Multi-Dimensional Clustering: A New Data Layout Scheme in DB2

Sriram Padmanabhan
Bishwaranjan Bhattacharjee
Tim Malkemus
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, New York, USA
{srp,bhatta,malkemus}@us.ibm.com

Leslie Cranston
Matthew Huras
IBM Toronto Laboratory
Markham, Ontario, Canada
{lesliew,huras}@ca.ibm.com

ABSTRACT

We describe the design and implementation of a new data layout scheme, called multi-dimensional clustering, in DB2 Universal Database Version 8. Many applications, e.g., OLAP and data warehousing, process a table or tables in a database using a multi-dimensional access paradigm. Currently, most database systems can only support organization of a table using a primary clustering index. Secondary indexes are created to access the tables when the primary key index is not applicable. Unfortunately, secondary indexes perform many random I/O accesses against the table for a simple operation such as a range query. Our work in multi-dimensional clustering addresses this important deficiency in database systems. Multi-Dimensional Clustering is based on the definition of one or more orthogonal clustering attributes (or expressions) of a table. The table is organized physically by associating records with similar values for the dimension attributes in a cluster. We describe novel techniques for maintaining this physical layout efficiently and methods of processing database operations that provide significant performance improvements. We show results from experiments using a star-schema database to validate our claims of performance with minimal overhead.

1. INTRODUCTION

We report on the design and implementation of a new data layout scheme in DB2 Universal Database Version 8 called *Multi-Dimensional Clustering*(MDC) [1]. Logically, every database schema defines a large multidimensional cube containing the active domains of its attributes. Many applications, OLAP and data warehousing in particular, usually consider several of these attributes as dimensions for processing and maintenance. For example, a retail warehouse contains dimension attributes based on time, region, customer, product, forecast, and others [6]. Applications and users access data from this database using subsets of dimensions. These access patterns can be considered as selecting regions from the large multidimensional cube.

Most commercial database systems now include primary clustering indexes in their repertoire [3, 5]. In this scheme, an index composed of one or more keyparts is identified as the basis for data clustering. All records are organized based on their attribute values for these index keyparts. Two records are placed physically close to each other if the attributes defining the clustering index keyparts have similar values. Note that the clustering index typically contains one entry for each record. In our terminology, these clustering indexes provide unidimensional physical clustering of data. Records are clustered in the order of the index keyparts. When a query contains predicates on attributes in the index *but* not conforming to the primary order, it requires skip sequential accesses of the data. These skip sequential accesses can generate significant I/O seek times as the scan jumps between pages on disk. In many cases, it may be cheaper to perform such accesses using a different secondary index or using a table scan method. Several relational database systems also support a coarser grain range-partitioning technology [5, 4]. Most of these mechanisms are also fairly unidimensional and provide rigid syntax that introduces many manageablility issues if they are to be extended to multiple orthogonal dimensions.

OLAP systems recognized this problem of relational databases and provide a solution that physically organizes the data in some multidimensional cube form [2]. A request to find regions in this cube is easily satisfied by OLAP systems. However, these systems usually suffer from sparsity and scalability problems. OLAP systems organize data by maintaining a physical cell for each possible dimension combination. When data elements are sparse, this results in many empty base cells and introduces significant space wastage. On the other hand, scalability and ability to address sparsity are strengths of relational databases. It is to be noted that there are many reasearch publications dealing with spatial clustering. We do not claim to invent a radically different idea for clustering. Instead, we extend basic research ideas to practically surmount the problem of providing efficient clustering using multiple attributes and describe the various processing techniques that are implemented to take advantage of this scheme.

We would like a scheme that maintains the robustness

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

and scalability provided by relational database systems but is able to provide efficiency for multidimensional data access. This is the main design goal behind MDC. An MDC table is defined to include one or more clustering dimensions. Unlike the unidimensional primary clustering index, we support a physical layout that mimics a multi-dimensional cube by associating a physical region for each unique combination of dimension attribute values. Records containing the same unique values for dimension attributes are clustered in one or more regions called *blocks*. These blocks are the units of addressability for our clusters. We provide a higher granularity indexing scheme, called *block indexes*, to address these blocks. Block indexes are quite compact since they identify several records using one entry. In all other aspects, block indexes resemble regular B-tree indexes. We also associate a data structure called the *Block Map* to maintain information about the state of blocks in a table. We will describe the organization of the table and these data structures in more detail. New techniques for query processing and maintenance operations using these data strucutures are overviewed. We present experimental results showing the effectiveness of this data layout scheme.

