Oracle8*i* Index-Organized Table and its Application to New Domains

Jagannathan Srinivasan
Eugene Inseok Chong
Ramkumar Krishnan
Souripriya Das
Mahesh Jagannath
Anh-Tuan Tran
Jayanta Banerjee
Chuck Freiwald
Aravind Yalamanchi
Samuel DeFazio

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

Abstract

Primary B^+-tree, a variant of B^+-tree structure with row data in leaf blocks, is an ideal storage organization for queries involving exact match and/or range search on primary keys. Commercially, primary B^+-tree like structures have been supported in DBMSs like Compaq Non-Stop SQL, Sybase Adaptive Server, and Microsoft SQL Server. Oracle's index-organized table is like a primary B^+-tree; however, it differs from its commercial counterparts in the following respects: 1) The storage organization does not require the entire row to be stored in the primary key index. Infrequently accessed columns can be selectively pushed into an overflow storage area to speed up access to columns that are frequently accessed. 2) Secondary indexes on index-organized tables support logical primary key-based row identifiers, and still provide performance comparable to secondary indexes with physical row identifiers by storing and making use of *guess-DBA* (Database Block Address). 3) Support for primary key compression leads to reduced storage requirements. This paper presents the index-organized table storage option in Oracle8*i* with emphasis on the novel aspects mentioned above. The applicability of index-organized tables to new domains such as the Internet, E-Commerce and Data Warehousing is discussed. A performance study is presented, that validates the clustering benefits of Oracle's primary B^+-tree implementation, and characterizes the impact of overflow storage area, guess-DBA use in secondary B^+-tree indexes, and primary key compression.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

1 Introduction

A significant number of applications deal with data sets where each individual row is identified by a primary key. The primary key could be a single column such as social security number for employees table in a HR application, or a multi-column entity such as <warehouse, district, order, order line> for orders table in a product sales and distribution business application [TPCC93]. For such applications, if the query workload is dominated by primary-key access, then clustering the rows of the table in the primary key order would be beneficial. In fact, several DBMSs provide a variant of B^+-trees [Com79] with row data in leaf node, also referred to as primary B^+-trees, to speed-up primary key-based access to the table data.

Along the same lines, in Oracle8, a new storage option ORGANIZATION INDEX is introduced. Tables created using this option, referred to as *index-organized tables*[1],

[1] The default storage organization in Oracle is a heap, and we refer to tables created using such an organization as *heap-organized tables*.

include not only the indexed columns, but implicitly also include all the remaining columns of the table in the primary B^+-tree. Each row consists of key and non-key columns, and the non-key columns are stored along with the key columns in a B^+-tree, making the whole table structure have an *index-organization*. Typically, the entire table data can be held in its primary key index. The benefits of this organization are:

- it provides fast *random* access on the primary key because an index-only scan is sufficient. Once a leaf block is reached, both the key as well as the non-key columns can be retrieved.
- it provides fast *range* access on the primary key because the rows are clustered in primary key order and they contain both key and non-key columns.
- it avoids duplication of primary key columns as in a heap-organized table with a primary key index.

The distinguishing features of index-organized tables when compared to other primary B^+-tree implementations are:

- support for a (heap-organized) overflow storage area that provides supplementary storage for columns. This allows controlling the placement of columns in the index vs. overflow storage area and provides the capability for tuning the number of rows that fit in an index leaf. Infrequently accessed non-key columns of the index-organized table can be pushed to the overflow storage area, by (1) specifying the percentage of space reserved for a row in the index block, and/or (2) specifying a column at which a row should be divided into index and overflow portions. This increases the *leaf row density*, that is, the number of index rows that can fit in a leaf block of the B^+-tree structure.
- support for secondary indexes with logical primary key-based row identifiers, which include the primary key as well as a database block address (DBA). This DBA, referred to as *guess-DBA*, is treated as a *guess* as to where the row *may* be found in the base table (primary B^+-tree). A valid guess will cost only a single block I/O. However, if the guess is invalid, the primary key is used to find the row. Thus, for valid guess-DBAs, the secondary index performance is comparable to that of secondary index with physical row identifiers. At the same time, the logical nature of secondary indexes enables faster reorganization and increased uptime of the base table since they need not be updated during such a reorganization. Support for online guess-DBA fixing allows regaining the guess-DBA based performance.
- support for compressing common (column) prefixes of the primary key. Since the rows are clustered in the primary key order, there is more likelihood of finding common prefixes.

Index-organized tables in Oracle8*i* have full-table functionality with features such as constraints, triggers, LOB and object columns, and horizontal partitioning. Index-organized tables are key components in several Oracle RDBMS features such as online index creation and rebuild, message queues [OAQ97], nested table columns [OSC97], domain indexes [SMSAD00] and time-series cartridge [OTS97].

Traditionally, primary B+-tree like structures such as index-organized tables have been used in OLTP applications that require fast primary key-based access. However, we argue that such a storage organization, combined with the novel features described above, can be equally useful to several new domains as summarized in Table 1 in Section 3.

Index-organized tables are suitable for order processing applications with 24x7 availability requirements such as for E-Commerce [BSZ98]. Specifically, faster reorganization is achieved due to the logical nature of secondary indexes. Index-based scan performance degradation is avoided through use of guess-DBAs and the guess-DBA based performance is retained by online fixing of any guess-DBAs invalidated during reorganization. Key-compressed index-organized tables are suitable for Internet applications that may require a hierarchical storage organization, such as portals and electronic storefronts. Internet search engines and text databases can implement the inverted index, the fundamental data structure needed for full-text search, as an index-organized table. The need to handle variable length rows in the inverted index [ZMS92] without degrading access to small rows can be met by using index-organized table column placement options. Index-organized tables can also be used for fact tables in data warehousing applications as described in Section 3.5.

1.1 Outline of the Paper

The related work is presented next. Section 2 gives an overview of index-organized table with emphasis on its novel features. Section 3 discusses the applications of index-organized tables to new domains. Section 4 presents experimental results that compare and contrast index-organized table and heap-organized table performance. Section 5 concludes with a summary and outlines future work.

1.2 Related Work

There have been several efforts to build specialized structures to hold table data based upon the primary modes of data access. Oracle RDBMS [OSC97] has traditionally supported *Index-clusters* and *Hash-clusters*. These can be used to cluster one or more tables based on

the cluster key**.** A single data block can hold rows from different tables with the same cluster keys**.** The primary motivation is to support efficient join-operations on the set of clustered tables.

