648, 1867, and 6012 rows respectively. The performance degradation ranges from 2.7% to 24.9% as shown in Figure 5. That is, as the size of the result-set increases, performance degrades further. It is difficult to measure the upper limit of performance degradation because as the number of rows fetched increases, the optimizer is likely to choose a different access path.

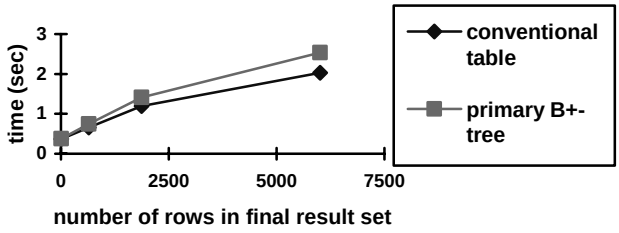

Figure 5: Star Query Performance: Primary B$^+$-tree vs. Conventional Table

7 Index-Organized Table Performance Evaluation: Bitmap vs. B$^+$-tree Index

This section presents a comparison of storage costs and query performance for the following two configurations:

- a bitmap index configuration on an index-organized table that includes a mapping table M and bitmap indexes BM_i, i=1..X, and
- a B$^+$-tree index configuration on an index-organized table that includes B$^+$-tree indexes BT_i, i=1..X.

7.1 Analysis

The cost of retrieving rows from the base table, and in the case of bitmap index configuration, also from the mapping table, is a significant (and often the dominant) portion of the overall index-based base table scan cost because each target row may reside in a different disk block. In this section we do a comparative analysis of this cost for the two configurations. The analysis is done for two types of queries: queries with conjunctive predicates and those with disjunctive predicates. We also compare the storage costs.

Storage Cost

Table 2: Parameters used for Storage Costs

Symbol	Definition
S_{bmap}	Storage costs for Bitmap index configuration
S_{btree}	Storage costs for B$^+$-tree index configuration
K_r	index key size of r^{th} row of the table
L_r	Logical row identifier size of r^{th} row of the table

Storage costs for the two configurations (excluding the index-organized table storage), can be expressed as

$$S_{bmap} = Size(M) + \sum_i Size(BM_i)$$
$$= \sum_r Size(L_r) + \sum_i Size(BM_i).$$
$$S_{btree} = \sum_i Size(BT_i)$$
$$= \sum_i (Size(Branch\ Nodes) + \sum_r (Size(L_r) + Size(K_r)))$$

Consider the two cases:

- *A single index (X=1)*: if $Size(BM_1) < Size(Branch\ Nodes) + \sum_r Size(K_r)$ then $S_{bmap} < S_{btree}$. This is usually true for indexes on low cardinality columns because bitmap index minimizes duplicate storage of keys and uses compressed representation of sets of physical row identifiers as bitmaps.

- *More than one indexes (X>1)*: the storage overhead for logical row identifiers in each additional B^+-tree index makes $S_{bmap} << S_{btree}$.

Intuitively, one can view each B^+-tree index on an index-organized table as having its own private mapping table embedded in the index structure, whereas a single common mapping table is shared for all bitmap indexes. This factor makes storage costs for bitmap configuration significantly lower when there are two or more indexes. For the case when there is only a single index present, the two storage costs are comparable but even there it can be lower for bitmap index especially for low cardinality columns.

Performance of Queries with Conjunctive Predicates

Table 3: Parameters used for Query Performance

Symbol	Definition
QC_{bmap}	conjunctive predicate query cost for bitmap indexes
QC_{btree}	conjunctive predicate query cost for B^+-tree indexes
N_i	number of rows selected using predicate P_i
N_k	number of rows selected using most selective predicate P_k
N	number of rows in the final result-set
f_c	= N/ N_k, range of its possible values is [0, 1]

We consider a query with WHERE clause of the form: "WHERE P_1 AND P_2 AND … AND P_x". Here each P_i, for i = 1 to X, is a predicate that can be evaluated using the i^{th} (bitmap or B^+-tree) index.

