Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2

Bishwaranjan Bhattacharjee
Sriram Padmanabhan
Timothy Malkemus
IBM T. J. Watson Research Center
Hawthorne, NY, USA
{bhatta,srp,malkemus}@us.ibm.com

Tony Lai
Leslie Cranston
Matthew Huras
IBM Toronto Laboratories
Markham, Ontario, Canada
{tonylai,lesliew,huras}@ca.ibm.com

Abstract

We have introduced a Multi-Dimensional Clustering (MDC) physical layout scheme in DB2 version 8.0 for relational tables. 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. Each clustering key is allocated one or more blocks of physical storage with the aim of storing the multiple records belonging to the cluster in almost contiguous fashion. Block oriented indexes are created to access these blocks. In this paper, we describe novel techniques for query processing operations that provide significant performance improvements for MDC tables. Current database systems employ a repertoire of access methods including table scans, index scans, index ANDing, and index ORing. We have extended these access methods for efficiently processing the block based MDC tables. One important concept at the core of processing MDC tables is the block oriented access technique. In addition, since MDC tables can include regular record oriented indexes, we employ novel techniques to combine block and record indexes. Block oriented processing is extended to nested loop joins and star joins as well. We show results from experiments using a star-schema database to validate our claims of performance with minimal overhead.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

IBM's DB2 Universal Database version 8 for Unix, Windows, and open platforms introduces a new feature called *Multi-Dimensional Clustering* (MDC) [2]. Multi-Dimensional Clustering provides a flexible and orthogonal physical clustering organization for a relational table using one or more attributes of the table as dimensions of clustering. We have described the overview of MDC - including how the clustering is maintained over time - in [5, 1]. In this paper, we report on the query processing enhancements to DB2 for processing MDC tables efficiently.

Many applications, such as OLAP, data warehousing and spatial, process a table or tables in a database using a multi-dimensional access paradigm. In the previous work [5], we described why the prevailing techniques of clustering such as clustering indexes [2, 4] or range partitioning [2, 4, 3] do not fully address the requirements of these applications. That paper also overviewed the unique features of the MDC implementation and how it addresses these issues better than the prevailing technology.

In this paper we describe new techniques for query processing operations on MDC tables that provide significant improvements. One of the important features of MDC is the concept of using blocks of pages for clustering data. The block oriented data organization naturally establishes the requirement for various *block based* processing techniques for obtaining performance efficiencies. As a first step, we introduce *block indexes* to manage and process these blocks of data efficiently. We extend the traditional access methods in DB2 to deal with block oriented processing. A basic block fetch routine acts as a building block for all these extensions. Additionally, since an MDC table behaves just like regular tables, it can have record indexes defined on it. Hence, we need to extend operations such as Index ANDing and ORing to combine block and record indexes efficiently. We also introduce a new *block predicate* that can be applied as a data predicate in a once-per-block fashion. We describe our enhancements to the query compiler and other run-time features such as prefetching that combine to create the right efficiencies for processing MDC tables.

We have evaluated the performance of MDC tables using a variety of atomic operations and more complex queries. We report on the performance of these experiments and show that the MDC organization can provide significant performance improvements for a variety of queries and workloads.

The rest of this paper is organized as follows. Section 2 describes the overview of our MDC scheme including the data layout, the auxiliary data structures, and the basic aspects of design such as choosing dimensions. Section 3 provides an overview of the new query processing techniques that are made possible using the MDC data layout. Section 4 describes the enhancements to the various access methods in more detail. Section 5 describes the impact of MDC on joins, group by, and other operations. Section 6 describes the enhancements to the query compiler and optimizer for supporting these query processing features. Section 7 describes results from experiments comparing the MDC layout against a normal table clustered by a primary clustering index. Section 8 provides a conclusion.

2 MDC Overview

This section provides a brief overview of the main features of MDC [5]. 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,
date  shipDate,
date  receiptDate,
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

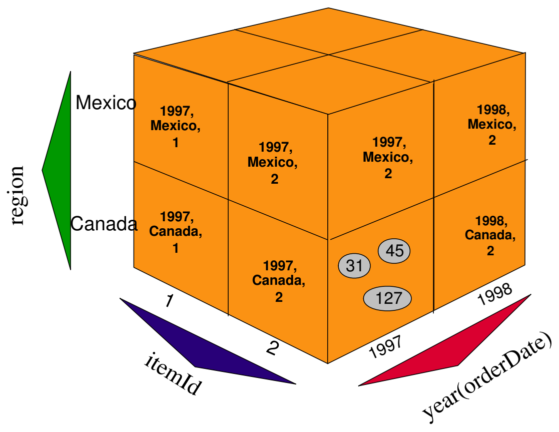

Figure 1: Logical view of physical layout of an MDC table

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 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

[1]Dimensions can be created using Rollup functions as explained in Section 6

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 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.