Non-Stop SQL[2] [Tand87] supports *key-sequenced* tables, which cluster data based on primary keys. Online reorganization of these tables is also supported [Troi96]. However, these tables support rows of only limited size and do not provide the flexibility to control the placement of columns between the index and overflow storage areas. Non-Stop SQL also supports secondary indexes with primary key based row identifiers, but these row identifiers are strictly logical and do not incorporate the guess-DBA mechanism found in secondary indexes for index-organized tables. Thus, they always incur an additional primary B-tree traversal for index-based scan.

IBM DB2 [DB2V5.2] allows the secondary index structure to include additional columns. For example, users can create an index for an employee table on a column, say empno, but also include the column salary into the index. Thus, an index-only scan is sufficient for queries referencing both the columns empno and salary, but this speed-up is achieved at the cost of column duplication.

Sybase Adaptive Server [SYB95] and Microsoft SQL Server 7.0 [MS98] support the concept of a table with single clustered index, which forces the table rows to be maintained in the clustered index key order. However, like Non-Stop SQL, they do not support rows of arbitrarily large size, or flexible column placement between the index and overflow**.** Secondary indexes for these tables contain physical row identifiers, limiting the online availability of the base table.

Online reorganization of database objects, namely tables and indexes, is increasingly gaining importance and many database systems support some form of online operations. IBM AS/400 [SI96] allows online index (re)construction. Sybase provides capabilities for online resource management [RDC96]. [ZS98] discusses an approach for secondary index maintenance during online reorganization by piggybacking secondary index updates with user transactions. During online reorganization of index-organized tables, secondary indexes need not be maintained due to their logical nature.

2 Index-Organized Tables

This section gives an overview of index-organized tables and then discusses the novel features.

[2] Oracle8, Oracle8*i*, DB2, NonStop SQL, Microsoft SQL Server, Sybase Adaptive Server are registered trademarks.

2.1 Overview

Index-organized tables are like conventional tables in Oracle8*i* except for the fact that the data in the table is organized as a B^+-tree index built on the primary key for the table. For example, an inverted index which is typically used by a web text-search engine for providing content-based search capability can be implemented using an index-organized table in the following manner:

```
CREATE TABLE inverted_doc ( token CHAR(20),
doc_id NUMBER, token_frequency NUMBER,
CONSTRAINT pk_inverted_doc PRIMARY KEY (token,
doc_id))
ORGANIZATION INDEX TABLESPACE ind_tbs;
```

This creates a table where all the row data, namely the primary key columns plus the remaining columns, are stored in the primary key B^+-tree index leaf blocks (Figure 1).

Such a storage organization enables fast primary key-based access to table data for queries involving exact match and/or range search. Once the search has located the target primary key, the remaining columns are present at the same location. This eliminates the need to follow a row identifier to table data, as would be the case with a conventional table and index structure, thereby avoiding an additional I/O. Furthermore storage requirements are reduced as there is no duplication of key columns in the table and the index, and row identifiers are not needed in the primary key B^+-tree structure.

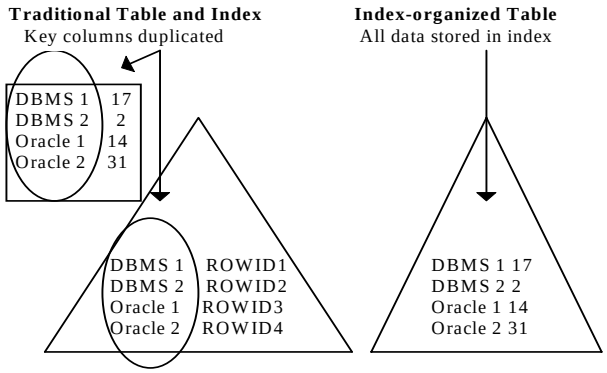

Figure 1: Conventional Table with an Index vs. Index-Organized Table.

Oracle8*i* applications can manipulate the index-organized table just like a conventional table using standard SQL statements.

2.2 Overflow Storage Area and Column Placement Options

Storing all non-key columns in the primary key B^+-tree index structure may not always be desirable or possible. Specifically:

- Each additional non-key column stored in the primary key index reduces the leaf row density of the

B^+-tree. To achieve better performance for access to frequently accessed columns, users may want to store only those columns in the index.

- Since a B^+-tree leaf block must hold at least two index rows, placing all non-key columns as part of index row may not always be possible.

To overcome these problems, an *overflow* storage area can be associated with an index-organized table. For example, if an additional column, say *token_offsets* is required for the *inverted_doc* schema, then the table can be created with an overflow storage area as follows:

```
CREATE TABLE inverted_doc ( token CHAR(20),
doc_id NUMBER, token_frequency NUMBER,
token_offsets VARCHAR(512), CONSTRAINT
pk_inverted_doc PRIMARY KEY (token, doc_id))
ORGANIZATION INDEX TABLESPACE ind_tbs
PCTTHRESHOLD 20 OVERFLOW TABLESPACE  ovf_tbs;
```

For such a table, the index row contains a <*key, head row-piece*> pair, where the *head row-piece* contains the first few non-key columns and a row identifier that points to the overflow portion containing remaining column values. Although this approach incurs the storage cost of one row identifier per row, key column duplication is still avoided.

Placement of columns into index and overflow storage area can be controlled by two options described below.

Placement Option for Handling Variable Length Rows

For a table with variable length rows, it is useful to allow each row to occupy not more than certain percentage of the index leaf block. This will ensure a lower bound on the leaf row density and enforce uniformity across rows as far as usage of the leaf block portion is concerned. This tuning is made possible using a percentage threshold parameter.

The PCTTHRESHOLD option, specified as a percentage of the leaf block size, determines the *last* non-key column that should be included in the index *head row-piece* on a per-row basis. The remaining non-key columns are stored in the overflow storage area as one or more row-pieces. Specifically, the last non-key column to be included is chosen such that the index row size (*key + head row-piece*) does not exceed the specified threshold, which in the above example is 20% of the index leaf block.

