ABSTRACT

MARTIN, MICHAEL W. Implementing Real-Time OLAP with Multidimensional Dynamic Clustering. (Under the direction of Dr. Rada Y. Chirkova).

This dissertation details multidimensional dynamic clustering (MDDC), a new multidimensional data clustering method that supports efficient On-Line Analytical Processing (OLAP) queries [24]. OLAP queries are generally multidimensional business intelligence queries that require large amounts of data, complex table joins, and aggregate data calculations to produce useful results [19]. MDDC allows OLAP queries to execute very efficiently on base tables while requiring fewer materialized views, aggregates, and secondary indexes than many previous solutions that support OLAP queries. MDDC also allows dynamic inserts, updates, and deletes on stored data without reorganization of the underlying data structures. This is especially important for real-time data warehouses that must provide efficient OLAP query access while continually adding data to and maintaining data in tables. Such data warehouses cannot stop query processing to reorganize data structures when data exceeds preset thresholds such as a fixed number of key values. This dissertation provides a precise definition of MDDC, including a comparison with multidimensional hierarchical clustering (MHC), and presents experimental results to substantiate the claims that MDDC is more dynamic than MHC and provides symmetry that is better than or equal to that of MHC.

Implementing Real-Time OLAP with Multidimensional Dynamic Clustering

by

Michael W. Martin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh

2006

Approved By:

Dr. Xiasong Ma	Dr. Ting Yu
Dr. Rada Y. Chirkova Chair of Advisory Committee	Dr. Christopher G. Healey

I would like to dedicate this dissertation to the three people who would not accept anything less than my having the highest level of education possible.

I dedicate it to my mother, Alta Irene Simmons Martin. In her own education, she never advanced past the seventh grade. She performed hard manual labor all her short life. Understandably, she insisted that I have as much education as possible so that I would have a better life.

I also dedicate it to my father-in-law, Malton Ruffin Tripp Sr., and my wife, Shelia Ann Tripp Martin. Through many years they insisted that I obtain a Doctor of Philosophy degree in Computer Science. They always provided positive encouragement and they never gave up on me even when I gave up on myself.

Biography

My educational background consists of a Bachelor of Science in Computer Science from the University of North Carolina at Wilmington in 1986, a Master of Science in Management from North Carolina State University in 1990, and a Master of Computer Science from North Carolina State University in 1997.

My professional experience consists of 19 years in information technology with positions of increasing responsibility including programmer analyst, database administrator, systems analyst, project manager, development manager, and most recently director of architecture and planning.

In addition, I won the GlaxoWellcome Inc. CEO's award for a national sales and marketing data warehouse and hold U.S. utility patent 6,003,036 relating to the storage, maintenance, and access of multidimensional data warehouses. Finally, the intellectual property in this dissertation is protected by a pending U.S. utility patent.

Acknowledgements

I would like to acknowledge my advisory committee for their help and guidance. I especially would like to acknowledge and thank Dr. Rada Y. Chirkova whose patience, assistance, and expert advice made this dissertation possible.

Contents

List of Figures vii

List of Tables viii

1 Introduction 1

2 The Real-Time OLAP Problem 4
2.1 OLAP Data Model . . . 6
2.2 OLAP Dimensions and Hierarchies . . . 6
2.3 OLAP Facts . . . 7
2.4 OLAP in Real Time . . . 7
2.5 No B-tree Equivalent for OLAP . . . 8

3 Related Work 10
3.1 Non-Clustering Indexes . . . 10
3.2 Clustering Indexes . . . 11
3.3 MHC . . . 11
3.3.1 UB-trees . . . 12
3.3.2 Z-ordering . . . 12
3.3.3 Z-ordering Example . . . 13
3.3.4 Hierarchical Clustering . . . 16
3.3.5 Z-ordering and Hierarchical Clustering . . . 17

4 MDDC Theory 21
4.1 Variable Length Bit Strings . . . 22
4.2 Dynamic Bit Interleaving . . . 23
4.3 MDDC Operations . . . 24
4.3.1 Queries in MDDC . . . 24
4.3.2 Point Queries . . . 25
4.3.3 Slice or Partial Match Queries . . . 25
4.3.4 Maintenance . . . 25
4.3.5 Key Recycling . . . 26
4.3.6 Key Participation . . . 26
4.4 Incremental Costs . . . 26

4.5 Summary of MDDC 27

5 Implementation **28**
5.1 Simulation Platform 28
5.2 Simulation Software 28
5.3 Simulation Parameters 29
5.4 Simulated SQL Statements 29
5.5 Simulated Database Tables 30
5.6 Special Simulation Challenges 30

6 Experimental Results **31**
6.1 National Grocery Chain 31
6.1.1 Data Model 31
6.1.2 Test Criteria 31
6.1.3 Data Loading 32
6.1.4 Queries 32
6.1.5 Results 33
6.2 TPC-H 34
6.2.1 Data Model 38
6.2.2 Test Criteria 38
6.2.3 Data Loading 39
6.2.4 Queries 39
6.2.5 Results 39
6.3 Final Evaluation of MDDC and MHC 41

7 Conclusions and Future Work **44**

Bibliography **46**

A Introduction to Appendices **50**

B Tables definitions for experiments **51**

C Queries for experiments **55**

D Detailed experimental results **58**

E Examples **69**

List of Figures

1.1 Hierarchical SQL Query 2
1.2 Slice SQL Query 2

2.1 OLAP SQL query 5
2.2 OLAP SQL query 5

3.1 Z-order Product SQL Query 15
3.2 Z-order Month SQL Query 15
3.3 Typical OLAP SQL Query 17

4.1 Uniform Depth Z-address Space 24
4.2 Variable Depth Dynamic Bit Interleaving Space 25

5.1 Simulated SQL Query 30

6.1 Grocery Sales Product Dimension Query 1 34
6.2 Grocery Sales Product Dimension Query 2 35
6.3 Grocery Sales Promotion Dimension Query 1 35
6.4 Grocery Sales Promotion Dimension Query 2 36
6.5 Grocery Sales Store Dimension Query 1 36
6.6 Grocery Sales Store Dimension Query 2 37
6.7 Grocery Sales Month Dimension Query 1 37
6.8 Grocery Sales Month Dimension Query 2 38
6.9 TPC-H Part Dimension Queries 41
6.10 TPC-H Supplier Dimension Queries 42