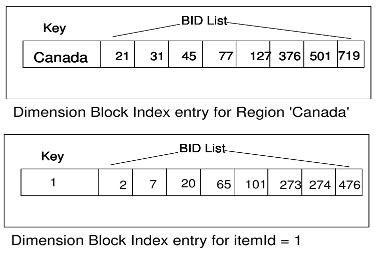
Key
BID List
Canada
21
31
45
77
127
376
501
719
Dimension Block Index entry for Region 'Canada'
Key
BID List
1
2
7
20
65
101
273
274
476
Dimension Block Index entry for itemId = 1

Figure 2: 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 2(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 3 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.

Design Considerations

A crucial aspect of MDC is to choose the right set of dimensions for clustering a table and the right blocksize parameter to minimize the space utilization [5]. If the dimensions and blocksizes are chosen appropriately, then the clustering benefits will translate into significant performance and maintenance advantages. On the other hand, if chosen incorrectly, the performance may degrade and the 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.

The first step is to identify candidate dimension attributes for a table. The main criterion is the need for clustering based on the workload. Attributes that are used in Range, equality, or IN-list predicate clauses, e.g., `orderDate > '1999-02-01'`, are potential candidates. Similarly, attributes used to load or purge batches of data, grouping columns, join columns in a star schema, and combinations of the above are potential candidates for clustering. Unique columns or columns that are frequently updated (by changing values) are NOT good candidates.

Given a candidate dimension, it is possible that it leads to relatively few duplicates for each unique combination. In such cases, we can use rollup hierarchies and improve the number of duplicates. For instance, suppose `orderDate` 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 recognize that dates can be rolled up to unique `yearAndMonth` values. Each `yearAndMonth` value will contain roughly 300 (30×10) records which may be sufficient for utilizing blocks. More details about this and other design considerations can be found in [5].

Impact on existing 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.

0	1	2	3	4	5	6	7	8	9	19	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 3: Block Map entries

3 Query Processing Overview

One of the major goals of MDC is to facilitate efficient query processing. To this end a lot of new query processing technology using block oriented techniques was developed. While doing this we needed to keep in mind that block indexes need to be combined with available record based indexes for processing too. We motivate the main query processing enhancements using examples in this section and describe these enhancements in more detail in the following sections.

Let us suppose that the Sales table described previously is organized along three dimensions, viz., `region`, `year(orderDate)`, and `itemId`. Suppose also that record indexes are created for the `shipDate`, and `receiptDate` columns. Consider the following queries:

```
Q1 :  What is the aggregate sales of
ItemId=1000 sold over a period of years
1997-1998?
Q2:  What is the aggregate count of items
ordered in 2002 and shipped in 2003?
Q3 :  What is the aggregate sales of all
items ordered in 2002 or received in 2002?
Q4 :  List the distinct set of items ordered
from stores in region Mexico in 2002.
```

The choices for processing query Q1 include the following:

Table Scan: Scan the entire table and only select the rows with ItemId=1000 and OrderDate falling in the specified period of 2 years.

Block Index Scan: Use the Block Index on ItemId to narrow down to specific set of blocks and process the blocks.

Block Index ANDing: Use the Block Index on ItemId to obtain a set of block ids which satisfy ItemId=1000, then use the Block Index on year(orderDate) to obtain another set of block ids which satisfy the orderDate clause, intersect them and process the resulting set of blocks.

For processing query Q2 we could employ the following schemes in addition to the table scan and block index scan techniques described above:

Record Index Scan: Use the `shipDate` index to identify records and process these records by applying the remaining condition.

Mixed Index ANDing: Use a Record ID based index on `shipDate` to obtain a set of record ids which satisfy 2003; use the Block Index on `year(orderDate)` to obtain a set of block ids which satisfy the orderDate clause; intersect the set of record ids with the set of block ids to obtain a set of record ids which can then be processed.

When considering Q3, apart from relation scans and block index scans, this query could also be processed as:

Index ORing: Use a Block Id based index on `year(orderDate)` to get a set of blocks which qualify for 2002; use a Record Id based index on receiptDate to get a set of Records Ids which qualify 2002, take the union with duplicate elimination of the 2 sets and process the resulting set of blocks ids and record ids.

If `receiptDate` had been a clustering column, this query could be answered by an index ORing of 2 block indexes. This could have resulted in a set of blocks which then would have to be processed.

Finally, for query Q4 we could employ an *Index Only* scheme based on the composite block index which includes all dimension columns by selecting on the region and order year and listing the `itemIds`.

Now suppose that the database contains an `Items` table. Consider the following query.
`Find aggregate sales for items in`
`'sportswear' ordered in 2002?`
Such a query requires a join between the `Items` table and the `Sales` table on the `itemId` column which can be processed as nested-loop join using the block index on `itemId`.

