Efficient Search of Multidimensional B-Trees

Harry Leslie Rohit Jain Dave Birdsall Hedieh Yaghmai

Tandem Computers Incorporated
10100 N. Tantau Ave., LOC 251-05
Cupertino, CA 95014-2542
{leslie_harry, jain_rohit, birdsall_dave, yaghmai_hedieh} @tandem.com

Abstract

Data in relational databases is frequently stored and retrieved using B-Trees. In decision support applications the key of the B-Tree frequently involves the concatenation of several fields of the relational table. During retrieval, it is desirable to be able to access a small subset of the table based on partial key information, where some fields of the key may either not be present, involve ranges, or lists of values. It is also advantageous to allow this type of access with general expressions involving any combination of disjuncts on key columns. This paper describes a method whereby B-Trees can be efficiently used to retrieve small subsets, thus avoiding large scans of potentially huge tables. Another benefit is the ability of this method to reduce the need for additional secondary indexes, thus saving space, maintenance cost, and random accesses.

Introduction

In the last few years various factors have resulted in Decision Support Systems (DSS) gaining popularity. Some of these factors have been:

- A downward trend in hardware server and disk costs.
- The evolution of database products, which are giving improved performance on an increasing number of hardware platforms.
- Use of information for competitive advantage. This trend has been prompted by a desire to provide higher levels of service to customers or improve targeting of customers to draw them away from increased competition. This has manifested itself in promotions such as the frequent flyer or buyer programs. Also, information is being used to decrease costs such as store inventory, etc.
- The advantage of seeing more and more detail combined with improved performance has prompted many users to move to DSS platforms. If the movement of men's jeans off the shelf were being monitored before, now it is the size 32, black, brand X, style Y, men's jeans that is under close scrutiny. Also, the time dimension being considered has slowly shrunk from quarterly trends to monthly, weekly, and now daily trends. In a word, micro-marketing is in vogue. Multidimensional databases that hosted such DSS often cannot support such level of detail.
- Increased availability of client tools that provide easier access to the information along with desktop tools that facilitate the further filtering, summarization, and presentation of the information. Multidimensional front end tools are also gaining popularity that provide the capabilities previously available in multidimensional databases, but now are targeted towards large relational data warehouses.

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

The above is by no means an exhaustive list, but it is a reality today that Decision Support Systems are being implemented by many DP shops and the sizes of DSS databases often dwarf the size of the largest operational databases. This trend can be substantiated by many studies done by firms such as the Gartner Group and the Meta Group.

Along with the above trend has been the increase in *dimensional modeling*. This is the design of the database based on *key dimensions of the business*. These dimensions may be time, geography, product, and so forth. Typically such dimensions are used to query these large databases first for summary information and then for detail information as the user drills down to increasing levels of detail. Summary information provides high level trends, but many important insights offering competitive advantage can only be discovered from the details.

When providing the criteria for these dimensions in a query, a user may or may not provide values for all the dimensions. So for a Sales table with dimensions such as Date, Store and Item the user may want sales for a week for an item across all stores. No matching criteria may be provided for the Store dimension, or for a particular dimension the user may provide a set of values. So the user may want information for 4 weeks of sales for a group of stores for certain items. The criteria specified for these dimensions may vary considerably from one query to another based on the type of user. A buyer for a large retailer has different objectives from a person in the marketing or inventory control departments accessing the same database. There is also a difference between users consistently submitting similar queries for operational decisions versus those users who are looking for strategic information. The latter type of user is data mining or data surfing, or in other words looking for some correlation or trend which may not be obvious from standard reports, and is best found by detailed investigation of anomalies. All of this leads to a challenge in index design. When the database is very large (100 GB to Terabytes) this may mean alternate indexes are prohibitively expensive. So one has to rely on a single index (the B-tree clustering index) to meet the requirements of these varied demands.

Typically most DBMSs have to scan the whole table when faced with the kind of queries mentioned above -- with a set of values specified for multiple columns of the index, ranges on intervening columns, or no predicates specified for leading or intervening key columns.