List of Tables

3.1 Product Dimension 14
3.2 Month Dimension 14
3.3 Sales 14
3.4 Customer Dimension 18
3.5 MHC HSE 19

4.1 MDDC HSE 22

B.1 Grocery store dimension 52
B.2 Grocery product dimension 52
B.3 Grocery promotion dimension 52
B.4 Grocery month dimension 52
B.5 Grocery sales fact 52
B.6 TPC-H estimated database size 53
B.7 TPC-H PART dimension 53
B.8 TPC-H SUPPLIER dimension 53
B.9 TPC-H LINEITEM fact 54

C.1 Grocery sales experiment - queries 56
C.2 TPC-H experiment - queries 57

D.1 Grocery sales results - part 1 59
D.2 Grocery sales results - part 2 60
D.3 Grocery sales results - part 3 61
D.4 Grocery sales results - part 4 62
D.5 Grocery sales results - part 5 63
D.6 TPC-H Lineitems results - part 1 64
D.7 TPC-H Lineitems results - part 2 65
D.8 TPC-H Lineitems results - part 3 66
D.9 TPC-H Lineitems results - part 4 67
D.10 TPC-H Lineitems results - part 5 68

E.1 Product dimension 70
E.2 Month dimension 70

E.3 Store dimension 70
E.4 Sales fact 70

Chapter 1

Introduction

On-Line Analytic Processing (OLAP) applications typically provide business analysts and managers with multiple views and organizations of critical business data. OLAP applications require underlying data structures to efficiently provide access to data from any combination of the component keys or dimensions in the data. OLAP queries typically restrict the data by some combination of dimensions and then aggregate or compute statistics on some subset of the attributes of the data [19].

OLAP data structures and methods must overcome several difficult challenges such as the requirement for performance symmetry given that a query might restrict the data on any combination of dimensions. OLAP data structures and methods must also deal with the "curse of dimensionality" or the decreasing selectivity in any one dimension as the total number of dimensions in the data structure increases. Finally, OLAP data structures must also deal with the common problem of maintaining a balanced and efficient data structure as data content changes. Increasingly, OLAP data structures and methods must also maintain data in real time [20, 7] while simultaneously executing efficient OLAP queries. All the while, the size and corresponding performance challenges of these data warehouses are increasing at a staggering rate.

OLAP slice searches typically restrict data by dimension key values or hierarchical key values within the dimensions, as the first SQL query in Figure 1.1 that follows demonstrates. OLAP applications rarely restrict data by key order slice searches such as the second query in Figure 1.2 that follows demonstrates [19, 21, 23].

Practitioners and researchers have created many techniques and methods in an attempt to efficiently handle such OLAP queries. One very common technique is aggrega-

```
SELECT DOLLARS FROM SALES, CUSTOMER
WHERE SALES.CUSTOMER_ID=CUSTOMER.CUSTOMER_ID AND
CUSTOMER.STATE='NC'
```

Figure 1.1: Hierarchical SQL Query

```
SELECT DOLLARS FROM SALES
WHERE CUSTOMER > 4 AND CUSTOMER < 50
```

Figure 1.2: Slice SQL Query

tion. This involves the replication of pre-computed and pre-joined data into one or more tables. While this technique is very helpful, it is expensive in terms of space and time and inhibits real-time updates since the database management systems (DBMS) must update and synchronize multiple tables when an update occurs. The use of multiple indexes, especially bit mapped indexes, is another common technique. The indexes provide multiple views and increase query efficiency. But, they also delay update operations in proportion to their numbers and do nothing to cluster the data. Since they do not cluster the data they are more suited for small result sets [21, 23]. Much work has been done in pursuit of multi-dimensional clustered indexes. While clustering is a very effective technique for increasing query efficiency, clustering data in more than one dimension is very difficult. Most such indexes are overly complicated, inefficient to manage, and are not effective in maintaining the proper data organization for which they were originally designed [16]. Multidimensional hierarchical clustering (MHC), a method involving the B-tree [21, 23], which has proven itself so well in on-line transaction processing (OLTP) applications, has emerged. The technique uses Z-ordering which interleaves bits from multiple keys to enforce symmetry and then stores the interleaved key in B-trees [2]. It also encodes dimensional hierarchies into its keys to aid clustering. This technique provides efficient multidimensional clustering with all the proven advantages of B-trees since it is in fact uses B-trees. It is not without problems,

however. MHC uses fixed length keys for each level in each hierarchy. This limitation, impedes symmetry and prevents the technique from being completely dynamic and thereby not providing a real-time capability.

Multidimensional dynamic clustering (MDDC) [24] like MHC capitalizes on all the advantages of MHC but unlike MHC uses variable length keys and dynamic bit interleaving in lieu of fixed length keys and Z-ordering. As a result MDDC is completely dynamic and provides a real-time capability for data warehouses since it never requires reorganizations. In addition, MDDC demonstrates symmetry that is superior to that of MHC. The only disadvantages of MDDC when compared to MHC is the additional space that MDDC requires to store the variable length keys and an inability to handle conventional range queries. Depending on the number of dimensions, hierarchies in each dimension, and metric fields this additional space requirement could be very high but based on the experiments in this dissertation is less than 10%.

This dissertation fully details MDDC and then provides empirical evidence to substantiate the claims associated with MDDC.

Chapter 2

The Real-Time OLAP Problem

The real-time OLAP problem involves dynamic storage, maintenance, and access of multidimensional data for OLAP queries [7].

Consider a database consisting of sales data in dollars, with a primary key composed of product, month, and store. Further, suppose that all products each fall into a specific product category, months are related to a specific quarter and year, and stores each fall into one and only one zip code, state, and region. Tables E.1, E.2, E.3, and E.4 provide an example of such a database.

The real-time OLAP problem involves real-time query, insert, update, and delete operations against such tables. Applications or users are able to efficiently and dynamically insert, update, or delete records from any dimension table or fact table in the database without reorganizing any data structures. There are also not any preset upper limits on the values of keys in any of these tables. Queries involve restrictions or criteria on any combination of dimensions. The following two SQL queries in Figures 2.1 and 2.2 represent typical queries involving combinations of dimensions. Query 2.1 returns the total sales by store from all store sales where the year is 1998 and the state is 'NC'. Query 2.2 returns the total sales by product for all products that are categorized as 'DRY GOODS'.