We will describe in detail the available options outlined above in the following sub sections. We begin with some features common to most of the access paradigms.

3.1 Block Predicates

A predicate is a condition in the query. In Q1 the conditions include one on `region` with the query asking for `Canada` and `Mexico`. Predicates are of 3 types - index, data and deferred - depending on which object they are applied and when. Data predicates are applied when the data record is accessed from disk and deferred predicates are applied at a later stage in the query. In current technology these predicates are applied on every record. With MDC we have introduced a new type of predicate called a block predicate which is applied only 1 time per block. This subsection explains the benefits of block predicates and its usage.

In a block index there is one index entry for a block of data. So any index predicate is automatically applied one time per block instead of once for each record in the block as could happen with record indexes. Predicate processing can be quite costly in terms of CPU usage and this savings translates to more efficient query processing.

Data predicates are applied on a data record object. For MDC we are able to take advantage of the fact that all records in a block will have the *same value* for the clustering attribute(s) and if some or all of the predicates deal with these attributes then it should be okay to apply them on only 1 record in the block. This is called a block predicate. For example, if we find the first record of a block has value Africa for attribute `region` then so will the rest of the records in the block. Thus the region predicate in Q1 could be applied as a block predicate.

If it turns out to be false then we skip processing the entire block thus saving on CPU cost of processing the remaining records and any remaining I/O operations on that lock. If the predicate turns out to be true then we need not apply the block predicate on the remaining records of the block while processing them. We just need to apply any remaining data predicates. In both cases, we obtain faster query execution with block predicates in comparison to current record predicates.

3.2 Block Identifiers

In DB2 a record is identified by a record identifier or RID. This RID consists of a 24 bit page number which identifies the page it belongs to and a 8 bit slot number which indicates the position of the record within a page. Within a physical partition for a table this RID is unique.

A block identifier or BID is similar to a RID in structure. While it is made up of a 24 bit page identifier and an 8 bit slot, not all the bits are relevant to identifying a block. A very important characteristic of a BID is that it is the RID of the first record of a block which is always a system record. So it is very easy to distinguish a BID from a data RID and is required for operations like mixed index ORing. The 8 bit slot number in a BID is zero from its definition above. Even from the 24 bit page number the number of bits needed to identify a block depends on block size. Let b be the block size in number of pages, then the number of relevant bits is given by $(24 - \log_2 b)$. This implies that for a smallest block size of 2 pages we need 23 bits to identify a block and with the largest of 256 pages we need just 16 bits. With the default block size of 32 we will need 19 bits to identify a block. The remaining bits are carried with the BID and help maintain compatibility between a RID and a BID.

The smaller size of the BID is of a great advantage in certain access methods like index ANDing and helps in efficient query processing.

3.3 Block Fetch Operation

With the introduction of a block we need to be able to process all the records in a page of a block and all the pages of a block given a block identifier. This operation is accomplished by the Block Fetch method. While processing the records the operator needs to apply block predicate once per block and the data predicates on every record of the block. It also needs to take care of locking that needs to be done for concurrency purposes.

An important duty of the block fetch operator is to take care of any intra block prefetching that needs to be done. Prefetching starts from the first page of the block, which can be identified from the block id and continues till the end of the block which is a function of the block size.

The Block Fetch operation forms the core of quite a few of the query processing techniques discussed below. It also forms the core for many of the block based data maintenance operations like Reorg which has been described in [5].

4 Query Processing Access Methods

This section describes the details of the enhancements to the basic access methods of DB2 for MDC tables.

4.1 Block Index Scan

The Block index scan is a new operation that is introduced for MDC tables. It proceeds in the following steps:

1. From root of block index, traverse to the leaf node which satisfies the start key of a query predicate. It is to be noted that a block index has fewer number of levels than a corresponding record index. This translates to quicker access of leaf nodes from the root for a block index.

2. Scan the leaf to find a Block ID that satisfies the query predicate. During this time any predicates which the optimizer deems fit to apply as an index predicate is applied.

3. Apply the Block Fetch operator described in Section 3.3 on the qualifying block.

4. Goto step 2 until the stop key for the query is reached in the index.

The example query Q1 above is very likely to involve access to a whole stripe of blocks. Thus, the block scan sub-operation is likely to be the most efficient method of processing this query.

An important savings in block index processing is the significant reduction of the callback loops described in steps 2, 3, and 4. With a record index steps 2 to 4 will have to be repeated for every record. In contrast, these steps are performed once per block for a block index. This results in lesser CPU overheads in accessing the data records via the index. As a result, a block index based access is cheaper than a table scan over a higher range of selectivity than a corresponding record index.