Such tuning ensures a lower bound of leaf row density which in turn limits the height of the B^+-tree, thereby providing fast access to head row-pieces for all rows. Larger rows, however, incur the cost of additional I/Os required to get to the tail row-pieces stored in the overflow area.

Placement Option for Speeding Up Access to Frequently Accessed Columns

PCTTHRESHOLD option puts a constraint on the index row size, which may translate to different sets of columns being included in the index for different rows. However, many applications may benefit from allowing the *same* set of columns to be included in the index for *all* rows in the table. The motivation here is to speed up access to frequently accessed columns by forcing the infrequently accessed columns out to reside in the overflow storage area. This is achieved by means of the INCLUDING parameter. For example, the following CREATE TABLE statement includes all the columns up to the *token_frequency* column in the index leaf block and forces the *token_offsets* column to the overflow area.

```
CREATE TABLE inverted_doc  . . .       . . .
ORGANIZATION INDEX TABLESPACE ind_tbs
INCLUDING token_frequency
OVERFLOW TABLESPACE  ovf_tbs;
```

Such vertical partitioning of a row between the index and overflow storage areas allows for higher leaf row density, resulting in better query performance for the columns stored in the index. While this approach incurs the cost of one additional block access for columns stored in the overflow, this I/O overhead is no worse than that of a conventional table with an index.

Note that the INCLUDING option only ensures that all columns after the specified column are stored in the overflow area. If this specification is such that the corresponding index row size exceeds the specified threshold, then the last non-key column to be included is determined based on the PCTTHRESHOLD.

The two column placement options discussed above allow breaking the table vertically into two partitions. Vertical partitioning of a table enables user queries to deal with smaller relations, which may result in a smaller number of block accesses [NCWD84, Niam78]. General vertical partitioning, although studied extensively, is typically not supported in most commercial database systems, leaving the task of generating heuristic fragmentation and allocation schemes [HN79] to the application designer. Note that even if such a support is made available, that would require duplication of the key attributes or row identifiers (tuple-ids) in the individual fragments, which is avoided in the limited form of vertical partitioning supported in index-organized tables. An exception to this is the Projection Indexes in Sybase IQ [SYB99], which only holds partitioned column values. However, the column values are not maintained in any index order as in index-organized tables.

2.3 Secondary Index with Guess-DBAs

Secondary indexes can be created on index-organized tables to provide alternate access paths. Unique, non-

unique, as well as function-based secondary indexes are supported. However, secondary indexes on index-organized tables differ from indexes on heap-organized tables in several ways:

- They store logical row identifiers instead of physical row identifiers. Thus, a table reorganization operation does not make its secondary indexes unusable.
- The logical row identifier includes primary key columns. Thus, a query involving indexed columns and/or primary key columns can be satisfied by an index-only scan.

While the above properties are desirable, there is one drawback when compared to a secondary index with physical row identifiers. During an index-based scan of an index with physical row identifier, each lookup needs just one extra I/O to fetch the base table columns. But a similar lookup using a secondary index with logical row identifiers requires I/O's equal to the height of the primary key B^+-tree to fetch the base table columns. Such primary key based lookup during index-based scan can degrade the performance substantially.

Guess-DBAs: To mitigate this problem, a 4-byte Database Block Address (DBA) of the primary key index leaf block where the base table row can be found, is stored as part of the logical row identifier. This DBA, referred to as *guess-DBA*, identifies the block where the row is *most likely* to be found for the reasons mentioned below.

The DBAs are populated correctly as part of secondary index creation. Further, DML operations implicitly maintain the guess-DBAs for the rows affected. Specifically, for inserts and updates that do not cause leaf block splits, corresponding rows in the secondary indexes are updated to contain the correct guess-DBAs. However, DML operations may cause a set of rows in the primary key B^+-tree index to move due to a leaf block split. This results in invalidating guess-DBAs stored in the secondary index rows corresponding to the base table rows that moved to a different leaf block.

An argument can be made for fixing the guess-DBAs for the rows that moved to a different leaf block, as part of the DML operation that caused the move. But this scheme was rejected since it would impose an unpredictable performance overhead on the DML. Also, the current design retains the simplicity of traditional B^+-tree index maintenance.

If the table rows do not move, then the guess-DBA will identify the correct leaf block. However, if the guess-DBA is invalid, then the cost of one extra I/O is incurred to fetch the block pointed by guess-DBA, before resorting to a primary key based lookup.

Guess-DBA Quality: To minimize use of invalid guess-DBAs, a statistic called *guess-DBA quality* is maintained for each secondary index. It is defined as percentage of valid guess-DBAs with respect to total number of rows in the secondary B^+-tree index.

The optimizer consults the guess-DBA quality to decide if it should use the guess-DBA or directly fall back to primary key based lookup to access the base table row. This statistic is implicitly set to 100 after the index creation. However, like other index statistics, it gets recomputed as part of the ANALYZE index statement, which analyzes the index and generates statistics for subsequent use by the optimizer.

If the guess-DBA quality drops below a certain threshold, the guess-DBAs can be fixed *online* using an ALTER index statement. Note that a rebuild of the index will also result in fixing the guess-DBAs. However, rebuild of index is typically done when the secondary index itself is fragmented due to large number of inserts and deletes.

The performance of the (logical row identifier-based) secondary index using valid guess-DBAs matches that of the secondary index with physical row identifiers as confirmed by experiments described in Section 4.3. Storing guess-DBAs reduces leaf row density causing extra I/Os for range scans. This overhead, however, is only a small fraction of the overall cost, the larger component being the I/Os needed, one per index row, to access the base table columns.

2.4 Key Compression

Index-organized tables can be compressed by eliminating common primary key column prefixes. Unlike conventional tables, the rows in an index-organized table are naturally clustered in the primary key order and there is a high likelihood of finding common prefixes for the key values.

The salient characteristics of the scheme are:

- Key compression applies to primary B^+-tree with multi-column primary key and specifically to keys in the leaf blocks.
- A key is broken into a prefix entry and a suffix entry at column boundary.
- Compression is achieved by sharing the common prefix entries among all the suffix entries within a leaf block.