To provide an effective multidimensional data clustering, a multidimensional data structure that provides ordered and direct access to data must support slice and point queries on any combination of participating dimensions with the same efficiency as if the combination of dimensions make up a prefix of a one-dimensional data structure supporting clustered access to the data [13].

```
SELECT STR_NM, SUM(SLS_DLR) FROM SLS_FCT,
STR_DIM, PROD_DIM, MO_DIM
WHERE STR_DIM.STR_ID=SLS_FCT.STR_ID
AND PROD_DIM.PROD_ID=PROD_DIM.PROD_ID
AND MO_DIM.MO_ID=SLS_FCT.MO_ID
AND STR_DIM.ST_NM='NC'
AND MO_DIM.YR_NM=1998
GROUP BY STR_DIM.STR_NM
```

Figure 2.1: OLAP SQL query

```
SELECT PROD_NM, SUM(SLS_DLR) FROM SLS_FCT,
STR_DIM, PROD_DIM, MO_DIM
WHERE STR_DIM.STR_ID=SLS_FCT.STR_ID
AND PROD_DIM.PROD_ID=PROD_DIM.PROD_ID
AND MO_DIM.MO_ID=SLS_FCT.MO_ID
AND PROD_DIM.PROD_CAT_NM='DRY GOODS'
GROUP BY PROD_DIM.PROD_NM
```

Figure 2.2: OLAP SQL query

2.1 OLAP Data Model

While both dimensional and normalized data models accommodate OLAP data, the problem specification and theory in this dissertation assumes that the data model is dimensional for simplicity, query efficiency, ease of use, and increased probability of correct results. Tables in OLAP dimensional databases are divided into dimension tables and fact tables [19].

Dimension tables such as E.1 contain all the primary key sources for all foreign keys in a dimensional model. Each dimension table has one primary key, typically a tuple identifier. Dimension tables contain most of the textual or descriptive attributes in a dimensional model. Dimension tables also contain one or more hierarchies related to each primary key for a dimension. For example, a customer dimension table might contain a city and state attribute for each customer. In a flat dimensional model, the customer dimension would contain all the attributes for city and state as well as customer. A flat dimensional model might also contain separate dimension tables for city and state in addition to the attributes for city and state in the customer dimension. A snow-flaked dimensional model would only contain foreign keys to city and state in the customer dimension and would house all other attributes for city and state in separate city and state dimension tables.

Fact tables such as E.4 contain foreign keys that reference one dimension. They also contain attributes that are usually but not necessarily numeric and are dependent on the primary key of the fact table. These attributes are known as metrics.

Fact tables are typically orders of magnitude larger than dimension tables. This is expected since fact tables contain rows that could include all the possible combinations of key values from all dimensions that the fact table includes in its definition. Albeit, fact tables rarely contain even a small fraction of all possible dimension combinations. The ratio of actual rows in a fact table compared to all rows and combinations of dimension values is known as the sparseness of the fact table [19].

2.2 OLAP Dimensions and Hierarchies

Each dimension table contains one or more hierarchies. There is always at least one trivial hierarchy for each dimension made up of only one level or the primary key from the dimension. Other hierarchies include the primary key and all the other higher levels for

that hierarchy. Each level of a hierarchy assigns one and only one key value to each primary key value in the dimension table. Therefore, for each dimension, D, there are h hierarchies, each with some number of levels l for each hierarchy h.

2.3 OLAP Facts

If a fact table contains n dimensions $D_1, D_2, D_3, D_4, ..., D_n$, then a query can constrain the fact table on 2^n possible combinations of dimensions since there are n dimensions and each one can either be included or not be included as a constraint.

The most optimal way to store the fact data is to cluster or sort it by the dimensions that queries most often use to constrain the data [17, 21, 23]. But, this is a problem for conventional indexes such as B-trees because they require too much replicated data to support all the possible combinations.

A multidimensional data structure need not use all n dimensions for a given fact table to organize the data. The multidimensional data structure can use any subset of the n dimensions. When a fact table contains a large number of dimensions, including all of them in the data structure may dilute the organization of the data too much and cause "curse of dimensionality" problems. Depending on query patterns it is often more practical to only use the dimensions that queries most commonly constrain. The fact table can still use the remainder of the dimensions to fulfill any requirements for constraints such as primary keys.

2.4 OLAP in Real Time

OLAP applications conventionally load data in batch. As a result there is time to completely reorganize all data if necessary. Real-time OLAP applications are more similar to OLTP applications. They must handle new data incrementally as it arrives without major data structure reorganizations. Simultaneously, the OLAP application must provide query performance good as or better than conventional OLAP applications. This requires everything regarding the OLAP data structure to be completely dynamic. There can be no preset limits on key values or any preset number of records of any kind in the data structure. Each new record insert, update, or delete must result in only a limited amount of work as is the case with OLTP applications.

2.5 No B-tree Equivalent for OLAP

B-trees are ubiquitous to On-Line Transaction Processing (OLTP) database systems since their performance is so efficient, versatile, and predictable. The B-tree solves most index requirements in OLTP systems and therefore there is little demand for alternatives at this time [42]. OLAP database systems to date have not had such a widely successful solution to implement.

A B-tree is also called a multi-way tree, it is a fast data-indexing method that organizes the index into a multi-level set of nodes. Each node contains a sorted array of key values. Two important properties of a B-tree are that all nodes are at least half-full and that the tree is always balanced. That is, in access operation must read an identical number of nodes in order to locate all keys at any given level in the tree. A well-organized B-tree, with well sized nodes, will have only three to four levels [40].

Most OLTP applications depend on the B-tree to limit the amount of work required by each transaction since a B-tree has a limited number of levels for any database operation to access or update. The conventional B-tree, which collectively here refers all derivatives of the B-tree, has evolved into a very robust and efficient solution to implement conventional operational or OLTP databases. The B-tree data structure provides good concurrency, recoverability, maintenance, predictable query performance, predictable update performance, and does not require periodic reorganizations. B-tree structures in their native forms, however, do not provide a good solution to implement OLAP databases. They only work well when queries specify a complete key or a prefix of the key [42, 13, 40].

As the next chapter illustrates, conventional B-trees are not the only data structures that fall short in this area. There is not a robust and dynamic data structure to support OLAP databases. The crowded field of candidate data structures for OLAP evidences this fact. Each of the current solutions work well for some applications and not others or work well in one aspect of their functionality but not in other aspects of their functionality [11, 39]. The lack of a solution as versatile and robust for OLAP applications as B-trees are for OLTP applications remains.