This is where the MultiDimensional Access Method (MDAM) steps in. It is a new access method that works efficiently with such multidimensional access against standard B-Tree based tables.

Multidimensional B-Trees

A multidimensional B-Tree is one that is defined on multiple columns (dimensions).

Subsets of data can be read efficiently from a B-Tree because they are clustered according to the values contained in their key columns. This is because records containing successive key values are stored in one or more blocks of the B-Tree. So, many records are read by a single access of these contiguous blocks.

In figure 1 below a B-Tree is represented with three dimensions (or columns). The total number of rows in this table is 27. Each dimension has three values. Note that for dimension 2 each value is repeated for each value of dimension 1. The first (smallest labeled a) shaded area shows how much is retrieved for an equal predicate on all three dimensions (dimenision_1 = 1 and dimension_2 = 2 and dimension_3 = 2).. The second shaded area (b) shows how much is retrieved for equal predicates on the first two dimensions (dimension_1 = 2 and dimension_2 = 2). and the third shaded area (c) shows how much is retrieved for an equal predicate on dimension 1 (dimension_1 = 3).

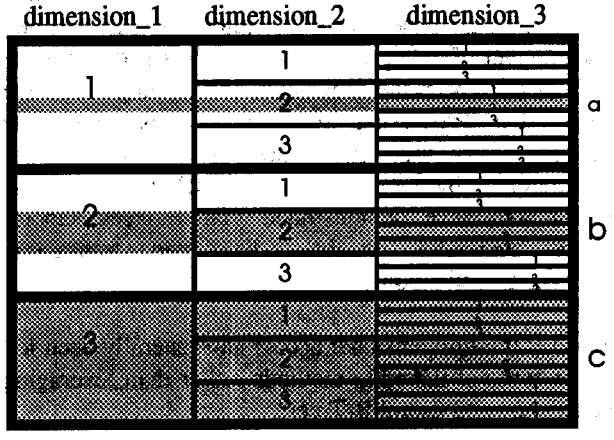

Figure 1

Figure 2 shows the same B_Tree, but shows what is retrieved when there is only a predicate on the third column (dimension_3 = 2).

In this case (dimension_3 = 2) the same value for dimension_3 exists for each value of dimension_2 (1, 2 and 3) and each value of dimension_1 (1, 2 and 3).

MDAM can do this type of access (with a predicate only on the third dimension) using an index, whereas other data base management systems using B-Trees would have to read the entire table or require a secondary index.

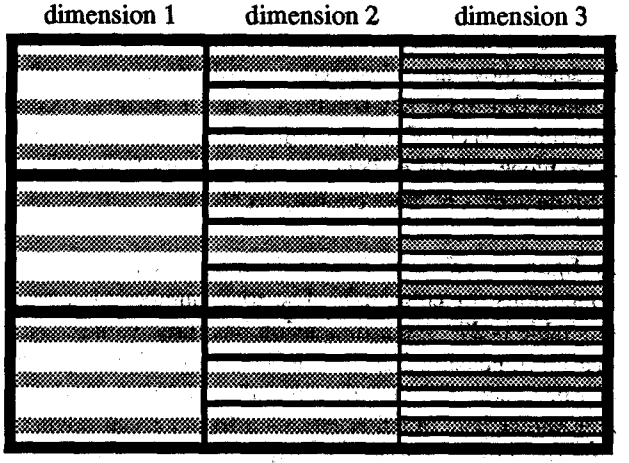

Figure 2

Multidimensional Access to B-Trees

Much has been written over the years about B-Trees. Almost all vendors implement them as the primary type of index structure. They perform well for a large variety of applications. The updating, balancing, caching, and management of B-Trees have well understood solutions. B-Trees have been the foundation of very large On-Line Transaction Processing (OLTP) systems and have facilitated very high transaction processing rates, as demonstrated by various TPC-C benchmarks. These benchmarks have demonstrated that B-Trees support highly available, scalable, distributed transaction processing applications.

However multidimensional access has generally been left to new types of indexing methods rather than leveraging the existing and useful B-Trees.