The rest of this paper is organized as follows. Section 2 describes the overview of our MDC scheme describing the data layout as well as the auxiliary data structures. Section 3 describes the basic aspects of choosing dimensions and organizing an MDC table. Section 4 describes the new query processing techniques that are made possible using the MDC data layout. Section 5 describes some of the maintenance related aspects of our scheme. Section 6 describes results from experiments comparing a table using the MDC layout against a table clustered by a primary clustering index. Section 7 provides a conclusion.

2. MDC OVERVIEW

This section provides a brief overview of the main features of MDC. Using this feature, a DB2 table may be created by specifying one or more keys as dimensions along which to cluster the table's data. We have created a new clause called `ORGANIZE BY DIMENSIONS` for this purpose. For example, the following DDL describes a Sales table organized by `storeId`, `year(orderDate)`, and `itemId` attributes as dimensions.

```
CREATE TABLE Sales(
int   storeId,
date  orderDate,
int   region,
int   itemId,
float price
int   yearOd generated always as year(orderDate))
ORGANIZE BY DIMENSIONS (region, yearOd, itemId)
```

Each of these dimensions may consist of one or more columns, similar to index keys. In fact, a 'dimension block index' will be automatically created for each of the dimensions specified and will be used to quickly and efficiently access data. A composite block index will also be created automatically if necessary, containing all dimension key columns, and will be used to maintain the clustering of data over insert and update activity.

Every unique combination of dimension values forms a logical 'cell', which is physically organized as blocks of pages, where a block is a set of consecutive pages on disk. The set of blocks that contain pages with data having a certain key value of one of the dimension block indexes is called a 'slice'. Every page of the table is part of exactly one block, and all blocks of the table consist of the same number of pages, viz., the blocksize. In DB2, we have associated the block size with the extent size of the tablespace so that block boundaries line up with extent boundaries.

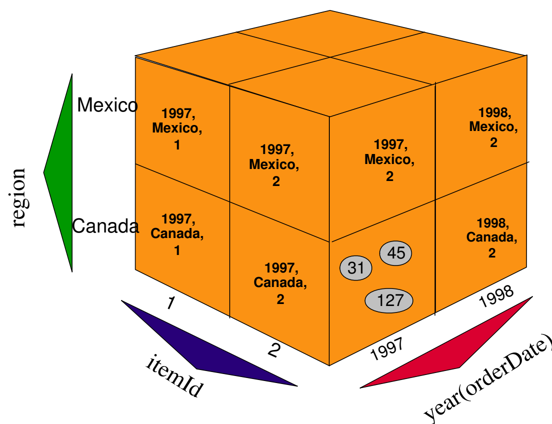

Figure 1: Logical view of physical layout of an MDC table

Figure 1 illustrates these concepts. This MDC table is clustered along the dimensions `year(orderDate)`[1], `region`, and `itemId`. The figure shows a simple logical cube with only two values for each dimension attribute. In reality, dimension attributes can easily extend to large numbers of values without requiring any administration. Logical cells are represented by the sub-cubes in the figure. Records in the table are stored in blocks, which contain an extent's worth of consecutive pages on disk. In the diagram, a block is represented by a shaded oval, and is numbered according to the logical order of allocated extents in the table. We only show a few blocks of data for the cell identified by the dimension values <1997,Canada,2>. A column or row in the grid represents a slice for a particular dimension. For example, all records containing the value 'Canada' in the `region` dimension are found in the blocks contained in the slice defined by the 'Canada' column in the cube. In fact, each block in this slice only contains records having 'Canada' in the `region` field.

Block Indexes

In our example, a dimension block index is created on each of the `year(orderDate)`, `region`, and `itemId` attributes. Each dimension block index is structured in the same manner as a traditional B-tree index except that at the leaf level the keys point to a *block Identifier* (BID) instead of a record identifier (RID). Since each block contains potentially many pages of records, these block indexes are much smaller than RID indexes and need only be updated whenever a new block is added to a cell or existing blocks are emptied and removed from a cell. A slice, or the set of blocks containing pages

[1]Dimensions can be created using Rollup functions as explained in Section 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
U	U	F	F	U	U	U	L	U	F	F	U	C	F	F	U	U	F	L	...