As this dissertation details, MDDC provides a B-tree solution to the real-time OLAP problem. Like standard B-trees, MDDC enables real-time or dynamic updates for OLAP in much the same way that standard B-trees provide such functionality for OLTP. Unlike standard B-trees, MDDC provides symmetric and efficient query access no matter

which combinations of dimensions and associated hierarchies that query selects.

Chapter 3

Related Work

There are two major sectors of research regarding data structures for OLAP applications. The first sector focuses on data structures that only provide indexes and do not cluster the underlying data [39]. The second sector focuses on data structures that, in addition to any indexes they provide, do cluster the underlying data [14, 9, 4].

3.1 Non-Clustering Indexes

Solutions from the first sector of research typically utilize secondary indexes, inverted lists, or more recently bit mapped indexes to reduce the search space of multidimensional queries without regard to the physical organization or clustering of the data. Relational databases typically employ solutions from this sector since these solutions work well with heap table organizations. Relational databases depend on heap structures for OLTP [13]. But, research has shown that database management system (DBMS) software can achieve large gains by clustering data according to the keys that queries use to filter the data. This is true even for applications that are not multidimensional in nature [3]. When databases store records in a random fashion and do not cluster them by key values, they require virtually one I/O per record on average to fetch the data from secondary storage [17]. Not surprisingly then, such solutions only work well for queries that access a very small number of records relative to the overall search space. This is true even when the database is capable of combining multiple indexes in one query. More formally, queries must have very small conjunctive selectivity or combined dimensions query constraints to benefit from these solutions. In one study, OLAP queries attempting to use bitmap indexes perform no

better than full table scans for queries where the conjunctive selectivity exceeds 3.33% of the total table size. This study also demonstrates that the tendency of OLAP queries to constrain data based on hierarchically organized dimensions further amplifies this problem [21, 23].

3.2 Clustering Indexes

Solutions from the second sector of research attempt to efficiently solve the very difficult problem of multidimensional clustering. Unlike B-trees for OLTP, most of these conventional OLAP data structures such as grid files, hB-trees, and R-trees are not efficient and practical in one or more of the areas of space utilization, update predictability, query efficiency, recovery, concurrency control, and complexity [25, 9, 38, 13, 14, 15, 18, 3].

For instance, grid files are reasonably easy to maintain and provide good concurrency but do not provide predictable query performance and have a tendency to consume large amounts of disk space and require much unnecessary I/O. An inability to efficiently handle sparse data is the primary underlying cause of these problems in grid files [27, 34, 15, 40]. While a large number of OLAP software applications implement Hypercubes as their primary solution, they exhibit sparseness, disk space, and I/O query performance problems. hB-trees and R-trees attempt to furnish OLAP applications with the robustness that B-trees furnish OLTP applications but exhibit problems with concurrency, maintenance, complexity, and predictable query performance [14, 9]. Most other known OLAP data structures have one or more major weaknesses and are not able to provide the same level solution for OLAP databases that B-trees are able to provide for OLTP databases.

Of course the exception to this is the UB-tree [2] since it is in fact a B-tree with multidimensional ordering. But, the UB-tree alone does not cluster the data optimally for hierarchical OLAP data. Multidimensional hierarchical clustering (MHC) clusters data in UB-trees with dimensional hierarchies but requires a tradeoff between dynamic or real-time updates and symmetry.

3.3 MHC

MHC combines UB-trees and hierarchical clustering. MHC overcomes the difficult challenges of multidimensional clustering by encoding the clustering in one dimensional

key values and simply storing the key values in one dimensional B-trees. Therefore, the problems associated with clustering data by more than one dimension are absent. The only problems that remain are to assign key values so that symmetric clustering is enforced and so that the overall data structure is dynamic [21, 23].

3.3.1 UB-trees

The UB-tree or Universal B-tree [2], offers promise since it supports multidimensional clustering and is able to utilize the B-tree as an underlying data structure without altering the properties of the B-tree. This is true since the UB-tree employs Z-ordering or an interleaving of bits from each key in a composite key [28] to map multidimensional spaces to a linear sequence of points suitable for storage in linear data structures that preserve order such as B-trees. If a multidimensional composite key contains 3 component keys or dimensions of 16 bits each, the resulting interleaved key in the UB-tree has 48 bits with the priority of the bits alternating between the dimensions on each bit position. This technique merely alters the sorting or collating sequence of B-trees and does not affect its behavior in any other way. The UB-tree provides a solid host data structure for multidimensional clustering but does not address the multidimensional clustering problem itself.

3.3.2 Z-ordering

UB-trees and therefore MHC are based on the concept of bit interleaving from Z-ordering [28, 29]. This technique maps multidimensional spaces to one-dimensional spaces. For this reason, Z-ordering is known as a space filling curve. When a Z-ordering curve maps two dimensions into one dimension it has a distinctive Z shape, hence the name. Other research efforts have since made advancements in the area of space filling curves such as Gray and Hilliard Ordering [10]. MHC could very well make use of the other more advanced space filling curves that provide better proximity for spatial applications but, bit-interleaving via Z-ordering is sufficient for MHC since MHC targets OLAP applications and only partially orders the data as illustrated later.

The Z-ordering technique is very straightforward. It shuffles or interleaves the bits from multiple keys together to form one contiguous key. The technique can order the interleaved key as if it were a single key. Consider a composite multidimensional key made up of the keys or dimensions A, B, and C. Assume that each key is 3 bits long. Given a set

of values for A, B, and C of 101, 110, 001, a new interleaved key or Z-address takes the first bit from each of the three keys, then the second bit, and finally the third bit from each key so that the resulting single key in shuffled bit format is 110010101.

More formally, a Z-address Z(k) is the ordinal number of a multidimensional composite key k from a tuple or record on the Z-curve. Z-ordering calculates a Z-address for s bits in each of the d dimensions in the key k as follows:

$$Z(k) = \sum_{j=0}^{s-1} \sum_{i=1}^{d} k_{i,j} \cdot 2^{j \cdot d + i - 1}$$

Of course, this represents the case where each key has the same number of bits. In the case where one key exhausts its bits before the other keys, it simply fills in the missing bits with zeroes so as not to affect the collating. Note that with Z-ordering and other space filling curves, the technique predetermines the multidimensional space including the number of bit partitions or division points for each dimension and places data points into the multidimensional space as they become available.