As shown in this paper multidimensional access of existing B-Trees can be efficient for a large range of multidimensional queries. B-Trees have already proven their usefulness in dealing with large volumes of updates across large databases.

Tandem allows use of B-Trees for both the base table and secondary indexes. In the base table the leaf nodes are the data blocks that contain the data records. In a secondary index, each record in the leaf node block contains primary index columns that identify an individual data record in the base table. Both the base table and secondary indexes can be range partitioned by the leading key columns of the primary and secondary index respectively.

MDAM makes existing B-Tree indexes much more useful by extending their use to a broad range of queries that can use them efficiently, thus improving response time and reducing the need for additional secondary indexes. When the need for secondary indexes is reduced the disk requirements for large databases are also reduced. In addition the choices for database organization are expanded. (i.e. Columns of the base table that do not contain many unique values and are not often referenced in queries can now be used as leading key columns of an index.)

This paper demonstrates how MDAM exploits existing B-Tree indexes for many more queries far more efficiently than previous B-Tree access methods.

Major Features of MDAM

Earlier discussions allude to some of MDAM's advantages. They are summarized here:

- Range predicates on leading or intervening key columns.
- Missing predicates on leading or intervening key columns.
- IN lists on multiple key columns.
- NOT equal (<>) predicates.
- Multivalued predicates.
- Elimination of redundant predicates.
- Elimination of empty sets.
- Pre-execution duplicate elimination.
- Preserving index order for rows retrieved.
- Sparse or Dense Access

The significant point is that while providing all these capabilities MDAM reads the minimal set of records required to process the query. Also, it transforms the original set of predicates into predicates on disjoint

ranges. This avoids the overhead of reading the same row many times and then having to eliminate duplicates. The ranges are ordered according to the order of the index. So MDAM maintains the order of the rows returned to that of the index. This facilitates efficient grouping , merge joins, and reduces the sorts sometimes necessary to order the data.

Each of the above features is discussed below. The table used to illustrate the features is:

SALES with the columns:

- dept
- date
- item_class
- store
- item
- total_sales

With a key on columns dept, date, item_class, and store.

Intervening Range Predicates

An intervening range predicate is a range predicate specified on a leading or intermediate column of an index. Columns preceding and following this column may have equality predicates specified on them. An example of a query with such predicates is:

```
SELECT date, sum(total_sales)
  FROM Sales
 WHERE dept = 10
       and date between "06/01/95" and "06/30/95"
       and item_class = 20
       and store = 250
GROUP BY dept, date;
```

Normally the predicates on column item_class and store cannot be used as keys by most DBMS' because of the intervening range predicate on date. However MDAM allows the use of all four key columns.

MDAM processes range predicates by stepping through the values, existing in the table, for the column on which the range has been specified. Assume that the values for date in the database between 06/01/95 and 06/30/95 are 06/04/95, 06/11/95, 06/18/95, and 06/25/95.

MDAM first searches for the value a date greater than or equal to 06/01/95 and less than or equal to 06/30/95 where dept equals 10. It finds the value 06/04/95. Then it retrieves all rows that are qualified by the following set of predicates:

dept=10, date="06/04/95", item_class=20, store = 250

We call this retrieval of a set of rows an access. After it gets all the rows that satisfy such an access, it searches for the next value of date for dept 10 which is greater than 06/04/95. We call this search a probe. It finds 06/11/95. Next it retrieves all rows that are qualified by the following set of predicates:

dept=10, date="06/11/95", item_class=20, store=250

In this manner, it continues processing the next value for date in the database and so on.

Even when a large number of rows are being processed (often thousands to millions), the overhead of each access to find the next value for the column in the database, as illustrated in the above example, is minuscule. If there are many item classes and stores MDAM will have avoided accessing thousands, if not millions, of rows.

No access was required to another dimension table to determine all the possible values for date in that range and then perform a join to accomplish the same thing. Besides, the values resulting from such a table may be all possible values for the column, whereas the database may have only a subset of such values. Because MDAM maintains the order of the index, the aggregation specified in the query can be performed very efficiently, without a sort being necessary.