Key compression can be enabled by specifying the COMPRESS clause as part of the physical attributes of the table, along with a prefix length (as number of columns) that specifies how the key can be broken into a prefix and a suffix. For example,

```
CREATE TABLE inverted_doc  . . .      . .
ORGANIZATION INDEX TABLESPACE ind_tbs
COMPRESS 1 INCLUDING token_frequency;
```

Here, single column prefixes (that is, token column) will be compressed in the primary key <token, doc_id> occurrences. For the list of primary key values <'DBMS', 1>, <'DBMS', 2>,<'Oracle', 1>, <'Oracle', 2>, the repeated occurrences of 'DBMS' and 'Oracle' are compressed away.

The scheme used provides row-level compression as opposed to block-level compression, which enables index-organized tables and their key-compressed secondary indexes to support the same degree of concurrency as their uncompressed counterparts. Also, our design decision to identify common prefixes at column boundaries avoids the need to reassemble the index row. Specifically, the scheme allows extraction of columns directly from the prefix and suffix entries. Thus, both exact and range scan performance of compressed index-organized tables are comparable to those for uncompressed counterparts as confirmed by the experiments described in Section 4.4.

2.5 Online Reorganization

For index-organized tables, reorganization may be needed more often than conventional tables. The data resides in a B^+-tree structure, which can get fragmented due to large number of inserts, updates, and/or deletes. In the event of such fragmentation, users can either *coalesce* or *rebuild* the primary key B^+-tree structure. These operations can be performed online. Furthermore, secondary index guess-DBAs invalidated by these operations can be fixed online to maintain guess-DBA based access performance. These online operations are discussed below.

Coalesce operation on the primary key B^+-tree coalesces (merges) leaf blocks within the same branch. This is an online operation that locks a few blocks at a time, and quickly frees them as soon as the coalesce on those blocks is completed. For example, the following command will coalesce the primary key B^+-tree index for the *inverted_doc* table.

```
ALTER TABLE inverted_doc COALESCE;
```

Rebuild (also termed *move*) operation on the primary key B^+-tree results in creating a new tree. By default, the table is not available for other operations during the move. However, the logical nature of secondary indexes makes the reorganization window smaller for index-organized table when compared to a conventional table, because the secondary indexes do not have to be updated. This will help in environments which can afford a limited downtime for database maintenance. However, for applications requiring 24x7 availability, the online move is supported which allows DML and query operations during the actual move operation. For example, the following command rebuilds the *inverted_doc* table online.

```
ALTER TABLE inverted_doc MOVE ONLINE;
```

Key-sequenced tables in Non-Stop SQL also support online move and coalesce operations [Troi96]. However, the analogous operations on index-organized tables differ in their handling of secondary indexes, due to presence of guess-DBAs. In our case, the guess-DBA quality is set to 0 as part of the move operation. This ensures that the optimizer resorts to primary key-based access for rows identified by the index scan, giving a performance comparable to that of key-sequenced tables. However, users can fix the guess-DBAs of the secondary index online by issuing an ALTER index statement. Thus, for index-organized table, the secondary index-based scan performance temporarily falls back to primary key-based access (see Figure 5(a) in Section 4.3) as result of move but improves back to the performance of valid guess-DBA based access (Figure 5(b)) after the online guess-DBA fix-up operation completes.

3 Index-Organized Table Applications

The superior query performance and storage savings of index-organized tables (and primary B^+-tree like structures, in general) have traditionally made them ideal for OLTP applications. However, index-organized tables are equally useful in several new domains. The table below summarizes the applicability of index-organized tables, especially its novel aspects:

App.	*Idx*	*C-Pl.*	*S-Idx*	*K-Cmp*	*O-Reorg*
E-Order	√		√		√
I-Search	√	√		√	√
I- Portals	√	√	√	√	
E-Catalog	√			√	
D-Ware	√	√	√	√	
T-Series	√			√	
D-specific	√	*	*	*	*

*Table 1: Applicability of Index-Organized Table features (Idx: Index-Organization, C-Pl: Column Placement, S-Idx: Secondary Indexes, K-Cmp: Key Compression, O-Reorg: Online Reorganization, *indicates that applicability is domain-specific)*

3.1 Electronic Order Processing

Electronic order processing is quite similar to the scheme described in the TPC-C benchmark [TPCC93]. For example, in an online store such as Amazon.com, an order-entry transaction accepts the order for a book or CD, assigns the order a unique order_id and inserts the order into an orders table with the status as 'confirmed'. The delivery transaction retrieves the order entry by order_id, processes the order by validating if all parameters of the order such as payment, shipping address, etc. are satisfactory, and updates the status of

this order entry as 'processed'. The fulfillment transaction retrieves all processed orders by order_id and status, and ships the item, changing the status of the order as 'fulfilled'. Fulfilled orders are periodically moved into a data warehouse. Apart from these state changing DML operations, an order tracking system may also make queries against the table to determine the status of an order.

An index-organized table is an ideal storage structure for the 'orders' table, where the query and DML is predominantly primary-key based. Further, the heavy volume of DML operations is bound to fragment the table, requiring frequent table reorganization. An index-organized table can be reorganized without invalidating its secondary indexes, and moreover, this reorganization can be done online as described in section 2.5, thereby reducing or eliminating the window of non-availability of the orders table. Furthermore, the guess-DBAs invalidated during reorganization of the base table can be fixed online.

3.2 Internet Search Engines

Internet search engines use web crawlers to index contents of the web and provide full-text search capability. The majority of the current search engines and full-text searching in text databases do not employ database technology [Ber+98]. We believe that future systems will be DBMS driven, since they can implicitly benefit from the high availability and scalability characteristics of DBMS.

The fundamental component of a search engine is an inverted index, which can be modeled using an index-organized table as shown in section 2. The example shown maintains one entry per <token, doc-id> pair, that is, the unit of inversion is a document. Since the token occurrence data can be of variable size, we can limit the amount that is stored in the index using the PCTTHRESHOLD column placement option, and thereby ensure a lower bound on the leaf row density. This capability is even more desirable when the unit of inversion is a set of documents or the entire collection. In such cases, the occurrence data can be huge, especially for popular tokens. For a given token, the corresponding inverted index entry can be reached by just traversing the primary B^+-tree. In most cases, only a single physical I/O is needed to get to the inverted index row, since the branch blocks will be cached.

The inevitable fragmentation of the inverted index structure due to the huge volume of insertions will mandate frequent reorganization, which means downtime - which is highly undesirable for web sites. Index-organized tables, owing to their support for online reorganization, are suitable for such situations since they can help reduce, or eliminate, downtime.