Maintenance and queries for Z-ordering are relatively simple. Inserts, updates, and deletes simply compute the Z-address for the multidimensional composite key and then follow the rules of the underlying data structure.

Queries are slightly more complex. Queries performing partial matches or multidimensional slice searches utilize a wildcard or “don’t care” bit for each bit in any unspecified keys. Consequently, queries must search several paths in the underlying data structure pruning sections of the data structure where possible. Otherwise, queries invoke the underlying rules of the data structure as with inserts, updates, and deletes.

3.3.3 Z-ordering Example

Consider an OLAP example involving Sales dimensioned by Product and Month as Tables 3.1, 3.2, and 3.3 depict.

In this example, Z-ordering interleaves the bits from the Product and Month keys in the Sales fact table together to form one key per record as specified by Z-ordering. Notice that for each Z-address in the Z-ordering above the changes or breaks in bits from Product occur much earlier in the Z_ADDRESS and therefore dominate the collating sequence in the record order.

Table 3.1: Product Dimension

KEY		DESCRIPTION
DEC	BIN	
1	0001	Nylon Wind Breaker
2	0010	Rain Coat
3	0011	Full Length Overcoat
4	0100	Heavy Waist Coat
5	0101	Light Jacket
6	0110	Hiking Boots
7	0111	Rain Pants
8	1000	Leather Gloves

Table 3.2: Month Dimension

KEY		DESCRIPTION
DEC	BIN	
200301	110000111001101101	January, 2003
200302	110000111001101110	February, 2003
200303	110000111001101111	March, 2003
200304	110000111001110000	April, 2003
200305	110000111001110001	May, 2003
200306	110000111001110010	June, 2003
200307	110000111001110011	July, 2003
200308	110000111001110100	August, 2003
200309	110000111001110101	September, 2003
200310	110000111001110110	October, 2003
200311	110000111001110111	November, 2003
200312	110000111001111000	December, 2003

Table 3.3: Sales

Z_ADDRESS	PRODUCT	MONTH	DOLLARS
0101001000111001101101	1	200301	53
0101100000111001101111	2	200303	23
0101101000111001110110	3	200310	44
0111000000111001111000	4	200312	39
0111001000111001110001	5	200305	77
0111100000111001110010	6	200306	45
0111101000111001110100	7	200308	81
1101000000111001101101	8	200301	81

```
SELECT MONTH, DOLLARS
FROM SALES WHERE PRODUCT=3
```

Figure 3.1: Z-order Product SQL Query

```
SELECT PRODUCT, DOLLARS
FROM SALES WHERE MONTH=200301
```

Figure 3.2: Z-order Month SQL Query

Therefore, when a UB-tree stores these fact records, the product key dominates the placement of records. If the block size is two and the B-tree fully packs the blocks, then the first two records will be in block one, the second two in block two, and so forth. Consider the SQL query in 3.1 that restricts data based on Product.

Recall that Z-ordering executes partial match queries against the underlying data structure by specifying all available explicit bits and using the "don't care" or wildcard bits for unspecified bits. When queries search the data in the UB-tree, it uses these explicit bits along with the wildcard bits to prune blocks that the queries do not need in order to satisfy their requirements.

This particular query is able to prune all but one leaf block in the UB-tree. This is true because even when a query designates a wildcard for every Month bit, the UB-tree clusters all the records where Product is equal to 3 into one block. Since the UB-tree is a non-dense index and the index portion of the UB-tree can easily reside in random access memory (RAM), it is quite possible that this specific query will only generate one I/O for the one block that it needs. A query from the example selecting any value for Product should be able to utilize the index with similar efficiency.

In contrast, consider the query 3.2 that restricts data based on Month. This query is not able to effectively leverage the UB-tree and needs to scan every leaf block in the UB-tree or essentially perform a full scan of the data. When a query designates wildcards for all the bits for Product, the query cannot prune any blocks from the UB-tree. In this

example, this is true no matter which value of the Month the query uses to restrict the data.

As demonstrated, the content of the data in this example prevents balanced query performance and renders the UB-tree ineffective in producing symmetric access to OLAP data. These same deficiencies are present in Gray and Hilliard Ordering [10] data structures as well since they do nothing to control the relative bit placement in participating keys.

Research for the Variable UB-tree [41] also recognizes this problem and in fact is an attempt to resolve this problem. This research terms the problem the "puff pasty effect" since attempting to extract a hyperplane from a hypercube based on a partial match query in a non-symmetric UB-tree is analogous to cutting through a "puff pastry" the long way or splitting the layers in lieu of cutting through the short way or across the layers. The extraction or query must access too many blocks rendering it like a full scan of the hypercube in the worst case. Unfortunately, while the Variable UB-tree improves symmetry, it is a static solution and forces a reorganization anytime data changes.

Ideally, an OLAP multidimensional data structure should provide access to the data that is symmetric regardless which participating key or combination of participating keys the query uses to restrict the data. Moreover, this is the impetus for multidimensional data structures in the first place.

Clearly, while UB-trees can provide symmetric access to multidimensional data, the distribution of key values that are present in the data has the potential to destroy symmetry in UB-trees and render them ineffective.

3.3.4 Hierarchical Clustering

Thus, UB-trees in isolation are not enough. In their native form, UB-trees order data according to foreign key values from dimensions. For instance, an OLAP fact table with the dimensions customer, product, and time would have a multidimensional or Z-ordering based on the primary keys for customer, product, and time. Typically, OLAP queries do not base queries on such key values. This is especially true for conventional range queries. An OLAP query selecting all products with a primary key value less than or greater than a particular foreign key value corresponding to a dimension would not be common. This is because primary keys are often meaningless, tuple identifiers. It is not uncommon for OLAP queries to select all the data for combinations of single dimension values such as a

```
SELECT CUSTOMER.NAME, TIME.WEEK,
SUM(SALES.DOLLARS)
FROM SALES,
CUSTOMER,
TIME
WHERE SALES.CUSTOMER_ID=
CUSTOMER.CUSTOMER_ID AND
CUSTOMER.CITY='NEW YORK' AND
CUSTOMER.STATE='NEW YORK' AND
TIME.MONTH='MARCH' AND TIME.YEAR='2004'
GROUP BY CUSTOMER.NAME, TIME.WEEK
```

Figure 3.3: Typical OLAP SQL Query

single key value of customer and a single key value of product but, more typically OLAP queries select slices of data that correspond to hierarchical values in dimensions. As the SQL example in Figure 3.3 illustrates, a typical OLAP query might select all customers in a city and all days for a month [19]. Therefore, UB-trees must incorporate such dimensional hierarchies in their clustering of the data to be efficient for OLAP queries.