Missing Key Predicate

When no predicates have been specified for a leading or intervening key column, MDAM can still use the subsequent columns for keyed access. The following query is an example:

```
SELECT date, sum(total_sales)
  FROM sales
 WHERE
          date between "06/01/95" and "06/30/95"
       and item_class = 20
       and store = 250
GROUP BY dept, date;
```

Note that no predicate has been provided for the first key column dept. Most DBMSs would not be able to use the index for keyed access and would have to resort to a full table scan for such a query. However, MDAM can handle this query efficiently. It treats the missing predicate for dept as an implied range of MIN_VALUE to MAX_VALUE (note that this includes NULL values).

These are respectively the minimum and maximum permissible values supported for the datatype of the missing key column.

So let us assume that the values for the column dept in the table range from 1 through 100. MDAM first searches for a dept greater than MIN_VALUE. It finds the value 1. Next it finds the first value for date as describe above and then uses the following set of predicates for its first access:

dept=1, date="06/04/95", item_class=20, store=250

After retrieving the rows for this access, it will vary the value for the column date, as discussed earlier, to do the following accesses against the table:

dept=1, date="06/11/95", item_class=20, store=250
dept=1, date="06/18/95", item_class=20, store=250
dept=1, date="06/25/95", item_class=20, store=250

Having covered the range of dates, it then increments the previous value for dept by 1 to do the following accesses, starting with the first date value again:

dept=2, date="06/04/95", item_class=20, store=250
dept=2, date="06/11/95", item_class=20, store=250
dept=2, date="06/18/95", item_class=20, store=250
dept=2, date="06/25/95", item_class=20, store=250

This results in 4 probes per dept for 100 departments, for a total of 400 accesses, each of them requiring a probe to determine the next date value to retrieve. Extra probes are not required to retrieve values for the column dept, other than to get the starting value because this is a dense column. Dense columns do not need to be probed to determine their next value. (see Sparse vs Dense later in this paper.)

IN Lists

IN lists are essentially IN predicates specified for a key column. The predicate could also be of the form item_class=20 OR item_class=35 OR item_class=50 ... Such a predicate is also considered an IN list. So let us see how MDAM processes IN lists using the following query as an example:

```
SELECT date, item_class, store, sum(total_sales)
   FROM sales
  WHERE
          date between "06/01/95" and "06/30/95"
      and item_class IN (20, 35, 50)
      and store IN (200, 250)
GROUP BY dept, date, item_class, store;
```

Key columns dept and date are handled as discussed before. However, instead of using just the predicates item_class=20 and store=250 with each access, multiple accesses are done for each value provided in the IN list for these columns. So the following access in the previous example:

dept=1, date="06/04/95", item_class=20, store=250

is done for each value of item_class. For each value of item_class, each value of store results in an access as well. so the following accesses are done for the first values of dept and date:

dept=1, date="06/04/95", item_class=20, store=200
dept=1, date="06/04/95", item_class=20, store=250
dept=1, date="06/04/95", item_class=35, store=200
dept=1, date="06/04/95", item_class=35, store=250
dept=1, date="06/04/95", item_class=50, store=200
dept=1, date="06/04/95", item_class=50, store=250

Since the values for item_class and store have been provided in the query, no extra accesses are required against the table to determine their next values.

During all of this MDAM preserves the original order of the index, making the computation of aggregates very efficient. Materialization of a hashed or sorted intermediate is not required for forming groups.

For the above query MDAM would perform 2400 accesses. If Sales is a small table the SQL optimizer might decide to just read the entire table. When there are thousands or millions of rows qualifying for each access, however, MDAM may prove to be very efficient considering the rows it does not have to access. The rows resulting from the 2400 accesses may be a small subset of the entire table and may result in the query being executed in minutes instead of the hours it might take to do a full table scan. Note, that since the rows to be read for each access are clustered, MDAM can read these at very high scan rates using very efficient bulk I/O, pre-fetch, and virtual buffering capabilities.

Tables on Tandem systems are partitioned on a part or all of the primary or clustering key columns. Typically, DSS queries involve predicates on the key columns, with one or more of the key columns having range or In list predicates.