Although most popular Internet search engines can effectively index content stored in files, they still do not index the vast amounts of information tucked away in *databases*. Database search engines such as Jungle [BBD99] access and advertise databases on the web. Similarly, Infoseek's JavaSeek Search Server [JSEEK] provides full-text search capability for any field in an Oracle database, and uses index-organized tables to store the full-text index.

3.3 Internet Portals

A majority of web sites, including portals, foster online communities. Arsdigita Community System (ACS) is an open source toolkit, based on Oracle8 RDBMS, that can be used to build and manage online communities [Gre99]. One critical function of this toolkit is to track user activity. For example, the User/Content Map module tracks which users have read which pieces of web content. The table holding user/content map is modeled as an index-organized table with <user, page_id> as the primary key. In general, we believe that the toolkits such as ACS, used for building and managing web sites, will have a significant number of database tables with primary keys, which can be best modeled as index-organized tables.

Web applications such as portals and auction sites maintain a database of users. These user tables can be based on index-organized tables. Since only a part of the user information is accessed more frequently than the rest, the flexible column placement options provide the ability to push infrequently accessed data to the overflow storage area.

For queries requiring alternate access paths (based on, for example, zipcode, credit_card_id, etc.), secondary indexes built on these columns perform as well as indexes on conventional tables. This is because the base tables are typically non-volatile and hence help retain the validity of guess-DBAs in the secondary indexes. However, inserts can still cause splits leading to invalidation of guess-DBAs. In such an event, the indexes can be fixed online as described in Section 2.3.

The popularity of Yahoo's directory-like structure among portals confirms that organizing web content hierarchically by categories is, and will remain, the de-facto standard. Index-organized tables are ideal for storing the multi-column primary keys, composed of the category attributes which represent the hierarchy, along with the URL and any additional information. Such tables can also benefit from key compression.

3.4 Electronic Catalogs

Electronic catalogs are essential components of E-commerce. A typical procurement process involves product selection, source selection, negotiation of price,

ordering, order fulfillment and finally payment [AK97]. Two types of catalogs are usually needed:

Manufacturer's Catalog: Manufacturers supply several products which might have different sets of attributes, whose diversity can make the task of catalog management very challenging. [Dan98] suggests that the product descriptions be modeled as rows in a table with <product_id, attribute_name, attribute_value> such that there would be one entry per attribute per product. This generalized scheme can easily support addition of entirely new products or addition of new attributes to existing products.

Retailer's Catalog: A retailer selling items from multiple manufacturers needs to group the products into different (buyer-friendly) categories, which may have sub-categories depending on how the vendor wants to present the products. The retailer can either poll all the manufacturers and build a catalog (Table 2) or the catalog can be a "virtual" one which is dynamically constructed from the manufacturer's catalogs [AK97]. Additionally, the retailer might want to "materialize" the virtual catalog for better performance.

Category	*Sub-Category*	*ItemName*	*Price*
Book	Technical	Oracle8 Tuning	45
Book	Science Fictior	Time Machine	20
...	...	...	...

Table 2: A retailer's catalog constructed by polling manufacturer's catalogs

Assuming a database driven catalog, the resulting structure (Table 2) will have a set of category columns (determined by the retailer) in addition to the attributes retrieved from the manufacturer's catalog. An ideal storage organization should hierarchically cluster these entries on the category columns.

An index-organized table can be used to store these manufacturer catalogs, indexed on <product id, attribute name>, will cluster all attributes of a product together. Similarly, for the retailer's catalog, an index-organized table with a multi-column primary key matching the hierarchy of these categories is a natural fit. Key compression can be used to avoid same <product_id> and <category, subcategory> column value repetitions in the manufacturer's and retailer's catalogs, respectively.

3.5 Data Warehousing

The efforts to speed up data warehousing applications have focused on two areas: supporting ad-hoc queries, and supporting known or expected set of queries. For example, [OQ97] discusses how ad-hoc style queries can be efficiently evaluated using index and clustering techniques, whereas [JLS99] deals with known or given workloads, specifically with the problem of finding optimal ways to cluster records of a fact table to minimize I/Os. We believe that index-organized tables (or other similar primary B^+-tree structures) can be used to implement fact tables for the latter class of applications.

Consider a warehouse application illustrated in [JLS99] with relations:

```
location(state, city, lid)
jeans(type, gender, jid)
sales(lid, jid, sale)
```

A typical OLAP query is:

```
SELECT SUM(sales) FROM sales, location, jeans
WHERE  sales.lid = location.lid AND sales.jid
= jeans.jid AND   location.state = NY AND
jeans.type = 'LEVI';
```

Implementing the fact table ('sales') as an index-organized table with primary key <lid, jid> will yield better performance than a conventional table with a primary key index. The join result of <location, jeans> after applying the corresponding filters is used to look-up for <lid, jid> pair in the sales table. This fact table look-up will result in an index-based scan for a conventional table, whereas an index-only scan is sufficient for index-organized table (for the performance characteristics see Section 4.1). The data warehouse applications will also benefit from various novel aspects as indicated in Table 1 with similar reasoning as in Section 3.3.

Support for summary tables and materialized views in Oracle [Bel+98] are being implemented for index-organized tables, and bitmapped secondary indexes will also be available in a future Oracle8 release.

3.6 Time-Series

A time-series is a set of time stamped rows belonging to a single item, such as stock price. Data is accessed through an item identifier such as stock symbol and timestamp. By defining an index-organized table with primary key <stock symbol, timestamp>, the Oracle8 Time Series Data Cartridge [OTS97] is able to efficiently store and manipulate time-series data. Repeated occurrences of the item identifier (e.g., stock symbol) key can be compressed, leading to additional storage savings.

In our experience, individual rows are small in size and hence column placement options are not needed. Data is inserted at the end of subgroups, for example, adding new values for each stock symbol that monotonically increase with time. Deletions are rare. Thus, the primary B^+-trees do not tend to fragment, and hence reorganization is usually not required. The use of secondary indexes is also not very common.