Figure 2: Block Map entries

with all records having a particular key value in a dimension, will be represented in the associated dimension block index by a BID list for that key value. The following diagram illustrates slices of blocks for specific values of `region` and `itemId` dimensions, respectively.

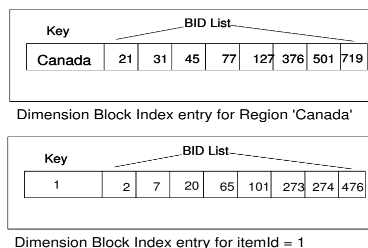

Figure 3: Block Index Key entries

In the example above, to find the slice containing all records with 'Canada' for the `region` dimension, we would look up this key value in the `region` dimension block index and find a key as shown in Figure 3(a). This key points to the exact set of BIDs for the particular value.

Block Map

A Block Map is also associated with the table. This map records the state of each block belonging to the table. A block may be in a number of states including **In Use**, **Free**, **Loaded**, **requiring Constraint enforcement**, etc. The states of the block are used by the data management layer in order to determine various processing options. Figure 2 shows an example blockmap for a table. Element 0 in the block map represents block 0 in the MDC table diagram. Its availability status is 'U', indicating that it is in use. However, it is a special block and does not contain any user records. Blocks 2,3,9,10,13, 14, and 17 are not being used in the table and are considered 'F' or free in the block map. Blocks 7 and 18 have recently been loaded into the table. Block 12 was previously loaded and requires constraint checking to be performed on it.

3. DESIGN CONSIDERATIONS

A crucial aspect of MDC is to choose the right set of dimensions for clustering a table as well as their proper granularities and table blocksize to minimize the space utilization. If the dimensions and blocksizes are chosen appropriately, then clustering benefits will translate into significant performance and maintenance advantages. On the other hand, if chosen incorrectly, performance may degrade and space utilization could be significantly worse. There are a number of tuning knobs that can be exploited to organize the table. These include

- Varying the number of dimensions.
- Varying the granularity of one or more dimensions.
- Varying the blocksize (extentsize) and pagesize of the tablespace containing the table.

One or more of these techniques can be used jointly to identify the best organization of the table.

Identify candidate dimension attributes

The first step is to identify candidate dimension attributes for a table. The main criterion is the need for clustering based on the workload. The sample workload should be examined for one or more of the following types of clauses or conditions that are likely candidates for clustering. These include

- Range, equality, or IN-list types of predicates on attributes. For example, `year > 1999`.
- Roll-in or Rollout of data. For example, Load data for `itemId=100` and delete data for `year=1999`.
- Group By clause attributes. For example, Group By year.
- Join clauses especially in a star-schema. In particular, if the join condition is 1:N (usually because of primary key-foreign key joins), then the column with N duplicates can be a good candidate for dimensions.
- Order By clauses.
- Combinations of the above.

Columns that are frequently updated (by changing values) are NOT good candidates.

Using Rollup hierarchies

Given a candidate dimension, it is possible that it leads to relatively few duplicates for each unique combination. In such cases, we can rollup along hierarchies and improve the number of duplicates. For instance, suppose date is a candidate dimension but each date value only has roughly 10 records in the table. At this level of granularity, each block is likely to waste a lot of space. In this case, we can recognize that dates can be rolled upto unique yearAndMonth values. If so, each yearAndMonth will contain roughly 300 records which should be sufficient for utilizing blocks.

4. QUERY PROCESSING

One of the goals of MDC is to facilitate efficient query processing. In particular, new processing techniques are introduced to take advantage of the block oriented clustering and indexing techniques. We briefly overview block based processing of index scans, index ANDing, and index Oring schemes below. There are several other processing enhancements which we are unable to describe due to space constraints.

Block Index Scan

Consider a query such as the following one. *What is the Aggregate sales of itemId=1000 over all dates, and regions?* This query can be efficiently processed by the block index

scan operation that is newly introduced for MDC tables. It proceeds in two steps: (i) scan the block index to find a Block ID that satisfies the query predicate. (ii) Process all the records in this block. This might involve additional predicates. The Block Scan operation is most effective when all or most of a block or entire sets of blocks need to be processed for a given query. This will involve just one I/O per block as it is stored as an extent on disk and can be read into the bufferpool as a unit. If the predicates for the query are only on dimension values, we need only apply the predicates on one record in the block due to the inherent clustering property of the block. These predicates are called *block predicates*. If other predicates are present, we need only check these on the remaining records in the block.