If the same query uses parallel execution, the 2400 access may not seem that confounding. If the sales table

contains 100 GB spread across 50 partitions, and is partitioned by dept, each partition is accessed by an executor server process in parallel. Therefore the number of accesses per process is now 48 (2400/50). Since each partition contains 2GB, and we are accessing only a small part of each partition, this query may actually be executed in seconds.

"NOT =" Predicates

Most DBMSs would not consider "NOT =" predicates to be very useful for keyed access. Predicates of the form NOT IN (3, 5, 8) which translates to NOT = 3 AND NOT = 5 AND NOT = 8 also fall into this category. MDAM can use such predicates for keyed access as well. Their efficiency depends on the selectivity of the column or the number of unique values in the table for the column. Consider a predicate of the form:

```
WHERE dept NOT IN (3,5,8)
```

Now, if there are 100 departments (as we assumed before) use of this as a key would not have much benefit because it eliminates only 3% of the depts. However, if there are only 10 departments, the impact on the execution time of using this predicate is a reduction of about 30%. The lower the number of unique values for the column and the higher the number of values in such a NOT list specified by a query the more the benefit from MDAM. MDAM uses the "NOT =" predicate by transforming it into a set of ORed predicates. So dept NOT=3 is transformed into dept < 3 OR dept > 3. For the query specified above the predicate is transformed to:

```
WHERE (dept<3 OR dept>3)
    and (dept<5 OR dept>5)
    and (dept<8 OR dept>8)
```

MDAM will access departments 1, 2, 4, 6, 7, 9 and 10. Not doing the access for departments 3, 4 and 8 can reduce the time needed to complete the query versus making a full table scan.

Multi-Valued Predicates

Multi-valued predicates (or in SQL-92 terminology "Row Value Constructors") can also be used by MDAM. A predicate of the form (dept, date, item_class ,store) = (10, "06/04/95", 20, 250) is considered the equivalent of equality predicates on each of the columns in the following form:

```
dept = 10
and date = "06/04/95"
and item_class = 20
and store = 250
```

Such a multi-value predicate is processed as discussed in earlier sections. However, when a range is expressed in a multi-value predicate (using one of the operators >, >=, <, <=) the transformation is quite different. A multi-value predicate of the form (dept, date, item_class) > (10, "06/01/95", 20) gets transformed to:

```
    (dept=10 and date="06/01/95" and item_class>20)
OR (dept=10 and date>"06/01/95")
OR (dept>10)
```

These predicates can be used by MDAM as will be discussed under *General OR Optimization*. Since the multi-value predicate has been converted to single-value predicates, they now can be combined with other single-valued predicates.

General OR Optimization

One of the most powerful aspects of MDAM is that it supports predicates with any combination of ORs and ANDs. This is accomplished by associating the predicates with different predicate sets in a variant of disjunctive normal form. IN lists are treated as a group. Therefore, the resulting predicate sets are not truly in disjunctive normal form. We will call each predicate set a disjunct. So let us take the following complex expression as an example:

```
    ((item_class=10 and date between "06/04/95" and
     06/25/95) OR dept IN (2, 4, 5))
and
    ((dept=4 and item_class=5) OR
     (item_class IN (5,10) and (date="06/04/95" OR
                                dept=2)))
```

The above expression will have its predicates associated with the following disjuncts:

```
    (dept=4 and date between "06/04/95" and "06/25/95"
     and item_class=10 and item_class=5)
OR (date between "06/04/95" and "06/25/95" and
    date="06/04/95" and item_class=10 and
    item_class IN (5,10))
OR (dept=2 and date between "06/04/95" and "06/25/95"
    and item_class=10 and item_class IN (5,10))
OR (dept IN (2,4,5) and dept=4 and item_class=5)
OR (dept IN (2,4,5) and date="06/04/95"
```

and item_class IN (5,10)
OR (dept IN(2,4,5) and dept=2 and item_class IN (5,10))

As you can see, the above disjuncts contain IN lists, which are actually OR expressions. So the expression which is finally processed is not truly in disjunctive normal form. MDAM essentially retrieves a UNION of the results of each disjunct to satisfy the query.