An important aspect of query processing is prefetching. DB2 supports sequential detection for record indexes. When it is noticed that pages being accessed are in sequence, then pages ahead of the sequence are prefetched from disk in anticipation of processing reducing the I/O wait times. However, sequential detection is mostly useful if the index is well clustered. It also has a build up phase during which there is no prefetching and it tends to prefetch more than required in some cases. Additionally, if the index access is on the inner of a nested loop join, sequential detection becomes more difficult.

Block index scans use Block Index Look Ahead Prefetching (BILA) instead of sequential detection. The idea here is to look into the index and prefetch only those blocks of data which qualify for the query. The significantly small size of the block indexes enables the usage of index lookahead techniques for precise prefetching. The start and stop keys of the query predicate are used to determine the prefetch boundaries. Thus prefetching is triggered even for small queries and is always exact. Prefetching is triggered irrespective of the inter-block clustering of the index. Additionally, BILA prefetching on the inner of a nested loop join is quite straightforward and effective.

In Figure 2 if the index scan is for 'Canada' then BILA will prefetch precisely the blocks 21, 31, 45, 77, 127, 376, 501 and 719. Since these blocks are not in sequence, sequential detection techniques would not work in this case. The prefetch amount is an existing configuration parameter and can be adjusted appropriately based on the I/O characteristics of the system. Overall the block fetch operation coupled with BILA for inter-block prefetching results in reduced CPU usage in comparison to conventional record based indexes and better utilization of available I/O throughput and this observation is validated by our experimental results.

4.2 Block Index Only Scans

There are a set of queries which could be answered by simply accessing just an index. The example query Q4 is interested in the items from region 'Mexico' in year 2002. This query could be answered by going through the composite block index that includes all three attributes with a start and stop key of `<Mexico, 2002>` and picking up the itemIds. This type of access method is known as index only.

There are 2 type of index only access. In the first we are concerned about the existence of a particular key (EXIST, IN etc). In the second, additionally, we are concerned about the number of occurrence of a key (e.g., COUNT).

Block indexes tend to be very small in comparison to a record indexes. In the experimental results described in Section 7 a RID index occupied 222054 pages whereas the corresponding BID index was only 71 pages - a size difference ratio of 3000. Additionally the BID index had half the number of levels of its corresponding RID index. This translates to very quick query processing for index only queries where we are interested in the presence or absence of a particular index key value(s) without counting (e.g., EXISTS, IN clauses).

Block indexes do not maintain a count of the number of records in a block. Hence, queries requiring counts of the key values cannot be answered in an index only fashion. If these queries are very important to the workload, one can always create a record index on that column in addition to the block index. The optimizer will automatically pick the cheaper access method.

4.3 Block Index Anding

Index ANDing allows DB2 to answer queries with predicates on 2 or more columns of a table linked by an AND clause (example Q1) by combining multiple indexes defined on those columns. For record indexes, ANDing is accomplished by determining the intersection of record IDs which satisfies the individual predicates and then fetching and processing the resultant set of records. At this stage any data and deferred predicates are applied. Currently, the ANDing process uses a bloom filter based technique which could result in false positives. False positives are eliminated by reapplying all the index predicates on the data in a deferred fashion. It is to be noted that RIDs are 32 bit numbers and the bloom filter technique uses multiple hashing schemes to map the RIDS from this 32 bit space to a smaller space.

With MDC we have also provided support for index ANDing of block indexes and also index ANDing of block and record indexes. Unlike RIDs where all the 32 bits are relevant to identify the record, the BID contains a significantly smaller number of relevant bits as described in Section 3.2. For example, a BID may be identified with as low as 19 bits for a block size of 32 pages. This number is small enough that we can afford to use a bit map for block index ANDing and eliminate the costly hashing steps. In our example, the amount of memory needed for such a bitmap would just be 16 pages assuming the default page size of 4KB.

Keeping this in mind for block index ANDing we first attempt a bit-map based intersection approach as described above if sufficient memory is available. This would lead to an exact result for the ANDing with no false positives. It will also be quite fast since there is no costly hashing involved. In the unlikely event this memory is not available we fall back to the bloom filter method of performing index ANDing. The net result of the operation is a set of BIDs which are then processed using the Block Fetch operator described before.

The following are the reasons why block index ANDing can perform significantly better than record based index ANDing.

- Scanning smaller block indexes versus large record indexes
- Bit map based approach to index ANDing is faster than a hash based one. It results in lesser CPU usage.
- Since we AND a block id on behalf of its set of records, the number of identifiers being ANDed is smaller.
- The resulting set of BIDs are processed using the Block Fetch operator in comparison to individual processing of a set of RIDs. This reduces the overhead of processing records.