For the bitmap index configuration, query execution involves evaluating each predicate using the corresponding bitmap index and then computing their intersection to arrive at the final result-set consisting of N mapping table row identifiers. Next, it retrieves each such mapping table row and then using the guess-DBA found in the mapping table row, corresponding base table row is accessed. Assuming that the guess-DBAs are correct, the total number of block accesses, for the mapping table and the base table, can be expressed as follows:

$$QC_{bmap} = 2 * N$$

For the B^+-tree index configuration, query execution involves scanning the most selective B^+-tree index to select N_k candidate rows. These rows are then accessed in the base table and other predicates are evaluated on those rows to determine the final result set. Thus, assuming that the guess-DBAs are correct, the total number of base table block accesses can be expressed as follows:

$$QC_{btree} = N_k = N * (1/ f_c)$$

To compare the two costs, we compute the difference in number of block accesses,

$$\begin{aligned}\delta &= QC_{btree} - QC_{bmap} \\ &= N * (1/f_c) - 2 * N \\ &= N * (1/f_c - 2)\end{aligned}$$

So, $\delta > 0$ when f_c is in range [0, 0.5). Thus, table block access cost will be lower for the bitmap index configuration when final result-set cardinality (N) is less than half of the cardinality of the set of rows returned by most selective predicate (N_k from the predicate P_k).

Performance of Queries with Disjunctive Predicates

Table 4: Parameters used for Disjunctive Query Cost

Symbol	Definition
QD_{bmap}	disjunctive predicate query cost for bitmap indexes
QD_{btree}	disjunctive predicate query cost for B^+-tree indexes
N_i	number of rows selected using predicate P_i
N	number of rows in the final result-set
f_d	$N/ \sum_i N_i$, range of its possible values is [1/X, 1]

Consider a query with the WHERE clause of the form: "P_1 OR P_2 OR … OR P_X" Here each P_i, for i = 1 to X, is a predicate that can be evaluated using the i^{th} (bitmap or B^+-tree) index.

For the bitmap index configuration, query execution involves evaluating, for i=1 to X, predicate P_i on bitmap index BM_i, and then computing their union. Next, for each row address in the union, a corresponding mapping table row is accessed. Finally, using the guess-DBA found in the mapping table row, the corresponding base table row is accessed. Thus, the total number of block accesses for the query can be expressed as follows:

$$QD_{bmap} = 2*N$$

For the B^+-tree index configuration, query execution involves executing each OR condition as a separate query using the corresponding B^+-tree index, retrieving a set of rows from the base table, and finally combining the results of all sub-queries. Thus, the total number of base table block accesses for the query can be expressed as follows:

$$QD_{btree} = \sum_i N_i = N * (1/f_d)$$

To compare the two costs, we compute the difference in number of block accesses,

$$\begin{aligned}\delta &= QD_{btree} - QD_{bmap} \\ &= N * (1/f_d) - 2 * N \\ &= N * (1/f_d - 2)\end{aligned}$$

So, $\delta > 0$ when f_d is actually in the [1/X, 0.5) range (which is possible only when X > 2).

Thus, table block access cost will be lower or comparable for the bitmap index configuration when 1) more than two indexes are used and 2) final result-set cardinality (N) is less than half of the sum of the cardinalities ($\sum_i N_i$) of the sets of rows selected by the individual indexes.

The above analysis is for selection of rows specified in the WHERE clause, hence applies to both single-table and star queries. In the following sections, we compare the actual query performance seen in our experiments.

7.2 Single-Table Query Performance

We created a synthetic database to validate the analytical comparison given in Section 7.1 and also to study the effect of change of number of distinct values for the indexed columns. The synthetic database schema is as follows:

- A base index-organized table IOT1 with four integer columns: *a, b, c,* and *d*. Column *a* is the primary key.
- Single column indexes are created on column *b* and on column *c*: a single mapping table and two bitmap indexes for the bitmap index configuration and two B^+-tree indexes for the B^+-tree index configuration.