Hierarchical clustering is a method that controls key values and structures these key values specifically for query access patterns. Hierarchical clustering capitalizes on the fact that many primary key to foreign key relationships are composed of hierarchies. As an example, a customer dimension might contain customers each of which the dimension assigns to one city. The customer dimension might further assign each city to a state and each state to a region. Hierarchical clustering incorporates these hierarchies into the primary keys for the dimensions. The Customer Dimension in Table 3.4 shows such a hierarchy.

3.3.5 Z-ordering and Hierarchical Clustering

MHC combines hierarchical clustering with UB-trees. If customers in an OLAP dimensional hierarchy roll up to a city, a state, and then a region, MHC utilizes all four of

Table 3.4: Customer Dimension

KEY	DESC	CITY	STATE	REGION
1	Johnson	San Diego	CA	West
2	Smith	Atlanta	GA	South
3	Davis	Boston	MA	Northeast
4	Watts	New York	NY	Northeast
5	Duncan	Wilmington	NC	South
6	Reaves	Columbia	SC	South
7	Bradford	Chicago	IL	Midwest

these customer levels, including customer, to cluster the data by the customer dimension in the UB-tree. MHC utilizes this same hierarchical clustering technique for all dimensions participating in the UB-tree. This technique allows MHC to outperform bit map indexes and other non-clustered indexes when supporting OLAP queries. Non-clustered indexes such as bit mapped indexes tend to only work well for very small result sets since non-clustered indexes require an average of one Input/Output (I/O) for each record that they include in result sets [17, 6, 26, 35, 31, 30, 8, 37, 36]. MHC does not have this limitation. It takes advantage of locality in OLAP queries and concentrates records with the same dimensional hierarchy values into much smaller numbers of database blocks thereby increasing query efficiency by reducing I/O and ultimately speeding up queries. Simultaneously, MHC provides symmetrical multidimensional access for any combination of participating dimensions through the use of UB-trees with Z-ordering. This allows MHC to be useful for queries that do not restrict dimensions or keys in the prefix of the underlying B-tree [23].

To combine Z-ordering and hierarchical clustering, MHC uses compound surrogate keys or hierarchical surrogate encoding (HSE) keys in conjunction with Z-ordering. These keys reserve a fixed number of bits for each level in a dimension hierarchy. The fixed number of bits at each level depends on the number of unique key values for all parent keys at that level in the dimension hierarchy. For the Customer dimension if there are 6 regions overall, the maximum number of states in any of the 6 regions is 20, the maximum number of cities in any state is 150, and the maximum number of customers in any city is 17, then the HSE would require 3 bits for the region level, 5 bits for the state level, 8 bits for the city level, and 5 bits for the customer level. Therefore the Customer dimension would require a total of 21 bits for each of its primary keys. Note that each level requires only the number of bits needed to uniquely identify the maximum number of unique values at that level within

Table 3.5: MHC HSE

REGION	STATE	CITY	CUSTOMER
001	10101	00100001	00011

the context of its parent at the next higher level. This greatly reduces the overall length of the compound surrogate in contrast to storing independent primary keys for each hierarchy level. MHC also makes provision for variable length compound surrogate keys in the event that different keys have different numbers of hierarchical levels or unbalanced hierarchies but, MHC does not make provision for variable length bits strings for each parent in each hierarchical level individually. If MHC requires 8 bits for the city level in the customer dimension, then any primary key from the customer dimension that includes the city level must also include all 8 bits [23]. Table 3.5 shows the primary key structure for this Customer Dimension.

Other than dimension key content dictated by HSE, MHC does nothing to alter the properties of host UB-trees and the Z-ordering that it employs. Queries are the same as in UB-trees. MHC implements slice queries by explicitly specifying supplied values in the prefix of compound surrogates and using wildcards for the remaining bits just as it uses wildcard bits for completely unspecified dimensions in multidimensional queries.

Ironically, while the underlying UB-tree is completely dynamic in that it does not have any inherit limitations or preset thresholds, MHC is not. MHC incorporates variable length overall bit strings as dimension keys but uses fixed length bit strings for individual hierarchy levels within the dimension keys [23]. In MHC, the length L of each bit string that represents a hierarchy level is:

$$L = log_2 F$$

For keys not at the highest level of a dimensional hierarchy, F represents the maximum fan-out or largest number of children that any parent key, not just the parent key of the current child, in the next higher level in the dimensional hierarchy could have. If the key is at the highest level in the hierarchy, F simply represents the maximum number of possible keys at that level. MHC incorporates these fan-out values into the physical database design when it defines the dimensional hierarchies [23, 22, 33, 32]. If the actual size of the data that a data warehouse application inserts or loads into an MHC dimensional hierarchy exceeds any of these maximum fan-out values, the MHC dimensions and UB-trees

will require full reorganization. If MHC dictates maximum fan-out values that are much larger than actual data sizes, the physical database design then is likely to adversely affect the symmetry of one or more dimensions. Thus, MHC forces a tradeoff between dynamic or real-time updates and symmetry.

Because MHC uses fixed length bit strings to represent individual levels in HSE, there is a tradeoff between symmetry and dynamic or real-time updates in MHC. If MHC designates too few bits for a given hierarchy level, then the data might exceed this threshold and require full reorganization of all data in all UB-trees that include the affected dimension before MHC can process any further updates. If MHC designates too many bits for a given hierarchy level, then the leading bits of the hierarchy level might remain empty and cause the "puff pastry effect" or symmetry problems.