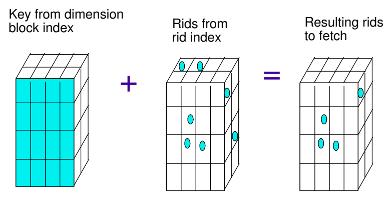

Figure 4: Index ANDing of Block and RID indexes

In the case of index ANDing of a set of block indexes with a set of record indexes, the ANDing is done in 2 stages. In the first stage we AND all the block indexes giving us a set of block ids which qualify. We then use the fact that given a record id it is easy to determine the block it belongs too. Using this we take the first record index in the ANDing and for every RID which qualifies we determine its BID and intersect it with the set of BIDs obtained by the ANDing of the block indexes. While doing this we preserve the set of RIDs which qualify from the intersection. This set of qualifying RIDs then forms the basis for further intersections with other record indexes if any in the index ANDing. The final result is a set of RIDs (in contrast to set of BIDs for pure block index ANDing) which are fetched and processed. Figure 4 graphically illustrates index ANDing of a block and record index.

Analysis of applications indicate that the ability to index AND a block index with record indexes results in efficient plans and thus was a key design and implementation issue.

4.4 Block Index ORing

Index ORing is a method which allows DB2 to answer queries with predicates on 2 or more columns of a table linked by an OR clause (for example, Q3) by combining multiple indexes defined on those columns. Currently these indexes are record indexes and ORing is accomplished by determining the union of RIDs which satisfies the individual predicates and then fetching and processing the resultant set of records. At this stage any data and deferred predicates are applied. The ORing process uses a SORT of all the RIDs with duplicate elimination to determine the union. The duplicate elimination ensures each id in the set is unique. With MDC we have introduced support for index ORing of block indexes and also index ORing of block and record indexes. In the former case the process is very similar to record index ORing except that the inputs to the SORT are BIDs and the union produces a set of BIDs which are then processed using the Block Fetch Operation instead of a set of RIDs. The major performance gains are obtained from

- Scanning smaller block indexes.
- SORT of smaller number of BIDs.
- Using Block Fetch operation.

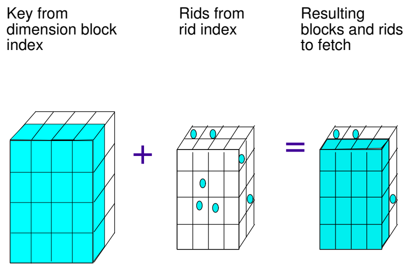

Figure 5: Index ORing of a block and RID index

In the case of an index ORing of block and record indexes the process is more involved since we need to find the union of 2 types of identifiers - BIDs and RIDs. First we do SORT with duplicate elimination of the qualifying BIDs and RIDs. This results in a set of unique BIDs and a set of unique RIDs but there exists a possibility that a record and its block are members of the SORT output. Processing this set could result in incorrect results. To solve this problem we use the property that the BID value of a RID can be determined easily. We go through the set of BIDs and RIDs and eliminate RIDs whose corresponding BIDs are present in the list. This pruned set is then processed with Block Fetch operation for BIDs in the set and direct fetch of the records represented by RIDs. We distinguish a BID from a RID by the fact that a BID is the the first record in the block and which is always a system record and cannot be a valid data record.

Figure 5 shows an index ORing operation that combines a block and a record index. As with index ANDing, analysis of applications indicate that the ability to index OR a block index with record indexes results in efficient plans and thus was a key design and implementation issue.

Relational Scans

A relational scan currently proceeds by reading every page and every record in the table, applying any predicates and projecting the required columns from the qualifying records. Prefetching is fairly simple since we are going to read every page in the table and requires fetching pages ahead of the current scan location from disk into memory. There could be sections of empty pages in the table which will be fetched and processed. Similarly, MDC tables could also have sections of empty pages and blocks. However, we can use the Block Map data structure (Section 2) to skip blocks that should not be processed including free blocks. This block map is also used by the prefetch logic to ensure empty or unwanted blocks are not brought into memory. While the consultation of the block map is an added processing overhead in comparison to non MDC relational scan processing it is quite minimal since the block map is quite small. On the other hand, if the table had a lot of updates which created empty blocks then consulting the block map results in I/O and CPU cost savings of processing these blocks.

MDC relational scans also make use of block predicates if any. These predicates would be applied once per block. If the predicate fails then we skip the block and save the associated cost of processing the remaining records in the block.

5 Query Processing: Joins and other operations

This section briefly describes the impact of block indexes and block based processing on joins and other relational operations.

Join Processing

Block Indexes can be used for processing the inner table of an index Nested-Loop join. In this operation, each outer join key is used to probe the block index and identify a set of qualifying blocks that need to be processed. This kind of processing scheme could be fairly typical in star schemas. BILA prefetching is used to prefetch the qualifying blocks for each outer access. As the experimental results show this makes a very big difference in performance.