Experiment 3: Validation of Analytical Study

The experiments are conducted on a 1 million row primary B^+-tree for the two configurations: 1) with a mapping table and two bitmap indexes, and 2) with two B^+-tree secondary indexes. The columns on which the two indexes are built have around 1000 distinct values, with each distinct value having about 1000 duplicate occurrences.

Storage Cost: For the single index (X=1) case storage costs are: S_{bmap} = 19.36M (Mapping table) + 4.56M (bitmap index) = 23.91 M, whereas S_{btree} = 26.69 M (B^+-tree). For the two index (X=2) case, storage costs are: S_{bmap} = 19.36M (Mapping table) + 4.56M (bitmap index) + 4.57 M (bitmap index) = 28.49M, whereas S_{btree} = 26.69 M (B^+-tree) + 26.69M (B^+-tree) = 53.38M. These results validate our analysis given in Section 7.1.

Conjunctive Query Performance: The following query was used:

```
SELECT SUM(d) FROM IOT1 WHERE b = :b AND c = :c;
```

The predicate on the column *b* returns 1000 rows and that on the column *c* returns 1100 rows. We measured the values of total time taken for executing the query under different values of f_c (achieved by changing the extent of overlap between rows returned by the most selective index, i.e., the index on *b*, and the final result-set). The results (shown in Figure 6) were consistent with our expectation based upon the difference in number of table block accesses, $\delta = N * (1/f_c - 2)$, computed in Section 7.1.

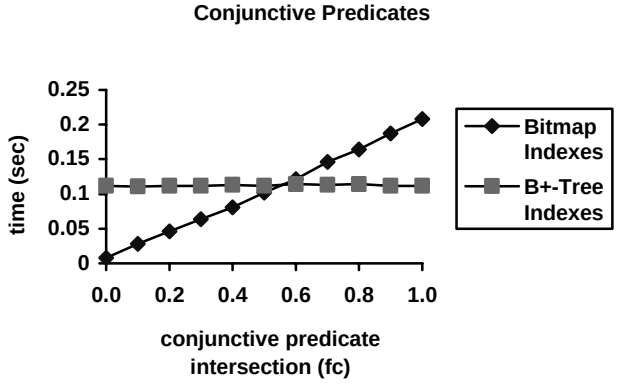

Figure 6: Comparison of Query Performance for Conjunctive Predicates

Disjunctive Query Performance: The following query was used:

```
SELECT SUM(d) FROM IOT1 WHERE b = :b OR c = :c;
```

The predicate on the column *b* returns 1000 rows and that on the column *c* returns 1100 rows. We measured the values of total time taken for executing the query under different values of f_d (achieved by changing the extent of overlap between rows returned by the indexes on *b* and *c* and the final result-set). The results (see Figure 7) were consistent with our expectation based upon the difference in number of table block accesses, $\delta = N * (1/f_d - 2)$, computed in Section 7.1.

It may be noted that, since only two indexes (X=2) were used, bitmap solution was not expected to outperform the B^+-tree solution. However, our analysis in 6.1 showed that for queries involving higher number of indexes, if f_d is in the range [1/X, 0.5) then the bitmap solution will perform better than the B^+-tree solution.

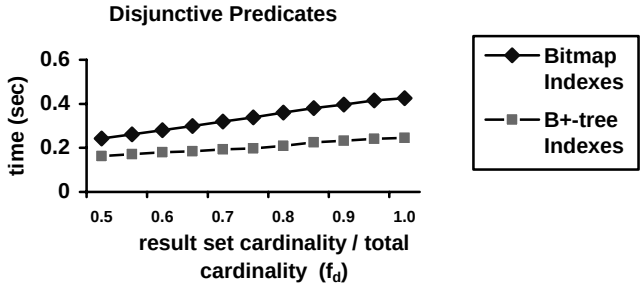

Figure 7: Comparison of Query Performance for Disjunctive Predicates

7.3 Star Query Performance

To study performance implications of bitmap index vs. non-bitmap based star query performance, we conducted experiments on a star schema described in Section 6.3.

Experiment 4: Star Query on Primary B^+-tree (Bitmap vs. B^+-tree Index)

The basic star query processing is shown in Figure 8. Four such star queries with the predicate on Customers table returning varying number of rows (V) is used. The number of rows (W) in the result-sets corresponding to these four queries is given in Table 5.