Chapter 4

MDDC Theory

MDDC removes the tradeoff between symmetry and dynamic updates in MHC with a simple but powerful modification to the MHC key structure. Both MHC and MDDC benefit from their more narrow focus on OLAP queries by encoding compound surrogates into the primary keys of dimensions. With this encoding, MHC imposes a partial ordering and not a full ordering of the data. Specifically, MHC sorts key values within the order of parent keys. In the Customer Dimension example, MHC sorts each CUTOMER within the context of a REGION, STATE, and CITY. MHC might group CUSTOMER 1, 3, and 5 in one REGION, STATE, and CITY and CUSTOMER 2, 4, and 6 in other REGION, STATE, and CITY values depending on data values and relationships. Since hierarchical clustering already forces a partial ordering that orders data according to the dimension hierarchies, MDDC further takes advantage of this tradeoff with no additional cost. Research related to MHC has shown that mirroring the bits of each level in compound surrogates increases entropy and therefore symmetry [22]. MDDC introduces a mechanism for variable length bit strings at each level in HSE keys and uses this in conjunction with bit mirroring. MDDC also employs a third derived bit value that collates before 0 or 1 to mark the end of bit strings in each HSE level so that HSE keys with the same prefix but different HSE level values do not intermingle in the collating sequence. These techniques allow MDDC to be completely dynamic while maintaining symmetry equal to or better than MHC. Specifically, MDDC has superior symmetry when it uses fewer bits in an HSE level than the fixed number of bits that MHC predefines. This is a special problem in all HSE levels other than the highest level since MHC requires the number of bits at an HSE level account for the maximum number of children for any parent at the next higher level in the data. Additionally, the symmetry

Table 4.1: MDDC HSE

REGION	STATE	CITY	CUSTOMER
10000000	10101000	10000100	11000000

of MDDC can never be worse than MHC by definition because MHC must allocate enough bits to cover the maximum distinct values in the level. MDDC will never use more than this maximum number of bits unless there are more distinct values in which case MHC would need to be reorganized with a higher value for maximum number bits at that level. Therefore, MDDC introduces dynamic capabilities not found in MHC, improves upon the symmetry of MHC, and does not introduce any significant tradeoffs.

4.1 Variable Length Bit Strings

Introducing variable length bit strings for each level in the HSE key requires an efficient storage and processing technique. Typically, variable length fields use a length byte or end of string marker such as a NULL byte to indicate their size. This would not make sense for HSE levels since the actual bits that compose the level itself are likely to fit in less than one byte. A better approach is to allocate the last bit of each byte as a continuation bit so that 7 bits of each byte are usable by MDDC. This approach would have to exceed 8 bytes or 56 usable bits in a level of an HSE key before the continuation bit approach becomes less efficient than the length byte or NULL byte approaches. This is not likely to occur in the vast majority of HSE key values since each level needs only be unique in the context of its ancestors. Note also, that each level in a variable length HSE key requires at least one byte and also uses a whole number of bytes for each level. This affects the length of HSE keys that MDDC must store but does not affect symmetry since MDDC does not use the continuation bits or any trailing zeros when interleaving bits. MDDC can discard all trailing zeroes since by definition all reverse bit strings end in 1 with the exception of the single value of 0. It is also important to point out that MDDC might also have one overall variable length byte for the entire HSE key just as MHC does to accommodate unbalanced hierarchies or hierarchies that allow a different number of levels for each key value.

In MDDC, the key from the customer example in the MHC section has an HSE in MDDC as Table 4.1 depicts.

4.2 Dynamic Bit Interleaving

In addition to varying the length of bit strings in HSE keys, MDDC combines HSE keys with dynamic bit interleaving in lieu of Z-ordering. If MDDC were to use Z-ordering and bit strings that preserve an integer ordering as MHC does, the relative position of each bit in these bit strings would need to shift as HSE key lengths change. This would alter the overall Z-ordering and require MDDC to completely reorganize all the data in its entirety. Using mirrored bits in conjunction with variable length bit strings and dynamic bit interleaving overcomes this problem.

No matter how long the HSE bit strings grow, the relative position of bits in the bit interleaving does not change. MDDC simply adds the additional bits for larger level values to the end of the bit string for that level in the HSE. With Z-ordering the depth of all HSE key values for each dimension are the same, for example 3 bits.

With dynamic bit interleaving in contrast, different keys in each dimension are likely to have different depths. Consequently, MDDC does not have a predetermined address space like MHC. MDDC subdivides the address space based on the density of local keys on an as needed basis. Figure 4.1 depicts the uniform depth of Z-ordering while Figure 4.2 depicts the variable depth of dynamic bit interleaving. Dynamic bit interleaving furnishes MDDC with some of the capabilities found in grid files, quad trees, and other similar data structures [42, 13, 12].

Using a third derived bit value that collates before 0 and 1 ensures that MDDC completely segregates distinct HSE key values. This does not alter the relative position of previous interleaved bit addresses in the overall data ordering and therefore does not require data reorganization.

Mirrored bits have one more important advantage. Except for the value of 0, each mirrored bit string ends with a 1 by definition. This allows MDDC to trim the remaining filler bits, which are all zeros, when interleaving bits. With this technique, MDDC always exhibits symmetry as good as MHC and better in some cases. If MHC specifies 8 bits for a given level in a compound surrogate but only uses one, then MHC wastes up to 7 bits before accessing a bit that makes a difference in the Z-ordering. In such cases, some dimensions that are using all bits in their compound surrogates might dominate the dimensions that have unused bits. For the previously listed reasons, MDDC does not include these bits when interleaving bits.

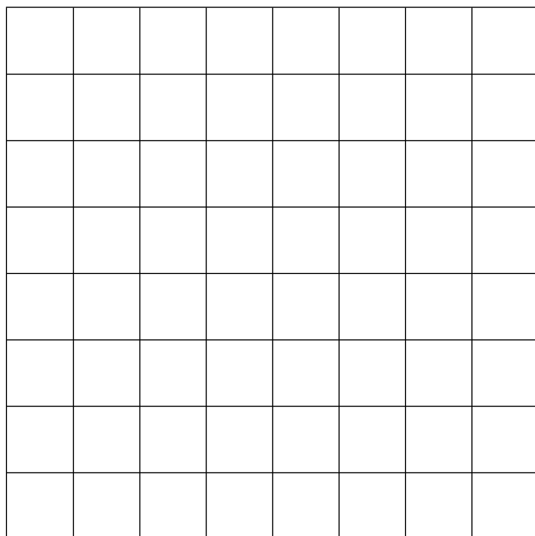

Figure 4.1: Uniform Depth Z-address Space

The following demonstrates how dynamic bit interleaving works. First, dynamic bit interleaving maps all relevant zero and one bits from the mirrored bit string to 10 for bit 0 and 11 for bit 1. Dynamic bit interleaving maps all separators or the third derived bit value to 01 and maps the end of string or NULL to 00. For example, in MDDC the following two bytes, 10100110000010000, with 8 bits for each level maps to 1110111010111101101010101101100. MDDC only uses these double length bit strings to collate keys. MDDC does not store the keys in this format. MDDC stores the keys in the mirrored bit format already described. Once the double bit format is calculated, MDDC can interleave the bits just as Z-order does. This allows MDDC to implement dynamic bit interleaving with minimal additional costs.