Block Indexes can also be used gainfully in star joins where the smaller size of the block indexes, usage of bit maps and other processing enhancements described for block index ANDing, and the final block fetch operations can result in improved performance.

SORT, Aggregations and Group By operations

Block indexes can be used whenever there are sorting, aggregation, or group by requirements on dimension keys or keyparts. For instance, if a query requires a GROUP BY on the `region` attribute, it could be processed by a block index scan.

Symmetric Multi-Processing

Block based index and table scans lend themselves naturally to Symmetric Multi-Processing (SMP). In an SMP system, each task or operation is typically allocated to several co-operating agents in order to utilize the multiple CPUs. Typically, tasks are allocated using a chunk model where the next chunk is allocated to a requester. MDC access operations are naturally extended to SMP processing by allocating chunks of blocks or multiples of blocks to a requesting agent.

There are cases when allocating a block to an agent could lead to load imbalance among the agents. For example, suppose there are only 10 blocks of data to process in a 20 CPU SMP system. Alternatively, there may be 10 blocks of data to process among 10 agents but predicate processing or varying numbers of records in the blocks results in some agents having to do extra amount of work. These cases are addressed by partial block processing where each agent is allocated only part of a block to process at one time. The synchronization of these allocations are handled by new extensions. In particular, additional care must be taken to apply block predicates, skipping free blocks, etc in this mode of processing.

The optimizer uses heuristics and the existing statistics to decide if partial block processing is required. Next, heuristics are employed to allocate the granularity of partial block processing for each agent. The final division of labor and synchronization among the agents is done at run time using the chunk allocation model.

6 Query Compiler and Optimization enhancements

Given this repertoire of enhanced processing strategies for MDC tables, it is imperative that the right query plan is chosen. Hence, the DB2 query optimizer has been enhanced appropriately to cost and select the appropriate access methods, joins, and aggregation operations when dealing with MDC tables.

The query compiler obtains the knowledge of the MDC dimensions and the corresponding dimension block indexes from the catalog tables and populates the internal Query Graph Model Data structures for the table accordingly. A major task of the compiler is to take advantage of dimensions that have been rolled up (using hierarchies) from base column definitions. The subsequent task is for the query optimizer to choose the appropriate access methods and other operations when dealing with MDC tables. The following sections briefly elaborate on these two tasks.

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. For example, we can rollup dates to months, quarters, or years. 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`, appropriately called *derived predicate*, and include it for optimization. This enables the query optimizer to choose the block index on year for processing this query.

Equality predicates (and others like IN, NOT IN) are always translatable to derived predicates on the rollup column. However, inequality and range predicates are only translatable when the deriving (rollup) expression is *monotonic*. That is

`if A` $>$ `B then expr(A)` $\geq$ `expr(B)`
`OR`
`if (A` $<$ `B) then expr(A)` $\leq$ `expr (B)`

The `year` function is monotonic non-decreasing. On the other hand, the `month(date)` function is non-monotonic when dealing with dates from different years.

We have implemented an expression analyzer component in the compiler to determine monotonicity properties of expressions. We are motivated by eliminating the potential for incorrect derivations and reducing the burden of the table creator or user. Our monotonic expression analyzer is fairly comprehensive and incorporates all the built-in SQL datatypes, functions, expression clauses and operators. For, example an expression such as `year(date) + 5` will be identified as monotonic.

Query Optimizer enhancements

The query optimizer has been enhanced to consider all the block oriented processing techniques described in the previous sections. The block indexes are included during the access method selection phase of the MDC table. The optimizer's cost models have been adjusted to account for block based access to the table whether it be for table scans, index scans, index ANDing, or ORing. The optimizer's repertoire of planning strategies has been enhanced to consider the mixed index ANDing and ORing by combining block and record indexes. A new data statistic called *active_blocks* has been added to the statistics of MDC tables. This new statistic when combined with existing index cardinality statistics, e.g., *FullKeyCard*, for block indexes are mostly sufficient for performing cost and cardinality estimations of the new techniques inside the optimizer. The optimizer also deals with *cardinality bias* issues when derived predicates are introduced. In particular, a derived predicate may be used for access method and/or join method selections but the original predicate is re-applied as a data or residual and the cardinality filter factors are adjusted so that the combined effect is the same as application of the original predicate.