3.7 Domain-Specific Indexing

Oracle8*i* introduces the Extensible Indexing Framework [SMSAD00], through which users can add a new access method to the RDBMS engine. Typically, domain-specific indexing schemes need some storage mechanism to hold their index data. Index-organized tables are ideal

candidates for such domain index storage. Oracle8 *inter*Media Text Cartridge [OIMT99] has implemented domain-specific indexing schemes that use index-organized tables for storing their index data.

4 Performance Study

This goal of this study is to validate the clustering benefits of primary B^+-tree, to illustrate the effect of overflow, to show the effect of logical row identifiers containing guess-DBA, and to illustrate the benefits of key compression.

All the experiments (except key compression) are conducted using Oracle8*i*, Release 8.1.6 configured with 4K database block size, 16MB of database buffer cache on a SunOS 5.6, single CPU Ultra-60 Sparc with 256MB of main memory. The order_line table of TPC-C benchmark [TPCC93], which models a product sales and distribution business, is used as the reference table.

```
CREATE TABLE order_line ( ol_o_id NUMBER,
ol_w_id NUMBER, ol_d_id NUMBER, ol_number
NUMBER,  ol_i_id NUMBER, ol_supply_w_id
NUMBER, ol_quantity NUMBER,  ol_amount
NUMBER(6), ol_delivery_date DATE, ol_dist_info
CHAR(24),
CONSTRAINT pk_orderline PRIMARY KEY (ol_w_id,
ol_d_id, ol_o_id, ol_number));
```

4.1 Performance of Index-organized Table without Overflow

Query and DML execution times are measured for data sizes varying from 100MB (~1.75 million rows) to 500MB (~8.74 million rows). The average and maximum standard deviation (s.d) over 10 iterations are reported. Data for the order_line table is generated in random order. The time for bulk-load into an index-organized table is more than that for the heap-organized table (for example, 28 minutes vs. 17 minutes for 200MB). This can be attributed to larger sort overhead due to inclusion of non-key columns in the sort entries in the case of index-organized table.

Storage requirement for index-organized table is less than that for heap-organized table (for example, 268MB vs. 344MB for 200MB data). The additional storage needed for heap-organized tables can be attributed to duplication of key columns in the table and the primary key index. For the heap-organized table, the primary key index storage required 93MB and the actual table storage required 251MB for 200MB data.

Query Performance

In order to compare the query performance for *random* and *range* access, total time taken to access 1000 random order_line rows and the total time for selecting 100,000 consecutive order_line rows are measured.

The index-organized table shows superior query performance for both random and range scans. The random access on the 100MB table took 0.92 seconds for index-organized table as compared to 1.16 seconds for the heap-organized table (See Figure 2). The same performance is observed for larger data sets. The faster random access for the index-organized table results from finding both key and non-key columns in the primary key index leaf block. The range scan query performance is consistently faster for index-organized tables (See Figure 2). For the 100MB table, a range-scan on index-organized table took 1.20 seconds as compared to 4.35 seconds for the heap-organized tables. Similar performance is observed for larger data sets. The speed-up can be attributed to the fact that the rows are clustered in primary key order for index-organized tables, plus an additional random I/O per row is avoided.

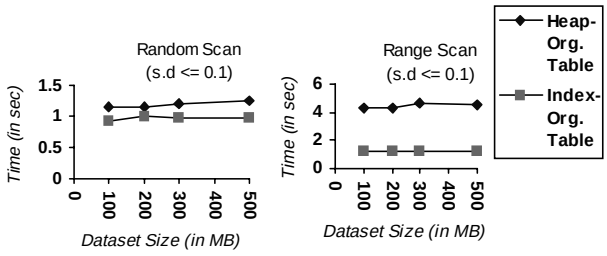

Figure 2: Random and Range Scan Time

DML Performance

Time taken for inserting, updating, and deleting 2000 order_line rows are measured and compared for the two organizations. The update statement involved modification of the delivery_date field. Overall, while insert and update performance of the two table organizations are comparable, delete performs better for index-organized tables (Figure 3). The anomaly in the 200MB case (Figures 3(a) and 3(b)), is because of a larger B^+-Tree height for the index-organized table as compared to that for primary key index on heap-organized table (4 vs. 3). Compared to updates where only an extra I/O is incurred, this difference in time is larger for inserts due to branch block splits in the additional level.

Single row updates in both heap and index-organized tables involve finding the target row via primary-key index and then modifying a single structure (table or index, respectively), so it is easy to argue that performance of such updates will be comparable. We focus instead on performance of updating multiple rows with consecutive primary-key values. When compared to just the primary key index of a conventional table, the number of index blocks accessed for selecting the rows that satisfy the where-clause is higher for an index-organized table, since inclusion of non-key columns

makes the size of its index rows larger than that of primary-key index rows. However, due to guaranteed clustered placement of consecutive rows, the number of disk block I/Os for accessing and modifying target base table rows is significantly lower, since heap-organized tables generally lack such clustering. The latter reduction in number of blocks is usually sufficient to offset the earlier increase. This is reflected in our results (Figure 3(b)) which shows comparable update performance for index-organized and heap-organized tables. Index-organized tables will usually have better delete performance because only a single structure (index) needs to be modified as opposed to modifying two structures (index and table) for heap-organized tables. This is reflected in our results (See Figure 3(c)).

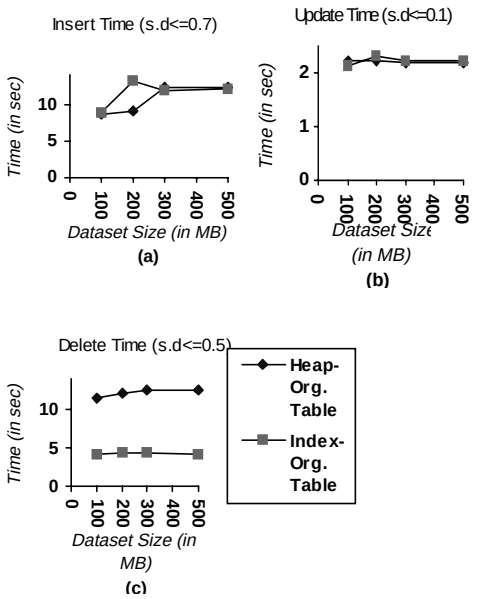

Figure 3: Insert, Update, and Delete Time: without overflow.