The example query above is very likely to involve access to a whole stripe of blocks. Thus, the block scan operation is likely to be the most efficient method of processing this query.

Block Index Anding

Now, if one wished to find all records having `itemId = 1000` and `region = 'Mexico'`, we can include an Index Anding plan for consideration. For MDC tables, we have implemented a block Index Anding technique that can combine one or more block and RID indexes. The block Index Anding employs a bit-map based intersection approach if sufficient memory is available. Since the block indexes are smaller than the RID based ones, the processing time for Index Anding is also significantly less. The intersecting list of blocks are accessed efficiently using block based I/O operations. Overall, the operation is extremely efficient and should be significantly faster than existing alternatives prior to this technique. We are also able to combine Block and RID indexes efficiently.

Block Index ORing

Likewise, a block based Index Oring operation has also been implemented. Suppose a query includes the condition: `itemId = 1000 or itemId = 3000`. The block index on itemId can be scanned for each value and the aggregated list of blocks can be obtained by an Oring operation. The Oring operation will eliminate duplicate blockIDs for conditions such as `region = 'Mexico' OR year > 1999`.

Deriving predicates when using rollup hierarchies

One very important aspect of our MDC implementation is the use of rolled up hierarchy level as a dimension attribute for generating denser blocks of data. However, this feature would be difficult to use if the queries being submitted by users and applications need to be modified. For example, suppose we decide to rollup date to year using the year() function for use as a dimension. Suppose the original query includes a predicate of the form `date > '1999-01-01'`. Realizing the presence of the year(date) dimension attribute, DB2's query compiler will automatically derive a predicate of the form `year >= 1999` and include it for optimization. This enables the query optimizer to choose the block index on year for processing this query.

Impact on exisiting techniques

It is natural to ask whether the new MDC feature has an adverse impact or disables some existing features of DB2 for normal tables. We are pleased to report that all existing features such as secondary RID indexes, constraints, triggers, defining materialized views, query processing options, etc. are available for MDC tables. Hence, MDC tables behave just like normal tables except for its enhanced clustering and processing aspects.

5. MAINTENANCE OF MDC TABLES

In this section, we briefly describe new techniques we have implemented for maintaining MDC tables. We describe schemes for loading, inserting, updating, and deleting these tables. It is to be noted that sophisticated clustering normally implies additional overhead for maintenance. In fact, maintenace overhead has been a crucial inhibitor for implementing more sophisticated clustering or indexing ideas in database systems. Below, we show briefly that the MDC scheme lends itself to relatively efficient maintenance. Most utilities such as Reorg, Backup, Restore, and Runstats have been modified appropriately for use with MDC tables.

Load

The load utility must employ an efficient scheme to insert large data sets into a table. Load into an MDC table must organize the input data along dimension values. Since we cannot rely on the input data to be sorted, we have designed and implemented a bucketization algorithm in order to perform this operation efficiently.The load utility creates bins for the data records based on the dimension attribute values. For example, a bin is created for the logical cell corresponding to `<year=1999, region='Canada', itemId=1000>`. All records with similar values of the dimension attributes will be assigned to this bin. Physically, each bin is represented minimally by a page in a block. Recently processed bins are maintained in memory and written to disk when they become full or if there is a need to bring other bins into memory.

Insert

It is absolutely critical for clustering of data to be maintained during insert operations. Consider Figure 1. Suppose we are inserting a record with dimension values `<2000, 'Mexico',1000>`. We would first need to identify the appropriate block for this new record by using the composite block index. We would look up the key value in the composite block index and find that there are two blocks, say 4 and 8, with this key value. These blocks contain records having these dimension key values. We therefore insert the new record in one of these blocks if there is space in any of their pages. If there is no space in any pages in these blocks, we either allocate a new block for the table or use a previously emptied block in the table. The block map is searched to find a free block. Note that in this example, block 2 (figure 2) is currently not in use by the table. We insert the record on a page in that block, and assign this block to this cell by adding its BID to the composite block index and to each of the dimension block indexes. Suppose that there are no more free blocks in the table. Then, a new block is allocated in the table and is used to insert the row. The indexes are updated in that case as well.