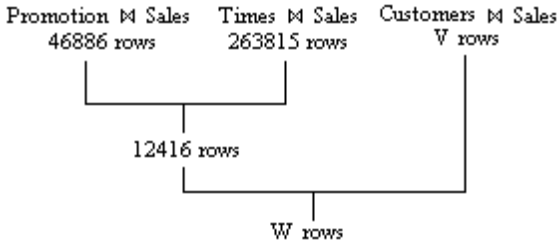

Figure 8: Star Query Predicate Variation

Table 5: Number of Result Rows

Query	V	W	min (V, 46886)
I	18338	240	18338
II	54990	648	46886
III	146713	1867	46886
IV	476936	6012	46886

The experimental results are given in Figure 9. As expected based on the analysis in Section 7.1, for the configuration with bitmap indexes defined on the fact table, query cost changes in a manner that is roughly proportional to the result-set size (W). Each additional row in the result-set could potentially require two

additional data block reads: one from mapping table and one from base table.

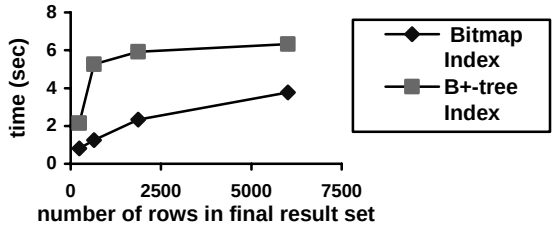

Figure 9: Comparison of Star Query Performance

For the configuration with B^+-tree indexes defined on the fact table, the same query uses the most selective of the available indexes and its cost is dependent on the number of rows returned by this index, that is, min (V, 46886) rows. For Query I, this value is 18338, and for Queries II, III, and IV, it is 46886. This is the dominant factor for query cost in this configuration. While the steep degradation in performance from query I to query II is mainly due to the increase in number of rows returned by the most selective index, the slight and gradual degradation from query II through query IV may be attributed to the added cost of joining the intermediate results with the varying results from the Customer table.

7.4 Query Performance after Bulk-load vs. Incremental Load

This experiment was conducted to study the performance implications for queries accessing primary B^+-tree rows via mapping table, where the primary B^+-tree configuration was arrived at after performing incremental load. This configuration was compared against a fully bulk-loaded primary B^+-tree. This experiment used conjunctive queries.

Experiment 5: Effect of Incremental Load

Initially, 5 million rows are bulk-loaded into the index-organized table and then additional data is loaded in increments of 0.5 million rows. The performance degradation with respect to bulk-loaded configuration ranges from 52 to 56% for incrementally loading 0.5 to 2 million rows. We also measured the performance using the best available alternative, namely using bulk-loaded B^+-tree indexes. Despite the degradation caused by incremental load, the bitmap-index based query outperforms the query using B^+-tree indexes as shown in Figure 10.

The B^+-tree index solution filters on the most selective index and then applies the remaining predicates to the qualifying base table rows, whereas a bitmap index solution utilizes all bitmap indexes to evaluate the predicates before accessing base table rows. Thus, the number of rows accessed in a bitmap index solution is typically much smaller. The benefits of bitmap operations thus outweigh the loss in performance due to incremental load.

8 Conclusions

The paper introduced a mapping mechanism for supporting bitmap indexes and other auxiliary structures on primary B^+-tree. The mechanism involves maintaining a mapping table containing logical row identifiers of the primary B^+-tree rows and using physical addresses of the mapping table rows to build auxiliary structures. By storing the mapping table physical row identifier as part of the primary B^+-tree row, we eliminate the need for a separate indexing structure to map the logical row identifiers to the corresponding physical row identifiers.