4.2 Performance of Index-organized Table with Overflow

This experiment compares the performance of order_line table with column ol_dist_info pushed out to overflow storage area, with that of the table without overflow. Two types of Queries and DML statements were used against 500MB of data: (1) those accessing only *index-resident* columns, that is, columns that have not been pushed out to the overflow, and (2) those accessing the *overflow-resident* column.

Query Performance

Pushing the column ol_dist_info to the overflow results in higher leaf row density as less number of column values per row needs to be stored in the index.

Accessing only index-resident column(s) : Higher leaf row density achieved through the use of overflow leads to improved performance for any query that accesses only index-resident columns. This improvement is particularly evident in the case of range access as less number of index leaf blocks needs to be accessed (Figure 4: 0.98 sec each for exact match, 1.21 sec. vs. 1.01 sec for range scan).

Accessing overflow-resident column(s) : Access to overflow-resident columns requires access to the index leaf block followed by an access to the appropriate overflow block, which in fact is similar to block access sequence needed for heap-organized tables. Thus, in this configuration, both random and range access query performance deteriorates from 0.98 sec to 1.31 sec and from 1.43 sec to 4.75 sec, respectively, which are comparable to query performance for heap-organized table (1.24 sec and 4.78 sec, respectively). The range access query performance for accessing an overflow-resident column improves to 2.09 sec if it follows a table rebuild, since the rebuild clusters overflow row-pieces in primary key order.

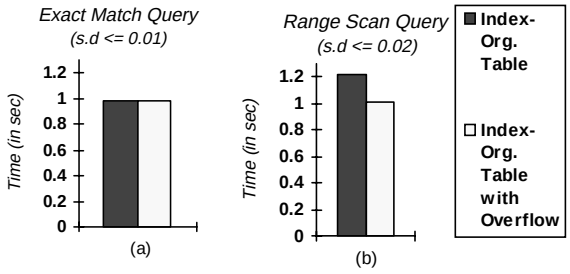

Figure 4: Time for random and range query accessing index-resident column(s) in 500MB index-organized tables.

DML Performance

Insert : Insert performance improves from 12.14 sec to 11.61 sec, mainly because reduction in index row size lowers the frequency of splits during insertion into the B^+-Tree.

Delete : Delete performance however, deteriorates from 4.17 sec to 11.03 sec (approaching delete performance for heap-organized table which takes 12.62 sec) as two structures, index and overflow, need to be modified for deleting the target base table rows.

Update : Higher leaf row density achieved through the use of overflow leads to improved performance for any update that modifies only index-resident columns. On the other hand, any update that modifies an overflow-resident column performs worse because it requires accessing the target index leaf block and also the appropriate overflow block.

4.3 Performance of Secondary Index

This experiment compares the performance of secondary index on a 500MB order_line table, created with index and heap organization. The secondary index is created on the ol_i_id column.

Guess DBA Usage: The *guess-DBA* is stored in the logical row identifier improves the performance of a secondary index-based scan by avoiding the primary key traversal when possible, as confirmed by this experiment. Index-based scan using valid *guess-DBAs* matches the performance of secondary index- based scan on heap-organized tables (Figure 5(b)). The following query is used for this test, and it returns 8000 rows:

```
Q1: SELECT SUM(ol_amount) FROM order_line
WHERE ol_i_id BETWEEN 100001 AND 100050;
```

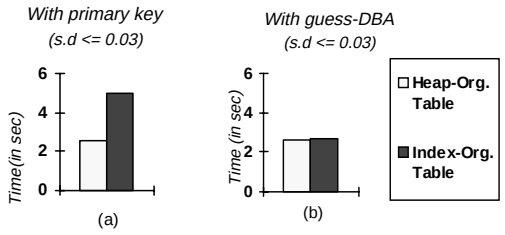

Figure 5: Time for index-based scan

Index Availability: A table reorganization renders the secondary index of a conventional table unusable. Subsequent to the reorganization, this would cause queries (for example, Q1 stated above) that originally required an index-based scan to incur the cost of a full-table scan. However, when an index-organized table is rebuilt, its secondary index remains usable (albeit with invalidated guess-DBAs), thereby allowing an index-based scan for such queries. Thus, for index-organized tables the time for index-based scan after reorganization is significantly better (5.07 sec for index-organized table vs. 74.98 sec for heap-organized table).

Index-Only Scan versus Index-Based Scan: The following query performs an index-only scan, and returns 200000 rows:

```
Q2: SELECT COUNT(*) FROM order_line WHERE
ol_i_id BETWEEN 100001 AND 100500;
```

In this experiment, the leaf row density of the secondary index for the index-organized table is lower, owing to its maintaining the 4-column primary key, when compared to the index on the conventional table. The resulting increase in leaf block fetches explains its poorer performance of index-only scan when compared to an index on a conventional table (0.7 sec for heap-organized table vs. 1.0 sec for index-organized table).

However, for a query that involves accessing the secondary key and the primary key columns, such as:

```
Q3: SELECT COUNT(*) FROM order_line WHERE
ol_i_id BETWEEN 100001 AND 100500 AND
ol_d_id=10;
```

(which fetches 20000 rows), a secondary index-only scan is sufficient for index-organized tables because the column ol_d_id is available as part of the index row. For a heap-organized table, this would require a secondary index-based scan. This results in significant performance benefit for a secondary index on index-organized table (0.1sec for index-organized table vs. 156.3 sec for heap-organized table).

4.4 Performance of Key-Compressed Index-Organized Table

This experiment compares the performance of a 500MB order_line table with and without compression. The experiment is conducted with Oracle8*i*, Release 2 on a dual CPU Ultra-60 Sparc with 512MB of main memory For the four column primary key, three column prefixes are compressed. The data set typically has the same prefix for 10 order lines. The compression resulted in 12% storage reduction in the primary key index leaf blocks.

Time taken to access 1000 random order_line rows and for selecting 100,000 consecutive order_line rows is measured for the index-organized table with and without compression. The random lookup query took 0.71 and 0.73 seconds for compressed and uncompressed configurations respectively. The additional CPU overhead to process compressed index-organized table did not have any significant impact on the random scan performance.

Range scan query took 0.77 seconds and 0.74 seconds for compressed and uncompressed configurations. For range scan, 12% fewer blocks are needed for the compressed organization when compared to the uncompressed organization. However, the savings in I/Os is offset by the additional CPU overhead incurred to extract the columns from the compressed table. This resulted in comparable range scan performance.