Why did we take such a complex expression as an example -- it does not look like one any human would code. When queries are generated by tools and/or queries are made against views with predicates and/or parameters or host variables are used complex expressions that contain many redundant predicates can be generated. We cannot always ask the user to simplify the query. The main reason though is to demonstrate how MDAM can handle even complex expressions such as the one above.

The next two sections will show how MDAM eliminates redundant predicates and does duplicate elimination before any data is read.

Elimination of Redundant Predicates

We will put the previous example of disjuncts into tabular form in table 1 to make it more understandable:

Disjunct	Dept	Date	Item_Class
1	=4	>="06/04/95" & <="06/25/95"	=10 & =5
2		>="06/04/95" & <="06/25/95" & ="06/04/95"	=10 & (=5 or =10)
3	=2	>= "06/04/95" & <= "06/25/95"	=10 & (=5 or =10)
4	(=2 or =4 or =5) & =4		=5
5	(=2 or =4 or =5)	="06/04/95"	=5 or =10
6	(=2 or =4 or =5) & =2		=5 or =10

Table 1

After values are assigned to parameters and host variables, MDAM resolves expressions and then eliminates any redundant predicates.

MDAM eliminates the redundant predicates in each of the disjuncts. In the first disjunct for item_class it finds two conflicting predicates. Item_class cannot be 10 and 5 at the same time. Therefore, this disjunct will not qualify any rows and will result in an empty set. So MDAM eliminates that disjunct altogether

For the second disjunct it picks the predicate date="06/04/95" over the date range and Item_class=10 since a row with Item_class=5 will not qualify. It also finds that there is no predicate specified for dept and it assumes a range predicate on the column of the form >=MIN_VALUE and <=MAX_VALUE (>=lo & <=hi are used in the following examples to mean the same thing).

After processing each disjunct similarly, it computes the following disjuncts as shown in table 2:

Disjunct	Dept	Date	Item_Class
2	>=lo & <=hi	="06/04/95"	=10
3	=2	>= "06/04/95" & <= "06/25/95"	=10
4	=4	>=lo & <=hi	=5
5	(=2 or =4 or =5)	="06/04/95"	=5 or =10
6	=2		=5 or =10

Table 2

Note that the first disjunct no longer appears in the list of disjuncts. Since item_class cannot be both = 5 and =10 it is completely eliminated.

Duplicate Elimination

MDAM removes duplicates before reading the data, so it does not have to do any post read operations to accomplish duplicate elimination (a common problem with OR optimization).

MDAM combines overlapping ranges among the disjuncts and separates the disjuncts into non-

overlapping accesses. So the disjuncts shown in table 2 are transformed into the following set of retrievals:

Retrieval	Dept	Date	Item_Class
1	<2	="06/04/95"	=10
2	=2	<"06/04/95"	=5 or =10
3	=2	>="06/04/95" & <="06/25/95"	=10
4	=2	>"06/25/95"	=5 or =10
5	>2 & <4	="06/04/95"	=10
6	=4	<"06/04/95"	=5
7	=4	="06/04/95"	=5 or =10
8	=4	>"06/04/95"	=5
9	=5	="06/04/95"	=5
10	=5	="06/04/95"	=10
11	>5	="06/04/95"	=10

Table 3

This set of retrievals (shown in table 3), cannot return duplicate rows. It eliminates the overhead of reading duplicate rows, which would have been incurred by the query had it used the disjuncts in the form shown in table 2. Also, then there is no overhead incurred to remove the duplicates after reading.

Maintenance of Index Order

In the process of creating non-overlapping disjuncts MDAM orders the retrievals, as you can see in the table, in the order of the index being accessed. The order may be ascending or descending. That is, MDAM maintains the index order even if it were reading the index backwards to satisfy the ordering requirements for the query to avoid a sort.

Sparse versus Dense

When range predicates exist for leading key columns (or there are no predicates available for these key columns) MDAM must go through the index and locate each value in the range. It does this in one of two possible ways, depending on whether a column is sparse or dense.