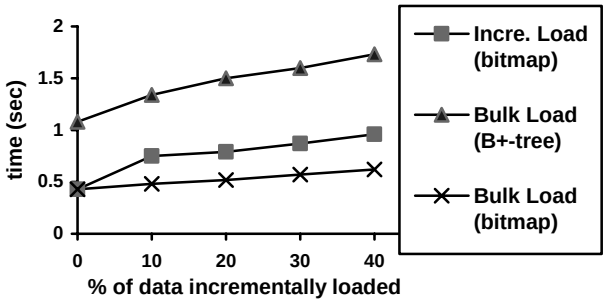

Figure 10: Query Performance after Incremental Load

Using the mapping mechanism, we have implemented bitmap index support on index-organized tables in Oracle9*i*. Two of three possible query plans involving bitmap indexes, namely, bitmap index-only scan and bitmap index-based mapping table scan, suffer no performance degradation when compared to similar queries on conventional tables. The third query plan, namely, bitmap index-based base table scan, incurs additional mapping table overhead. For this class of queries, the overall degradation due to mapping table overhead depends on result set size and the overall query processing cost. We identified the factors contributing to the mapping table overhead and ways to minimize their effect. We also compared storage overhead and query response time for bitmap index vs. non-bitmap (regular) B^+-tree index configurations on index-organized table. For queries with conjunctive predicates on bitmap index columns, our solution outperforms the B^+-tree index based approach for most common cases. Our analysis shows that this solution is storage efficient, and the storage savings are significant when multiple indexes are needed. The paper also discussed use of the mapping table mechanism for supporting other auxiliary structures such as secondary B^+-tree indexes, user-defined indexes, and materialized views.

References

[CI98] Chan, C. Y., Ioannidis, Y.E., "Bitmap Index Design and Evaluation," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp.355-366, 1998.

[CD97] Chaudhuri, S. and Dayal, U., "An Overview of Data Warehousing and OLAP Technology," *ACM SIGMOD Record*, 26(1), pp.65-74, 1997.

[CDFS+01] Chong, E.I., Das, S., Freiwald, C., Srinivasan, J., Yalamanchi, A., Jagannath, M., Tran, A., Krishnan, R., "B^+-Tree Indexes with Hybrid Row Identifiers in Oracle8*i*," *Proceedings of the 17th Int. Conf. on Data Engineering*, pp.341-348, Apr. 2001.

[Com79] Comer, D., "The Ubiquitous B-Tree," *Computing Surveys*, 11(2), pp.121-137, June 1979.

[Helm94] Helman, P., *The Science of Database Management*, Richard D. Irwin, Inc., 1994.

[JL99] Jurgens, M., Lenz, H.J., "Tree Based Indexes vs. Bitmap Indexes - a Performance Study," *Proceedings of the Intl. Workshop on Design and Management of Data Warehouses*, June 1999.

[MS98] Microsoft SQL Server, *SQL Server 7.0 Storage Engine*, White Paper., Oct. 1998.

[OQ97] O'Neil, P. and Quass, D., "Improved Query Performance with Variant Indexes," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 38-49, 1997.

[SI84] Shmueli, O. and Itai, A.: "Maintenance of Views," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 240-255, 1984.

[SDFC+00] Srinivasan, J., Das, S., Freiwald, C., Chong, E.I., Jagannath, M., Yalamanchi, A., Krishnan, R., Tran, A., DeFazio, S., Banerjee, J., "Oracle8*i* Index-Organized Table and its Applications to New Domains," *Proceedings of the 26th Int. Conf. on Very Large Data Bases*, pp. 285-296, Sept. 2000.

[SMSAD00] Srinivasan, J., Murthy, R., Sundara, S., Agarwal, N., DeFazio, S., "Extensible Indexing: A Framework for Integrating Domain-Specific Indexing into Oracle8*i*," *Proceedings of the 16th International Conference on Data Engineering*, pp. 91-100, 2000.

[SYB95] Sybase SQL Server, *Transact-SQL User's Guide*, Document ID:32300-01-1100-02., Dec. 1995.

[Tand87] The Tandem Database Group, NonStop SQL: "A Distributed, High-performance, High-availability Implementation of SQL," *Proceedings of 2nd Int. Workshop on High Performance Transaction Systems*, Springer Lecture Notes in Computer Science No. 359.