5 Conclusions and Future Work

The primary B^+-tree structure, with row data in leaf blocks, is an ideal storage organization for primary key access intensive applications. Traditionally, use of such structures has been limited to OLTP applications. However, we argue that an index-organized table with its support for several additional features, is equally useful in several new domains. Specifically, we discuss its use in Electronic Order Processing, Internet Search Engines,

Internet Portals, Electronic Catalogs, Time-Series, and Data Warehouse applications. The applicability and performance of index-organized tables is enhanced by support for column placement control, use of logical row identifiers with guess-DBAs in the secondary indexes, key compression, and online reorganization.

The performance study demonstrates the superior query performance for both random and range scans for index-organized tables and comparable DML performance to heap-organized tables. The experiment on the effect of overflow demonstrates the benefit of controlling placement of columns between index and overflow storage areas**.** Secondary indexes on index-organized tables with valid guess-DBAs showed matching query performance with those on heap-organized tables. The key-compression experiment illustrates the storage savings can be achieved without degradation in query performance.

We also plan to support bitmap indexes on index-organized tables to increase its applicability to data warehousing domain. For applications with large primary keys that require several alternate access paths, the current scheme of using primary key based logical row identifiers in secondary indexes is not suitable. For handing such applications, we are investigating an alternate scheme that can optionally fall back to physical row identifiers.

Acknowledgments

We thank Jonathan Klein, Bhaskar Himatsingka, Wei Huang, and Vishwanath Karra for implementing online move support, and their design reviews. For logical ROWID support, we thank Alex Tsukerman. We thank Gopal Krishnan for conducting the initial set of experiments on index-organized tables. Special thanks to Anil Nori and Franco Putzolu for their involvement in defining the index-organized table functionality.

References

[AK97] Keller, A., "Smart Catalogs and Virtual Catalogs," *Readings in Electronic Commerce*, Kalakota, R., Whinston, A.B.(eds.), Chapter 11: 259-274, 1997.

[BBD99] Bohlen, M., Bukauskas, L., Dyreson, C., "The Jungle Database Search Engine," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*: 584-586, May 1999.

[Bel+98] Bello, R. G., et al., "Materialized Views in Oracle," *Proceedings of VLDB*, pp. 659-664, 1998.

[Ber+98] Bernstein, P., et al., "The Asilomar Report on Database Research," *SIGMOD Record*, 27(4): 74-80, Dec. 1998.

[BSZ98] Bichler, M., Segev, A., Zhao, J.L., "Component-based E-Commerce: Assessment of Current Practices and Future Directions," *ACM SIGMOD Record*, 27(4): 7-14, Dec. 1998.

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

[Dan98] Danish, S., "Building database driven electronic catalogs," *ACM SIGMOD Record*, 27(4): 15-20, Dec. 1998.

[DB2V5.2] The SQL Reference. *DB2 Universal Database Version 5.2* Publication.

[Gre99] Greenspun, P., "Philip and Alex's Guide to Web Publishing," *Academic Press/Morgan Kaufmann*, Apr. 1999.

[HN79] Hammer, M., Niamir, B., "A Hueristic Approach to Attribute Partitioning," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp.93-101, 1979.

[JLS99] Jagadish, H.V., Lakshmanan, L.V.S.,Srivastava, D., "Snakes and Sandwiches: Optimal Clustering Strategies for a Data Warehouse," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp.37-48, May 1999.

[JSEEK] Javaseek Search Server. Available at *http:://software.infoseek.com/products/javaseek/applications.htm* .

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

[NCWD84] Navathe, S.B., Ceri, S., Wiederhold, G., Dou, J., "Vertical Partitioning of Algorithms for Database Design," *ACM TODS*,pp. 680-710, 1984.

[Niam78] Niamir, B., "Attribute Partitioning in a Self-Aaptive Relational Database System," *Technical Report 192*, LCS, MIT, 1978.

[OAQ97] *Oracle8 Application Developer's Guide - Advance Queuing*, Oracle Corp., Part # A58241-01, Jun. 1997.

[OIMT99] *Oracle8i interMedia Text Reference, Release 8.1.5*, Oracle Corp., Part No. A67843-01, Feb. 1999.

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

[OSC97] *Oracle8 Server Concepts Volume I & II*: Oracle Corp., Part # A54644-01 & A54646-01, Jun. 1997.

[OTS97] *Oracle8 Time Series Cartridge User's Guide Release 8.1*, Oracle Corp., Part # A64429-01, Jun. 1997.

[RDC96] Rengarajan, T.K., Dimino, L., Chung, D., "Sybase System 11 Online Capabilities," *Data Engineering Bulletin*, pp.18-23, Jun. 1996.

[SI96] Sockut, G.H., Iyer, B.R., "A Survey of Online Reorganization in IBM Products and Research," *Data Engineering Bulletin*, pp.4-11, Jun. 1996.

[SMSAD00] Srinivasan, J., Murthy, R., Sundara, S., Agarwal, N., DeFazio, S., "Extensible Indexing: A Framework for Integrating Domain-Specific Indexing into Oracle8i," *Proceedings of the 16th Data Engineering Conf.*, pp. 91-100, Mar. 2000.

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

[SYB99] Sybase Adaptive Server IQ Administration and Performance Guide, Adaptive Server IQ Release 12.0 Collection, Chapter 4, 1999.

[Tand87] The Tandem Database Group, "NonStop SQL: A Distributed, High-performance, High-availability Implementation of SQL," *Proc. 2nd Int. Workshop on High Performance Transaction Systems*, Springer Lecture Notes in Computer Science No. 359, pp60-104, 1987.

[TPCC93] Gray, J. (Editor), The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann Publishers, 1993.

[Troi96] Troisi, J., "NonStop SQL/MP Availability and Database Configuration Operations," *Data Engineering Bulletin*, pp.12-17, Jun. 1996.

[ZMS92] Zobel, J., Moffat, A., Sacks-Davis, R., "An Efficient Indexing Technique for Full Text Databases," *Proceedings of VLDB*, pp. 352-362, 1992.

[ZM98] Zou, C., Salzberg, B., "Safely and Efficiently Updating References during On-line Reorganization," *Proceedings of VLDB*, pp. 512-522, 1998.