A dense key column is one which has all (or almost all) of the possible values for the column. If a column has 100 unique values and the column ranges from 0 to 99, then this column is dense.

When a dense column is recognized then MDAM only has to add 1 to the previous value, and look for any values that satisfy the predicates for the remaining key columns. This was demonstrated by the example under the section Missing Predicates. This method will adapt to the actual values found in the database, and switch to the sparse method if it doesn't find that data is actually dense.

A column is recognized as sparse if it is missing at least 10 percent of its possible values. MDAM treats a sparse column differently than a dense one. Using the present key value it probes to find the next value in the index for this column. This value is inserted as a key value for that column and the required data is retrieved. An example of a sparse key column was the date column that had four non-consecutive values for the range specified. Access based on the sparse method was demonstrated by the example under the section Intervening Range Predicates. An example of a dense key column was the dept column with 100 values from 1 to 100.

Benefits

As mentioned in the introduction and demonstrated in the paper, the benefits of MDAM can be substantial for a multitude of queries. this is specially the case for large (GB to TB size) databases designed to be accessed by multidimensional queries.

The following section shows how MDAM on B-Trees is an improvement over hashed based databases.

Hashing vs. B-Tree

Even after demonstrating that B-Trees can after all be used efficiently for multidimensional access in large DSS, some may suggest that a hashed file organization is still better than B-Tree structures for large scale DSS. B-Trees have many advantages over hashed structures:

- B_Trees can handle inserts and set updates better than hashed structures can.
- With many hashed structures all tables have to be spread equally over available disk storage. This creates a problem in a constantly growing environment where frequent and massive reorganization may be necessary to re-distribute data across a larger number of disks. Most of the implementations today render the database unavailable during such re-organizations. With range partitioning of B-Tree structures there is a lot of flexibility in how data is spread across available

disks. Incremental growth in the database can be accommodated by various partition management utilities that allow partition splits, merges additions, deletions, movement, and changes in partition boundaries. All this re-organization can be done on-line while the system is available 24 x 7.

- The chosen index better be a good one to yield a balanced hashed organization. Otherwise, there is a problem in managing data distribution evenly. With B-Trees, partition boundaries can be specified based on the data distribution across key values to achieve balanced distribution.

- Queries needing sets of data that can be retrieved contiguously benefit from a B-Tree organization. MDAM extends this benefit to a broad class of queries. Bulk I/O, pre-fetch, and buffering capabilities can be used to take advantage of accessing clustered data with fewer I/Os.

- All B-Tree queries do not have to be executed in parallel if they can be satisfied by accessing a single partition. Sometimes only a few partitions need be accessed to satisfy the query instead of the entire table. Between the clustered bulk I/O benefit mentioned above and the reduction in partitions accessed, less CPU and disk resources are necessary to satisfy the query workload on the system -- a price/performance advantage.

- Hashed organizations have the problem that multidimensional queries with predicates missing, or range predicates on leading or intervening key columns, will result in full table scans which are resource intensive.

Hashed structures do have a perceived benefit over B-Trees in the automatic balancing of partitions across disks, as long as a good index is chosen. They are also perceived as promoting parallelism. (Note that one of the bullets above discusses the benefit of not having to be parallel all the time.) However MDAM and B-Trees can help provide the same benefits, but do it more efficiently.

There are two types of database organization on B-Trees that can provide this type of benefit. Both allow a leftmost column added to the primary index.

- The leftmost column added to the primary index is used to "hash" distribute the rows of a table. This can be an existing column of low unique entry count or an artificial column created by hashing columns of the table. The table can be partitioned on this column.

- The leftmost column can be assigned a value in a round robin fashion so that each row falls into a different partition. This will ensure even distribution of the data across partitions.

So we can achieve a balanced "hashed" distribution of rows across B-Tree partitions for ease of partition management. This is an improvement over the perceived advantage of hashed structures over B-Trees. So how do we access this table with a leftmost column that will probably have no predicates for it? MDAM will treat this column for all queries as a column with missing predicates and will deal with it as demonstrated above. The key then is to have a low number of unique values per partition for the leftmost column so that the number of probes to find the next values for it is small. At the same time, there should be a sufficient number of values for the column per partition, so that the partition can be split at a later date, to accommodate a growing database. With on-line utilities to re-organize the database and manage the partitions, MDAM offers you the best of both worlds using B-tree structures.