If a record with a new dimension values is inserted, then a new block or free block must be allocated. The new key values will be added to the dimension and composite block indexes.

When performing insert, we have to pay special attention to the insert of the first record in a block as well as the first record to a new page in a block. We use a page bit map per block to maintain the state of the pages in the block. The bit in the page bit map is set when the first record is inserted into the page. This bit map enables us to track the occupancy of pages in a block and helps us maintain the state of the block in the presence of inserts and delete operations.

Delete

The delete operation can free one or more records in a table. This is also the case for MDC tables. However, special attention is given to the state of pages in a block and the entire block as well. If we delete the last record in a page, then the page bit map is updated to clear the bit associated with the particular page. When all pages in a block are empty, this page bit map is fully cleared and this will indicate that the block can be marked free in the block map. This free block can be reused by future insert or load operations. When a block is freed, we must also update all the dimension indexes and remove the BID from the particular key(s) corresponding to the dimension attributes for the block.

Update

Update operations to non-clustering attributes are similar to regular tables. If the update results in the creation of an overflow, special care is taken to create the overflow record in a block belonging to the same cell. When there is an update to one or more clustering attributes, then the behavior is similar to a delete followed by insert into a new cell.

	storeId	salesDate	itemId
MDC Index Pages	71	72	222,086
Non-MDC Index Pages	222,054	222,054	222,086
MDC Index Levels	2	2	4
Non-MDC Index Levels	4	4	4

Table 1: Comparison of index size and levels

6. EXPERIMENTAL RESULTS

We have conducted a set of atomic experiments using a star-schema database called *POPS*. This schema includes a main fact table called Sales of size 36 GB and a number of dimension tables on dates, stores, products, customers. We compared the performance of the queries on a MDC Sales table organized by two dimensions (salesDate, storeId) against a regular table with primary clustering index (salesDate, storeId) and secondary indexes on other dimension keys. Both tables were loaded in sorted order of their clustering keys.

Table 1 compares the sizes and number of levels of the indexes on the MDC and non-MDC versions of the table. Note that the sizes of the block indexes on the MDC table are significantly smaller (only about 3%) than the equivalent RID indexes. Also, note that a RID index on the itemId column uses the same space for both MDC and normal tables. The number of pages of the MDC table is 689, 264 and is only slightly larger than the non-MDC table which has 681, 903 pages.

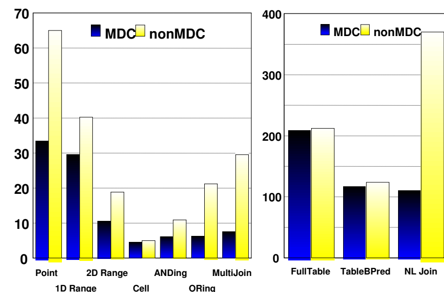

Figure 4: Performance comparison of MDC and normal tables

Figure 4 summarizes the comparison of queries executing against these two tables. The queries include table scans with and without block predicates, block index scans (point and range), Index Anding and Oring plans, and join queries. Note that the performance of the MDC table is usually better than the performance of the non-MDC table for all queries. The speedup improvements in these experiments ranged from 4% for table scans to 75% or more for Index Anding operations. Beta customers trying this feature are also obtaining similar large speedups on their databases.

7. CONCLUSION

Multi-Dimensional Clustering is a new data layout technique in DB2 Universal Database version 8. It provides an efficient block oriented clustering mechanism and associated new processing techniques for obtaining efficiency and better manageability of data. We believe that Multi-Dimensional Clustering is an effective data organization technique for many modern database applications. We described the design considerations, new query processing, and maintenance operations for this new scheme. Finally, we showed that the Multi-Dimensional clustering scheme is efficient for a wide range of atomic queries.

8. REFERENCES

[1] Method and System for Multi-Dimensional Clustering in a Relational Database System, 2002. Patent Filed, IBM Corp.

[2] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[3] http://www.ibm.com/software/data/db2/library.

[4] http://www.informix.com.

[5] http://www.oracle.com.

[6] R.E. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley, 1996.