7 Experimental Results

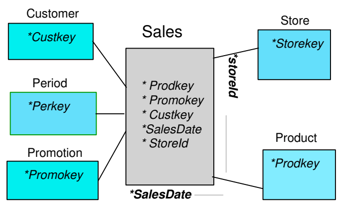

Figure 6: POPS schema

We have conducted a set of experiments using a star-schema database called *POPS* on DB2 V8 running in 8 way SMP mode on a HPUX 11i 64 bit kernel. The machine had 8 x 750 MHz PA-RISC Processors, 3 VA7400 disk arrays and 32 GB main memory. The POPS schema included a main fact table called Sales of size 36 GB and 5 dimension tables on dates, stores, products, customers and promotions. Figure 6 describes the schema. 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. All queries were run after doing a db2stop

	storeId	salesDate	itemId	TblSize
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	–
MDC Index Cluster Factor	1.0	1.0	0.894	–
Non-MDC Cluster Factor	0.99	1.0	0.896	–
MDC Table size	–	–	–	689,264
Non-MDC Table size	–	–	–	681,903

Table 1: Size comparison of MDC and normal tables and indexes

and db2start. This cold start ensures the buffers were flushed and empty. Additionally all queries were run using the db2batch tool.

The following subsections describes the experimental results. The queries include table scans with and without block predicates, rid scan with and without joins, block index scans, Index Anding and Oring and index only plans, and join queries. As can be seen the performance of the MDC table is usually better than the performance of the normal table. The speedup improvements in these experiments ranged from 4% for table scans to 75% or more for Index ORing operations while using 6% less storage space. Beta customers trying this feature are also obtaining similar large speedups on their databases.

Data Structure Comparison

The following table compares the MDC Sales table and its indexes with the non MDC version of the table and its indexes. The 3 key characteristics of an index which determine how good the index is are

1. number of levels we need to traverse from the root to get to the leafs.

2. number of leafs in the index. If we have fewer leafs for the same amount of data it means fewer pages have to be read and processed.

3. cluster factor of the index. Higher the cluster factor fewer will be the processing overhead in answering a query using that index.

The block index on salesDate has only 3% of the number of leafs of the corresponding record index. Additionally the number of levels for the index is half of the record index. This means the overhead of accessing the table via the block index is much lower than using the corresponding record index. This makes the block index based access better than a table scan for a larger range of selectivity. Thus making it more usable for range queries. The block index on the second clustering column - storeId - also shows similar characteristics as the one on storeId.

The record indexes on itemId for MDC and non MDC have same number of leafs and levels. The cluster factors are within 0.2% of each other. As the results indicate data accesses via these indexes show similar performance.

Due to the fact that we cluster data in blocks the MDC table is 1% larger than the non MDC. However this difference is very small and does not result in any significant degradation of relational scans.

Overall the space occupied by the MDC table and its indexes was approximately 6% lesser than the total space occupied by the non MDC table and its indexes.

Block Index Scans

A set of basic queries ranging from accessing a cell of data to accessing entire slices were run. The selectivity of the queries ranged from 0.1% to 25%. In the MDC case they resulted in block index scans and in the non MDC case they resulted in record index scan using the index of same definition as the block index. The following is a description of the queries

B1. Equality: `select sum(handling_charges) from sales where storeId = 1`

B2. Range on date: `select sum(handling_charge) from sales where salesDate between 1996050 and 1996090`

B3. Range on 2 dimensions: `select sum(handling_charge) from sales where salesDate between 1996010 and 1996050 and storeId = 2`

B4. Query of a cell: `select sum(handling_charge) from sales where salesDate = 1996030 and storeId = 1`

Figure 7 summaries the results.As can be seen with block indexes significant performance improvements were observed.

Combining Block and Record Indexes

The 2 fact tables had record indexes on prodkey field which is the product dimension. We ran 2 queries

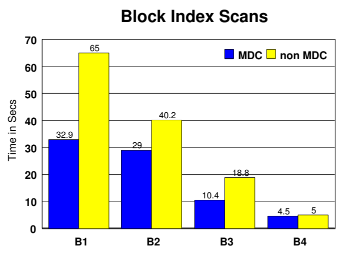

Figure 7: Block Index Scan. Results in secs

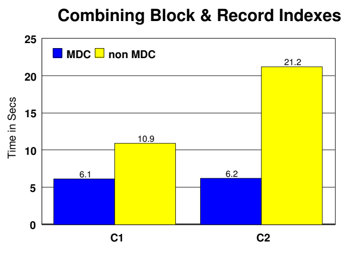

Figure 8: Combining block and record indexes.

which results in index ANDing and ORing plans. Query C1 has a range predicate of salesDate (a clustering column) and on prodkey linked by an AND clause. In the MDC case it results in an index ANDing of the block index on salesDate with the record index on prodkey. In the non MDC case it results in the index ANDing of the record index on salesDate with the record index on prodkey. The net result in both case is a set of RIDs which are then individually fetched.

In the case of query C2, the range predicates on salesDate and prodkey are linked by an OR clause. This results in index ORing plans. In the MDC case it results in the ORing of the block index on salesDate with the record index on prodkey. This results in a mixed set of BIDs and RIDs which are subsequently processed. In the non MDC case it results in the index ORing of the record index on salesDate with the record index on prodkey. The net result is a set of RIDs which are subsequently processed.