Performance

The following performance figures shown in table 4 compare MDAM performance to reading the entire table and reading through an alternate index. The tests were run with a Wisconsin table of 75,000 records occupying 18.15 MB. The table was created with the key columns of four, ten, twenty and onepct. The unique entry counts for these columns are 4, 10, 20 and 750. The following query was executed:

SELECT sum(unique1) FROM k75tup WHERE <pred>;

<pred> was varied to be (ten=1, twenty=1 and onepct=1). Secondary indexes were created for the columns ten, twenty and onepct.

<pred>	Table Scan Time	Secondary Index Time	MDAM Accesses	MDAM Time
ten=1	9 secs	38.0 secs	4	4.3 secs
twenty=1	9 secs	26.0 secs	40	3.4 secs
onepct=1	9 secs	2.5 secs	800	1.6 secs

Table 4

This simple experiment scales up to large databases. It shows that MDAM can significantly extend the use of B-Trees and give excellent performance.

Summary

The MDAM access method has been shown to extend the usefulness of the already important B-Tree index organization. Multidimensional access using B-Trees allows efficient clustered access to databases. Support of columns with no predicates allows users to extend the types of database design that can be used with B-Trees, allowing extremely efficient access and maintenance. The general processing of MDAM, allowing keyed access of extremely complex queries involving ranges, IN lists and arbitrary ORs allows queries to be expressed in an arbitrary manner, and still be executed efficiently, as MDAM will retrieve a record only once and still maintain index order. We have also show the multiple benefits of B-Trees for large databases using MDAM.

References

Bayer, R. and Schkolnick, M. "Concurrency of Operations on B-Trees" Acta. Inf. 9 1 (1977).

Bayer, R and Unterauer, K. "Prefix B-trees" ACM-TODS, Vol. 2, No. 1, March 1977.

Beckmann, N. and Kriegel, H. P. and Schneider, R. and Seeger, B "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles". ACM-SIGMOD 1990.

Bentley J. L. "Multidimensinal Binary Search Trees Used for Associative Searching" Comm. ACM, Vol. 18, No 9, 1975.

Chang J. M. and Fu K. S. "A Dynamic Clustering Technique for Physical Database Design" ACM-SIGMOD 1980.

Comer, D. "The ubiquitous B-tree" ACM Computing Surveys Vol. 11, No. 2, 1979.

Guttman, A. "R-trees: a dynamic index structure for spatial searching" Proceedings ACM-SIGMOD Conf., 1984.

Held, G and Stonebraker, M "B-Trees Re-examined" Electronics Research Laboratory, University of California, 1975.

Knuth, D. E. "The Art of Computer Programming" Vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

Lomet, D. B. and Salzberg, B. "The hB-tree: a Robust Multi-Attribute Indexing Method" ACM Trans. on Database Systems, Vol. 15, No 4. 1989.

Nievergelt, J and Hinterberger H. "The Grid File: An Adaptable, Symmetric Multikey File Structure" Trends in Information Processing Systems, Proc 3rd ECI Conference 1981.

Robinson, J. T. "The K-D-B-Tree: A Search Structure for Large Multi-dimensional Dynamic Indexes" Proc. ACM SIGMOD Conf. 1981.

Rothnie, J. B. and Lozano, T. "Attribute Based File Organization in a Paged Memory Environment" comm ACM, Vol. 17, No 2, Nov 1974.

Seeger, B. and Kriegel, H. P. "The Buddy-tree: an Efficient and Robust Access Method for Spatial Data Base Systems" Proc. 16th Int. Conf. on Very Large Data Bases, 1990.

Sellis, T. and Rossopoulos, N. and Faloutsos, C. "The R+ Tree: a Dynamic Index for Multidimensional Objects" Proc. 13the Int. Conf. on Very Large Data Bases, 1987.