C1. Index ANDing of BID and RID Indexes: `select sum(quantity_sold) from sales where salesDate between 1996032 and 1996038 and prodkey between 12000 and 12600`

C2. Index ORing of BID and RID Indexes: `select sum(quantity_sold) from sales where salesDate between 1996052 and 1996054 or prodkey between 12000 and 12050`

As Figure 8 indicates, significant performance improvements are observed for the MDC index combinations.

Table Scans

We ran 2 table scan queries - T1 and T2 - described below.

T1 Block Predicates: `select sum(float(handling_charge)) from sales where salesDate not in (1996000, 1996020, 1996180)`

T2 Normal : `select storekey, sum(handling_charge) from sales group by storeId`

Query T1 results in a table scan with block predicates on salesDate for MDC. This means all blocks with salesDate in (1996000, 1996020, 1996180) are not processed except the first record in these blocks. This results in a performance improvement for MDC which is visible in the bar chart.

Query T2 results in normal table scans for both MDC and non MDC. As seen in the bar chart the performance figures are similar. The slight improvement for MDC scan is due to the usage of larger block granularities in SMP processing.

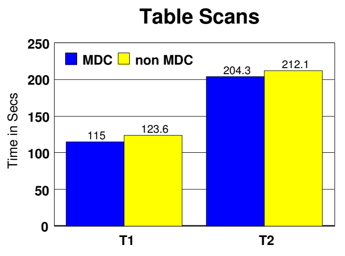

Figure 9: Table Scans

Multi Table Joins

We ran queries with 2 and 4 table joins in them as described below:

N1 Nested Loop : `select sum(quantity_sold),sum(shelf_cost),count(*) from store, sales where store.storekey=sales.storeId and store_number='10'`

S2 Multi Table : `select sum(quantity_sold), sum(total_display_cost) from period, store, product, sales where period.perkey = sales.salesDate and`

```
store.storekey = daily_sales.storeId and
product.prodkey = daily_sales.prodkey
and store_number = '01' and
product.category=42 and calendar_date
between '01/01/1996' and '01/28/1996'
```

Query N1 is a join of the fact table with the store dimension. It results in a nested loop join plan with the access of the inner being via the block index on storeId for MDC. In the non MDC case the access of the inner is via the record index on storeId. As the bar chart indicates there is a very significant performance difference. This is partly because with BILA we do a very efficient job of prefetching the nested loop join inner pages.

Query S2 is a multi table join of the fact table with 3 dimensions. In the MDC case 2 of the 3 join columns are also the clustering columns for the table. In this example we are trying to find the total quantity sold for 1 product category over 28 days in a store '01'.

The bar chart below the results of these queries. As we can see there is a significant performance improvement in the MDC case.

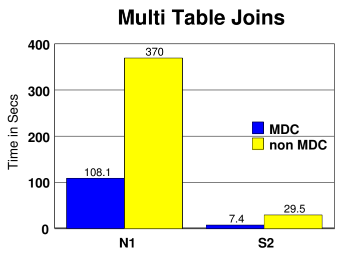

Figure 10: Multi Table Joins. Results in secs

Record Index Scans

The following 3 queries result in record index scans for both MDC and non MDC. They touch the 3 star logical dimensions on which record indexes have been defined.

R1 Nested Loop : `select sum(quantity_sold),sum(shelf_cost),count(*) from product, sales where product.prodkey=sales.prodkey and product.category=42`

R2 Equality : `select sum(handling_charges) from sales where promokey = 2`

R3 Range : `select sum(handling_charge) from daily_sales where custkey between 900000 and 900500`

Query R1 exercises the product dimension. It results in a nested loop join with the inner being a record index access.

Query R2 is on the promotion dimension. It has an equality predicate which is answered using the record index on promokey.

Query R3 runs on the customer dimensions. It has a range predicate which is tackled by a record index scan on custkey.

As the results below indicate there is no significant performance differences between MDC and non MDC.

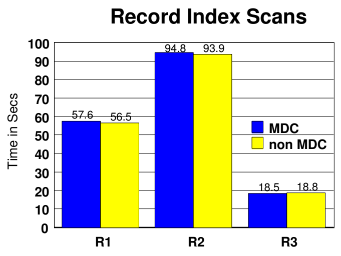

Figure 11: Record Index Scans. Results in secs

8 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 manage-ability of data. We believe that Multi-Dimensional Clustering is an effective data organization technique for many modern database applications. We described the the various enhancements to query processing in order to take advantage of this new feature. Our experimental results showed the significant performance improvements that are possible for a range of atomic and more complex queries.

References

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

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

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

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

[5] S. Padmanabhan et al. Multi-Dimensional Clustering: a new data layout scheme in DB2. In *Proceedings of the ACM SIGMOD Conference*, 2003. to appear.