4.3 MDDC Operations

In general operations in MDDC are very similar to those in MHC. The following sections provide details for MDDC operations.

4.3.1 Queries in MDDC

MDDC is like MHC in that it only alters the content of keys in B-trees and does not alter any of its other properties. Therefore, queries for MDDC follow the normal rules

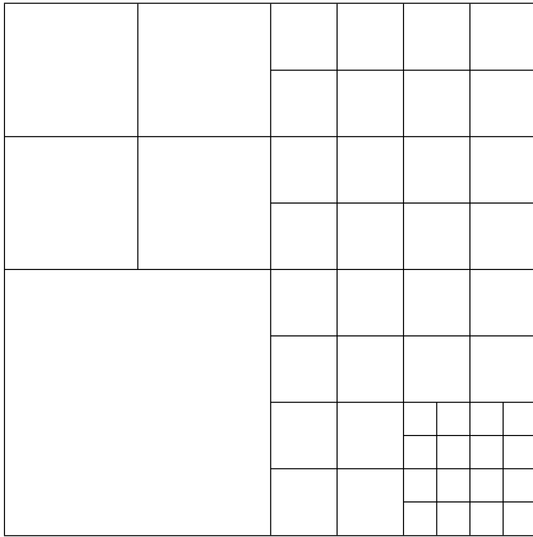

Figure 4.2: Variable Depth Dynamic Bit Interleaving Space

for B-trees. As with MHC, slice queries or partial matches are more complicated in MDDC.

4.3.2 Point Queries

Point queries are very simple in MDDC. When the query specifies a full key for each dimension, MDDC simply computes the single bit interleaved address and uses the B-tree to locate the record or records with the specified key values.

4.3.3 Slice or Partial Match Queries

MDDC uses "wild card" or "don't care" bits in interleaved bit addresses for dimension key values or parts of dimension key values that the query does not specify. This applies to situations where the query completely omits dimensions or where queries only specify dimension prefixes corresponding to levels in compound HSE dimension keys. In this regard, MDDC functions in the same manner as MHC.

4.3.4 Maintenance

Like MHC, MDDC does not alter any properties of B-trees during maintenance operations. MDDC inserts, deletes, and updates records with standard B-tree methods with the caveat that MDDC pre-computes bit interleaved addresses for all participating

dimension keys before comparison with other keys in B-trees. MDDC also pre-computes bit interleaved addresses for all comparison keys in the B-trees as they are needed. The pre-computed, bit-interleaved addresses determine the placement and organization of records within the B-tree. In doing so, MDDC does not in any way alter the properties of the host B-tree. Therefore, B-trees containing keys that MDDC algorithm organizes retain all maintenance advantages of B-trees including the perfect balance, shallow depth, granular concurrency control, and recoverability [40, 13]. Since MDDC does not require complete data reorganization, it is completely dynamic and can readily accommodate real-time updates.

4.3.5 Key Recycling

MDDC like MHC can take advantage of key recycling at all levels of any hierarchy [23]. MDDC works the same as MHC in this area. Realize that HSE keys need not assume the role of primary keys. HSE keys can play the role of internal row addresses in dimension and fact tables. Finally, the dimensions and optionally fact tables can store the primary key values in addition to the HSE keys.

4.3.6 Key Participation

As is the case with MHC, MDDC optionally includes any dimensions and any hierarchy levels that the dimensions contain into the HSE. MDDC can also include any proper subset of these dimensions and levels. When MDDC configures a data structure with a proper subset of dimensions and levels, it appends the HSE key with any remaining dimensions or other fields in the primary key to enforce uniqueness. MDDC can also combine more than one independent dimension into one virtual dimension via concatenation to establish priority.

4.4 Incremental Costs

As stated before, the only relevant tradeoff that MDDC makes so that it can provide dynamic symmetry is the forfeiture of integer ordering. But since MHC forces a partial integer ordering anyway and the most common OLAP queries work well with such a hierarchical partial ordering, the partial ordering with the variable length mirrored bits

from MDDC works just as well for OLAP queries [19] and there is no real tradeoff in this regard. Hence, MDDC supports OLAP point and OLAP slice queries just as well or better than MHC.

The only other cost that MDDC suffers in comparison to MHC is the requirement for more disk space. The total amount of additional disk space per table depends on the ratio of key fields to other fields in the table. However, since both MHC and MDDC, provide efficient query access without additional indexes, they both are very space efficient.

4.5 Summary of MDDC

By utilizing variable length HSE keys and dynamic bit interleaving, MDDC enables completely dynamic insert, update, and delete operations without full reorganization of any dimension, fact table, or underlying B-tree. In addition, this allows MDDC to provide symmetrical query efficiency that is good as or better than that of MHC. The only tradeoff is the requirement for additional disk space. The experimental results that follow will substantiate these claims.

Chapter 5

Implementation

The implementation here contains working versions of both MDDC and MHC so that the experiments that follow can compare the two algorithms directly. The purpose of the implementation is to demonstrate that MDDC as opposed to MHC is completely dynamic, has symmetry that is good as or better than MHC, at minimum additional costs. The following sections describe the simulation software, simulation parameters, simulated SQL tables, and simulated database tables.

5.1 Simulation Platform

This implementation is built on a Microsoft Windows laptop. The laptop is a Hewlett-Packard Pavilion dv5000. Its RAM is 1 Gigabyte, it has a single 84.4 Gigabyte hard disk drive, and its single AMD 64 bit processor has a clock speed of 1.79 Gigahertz. Its operating system is Windows XP Professional.

5.2 Simulation Software

In the experiments that follow, the implementation of MDDC and MHC uses the B-tree access method in BDB (Berkeley DB) software from Sleepy Cat Inc. BDB is open source software. The implementation here specifically implements MDDC and MHC with the C language API from BDB. All code for the implementation of MDDC in this study is in C language. The simulation uses Microsoft Visual Studio.net Professional to compile the C language code.

In order to implement MDDC and MHC with the standard B-tree access methods in BDB, this implementation alters the comparison routine of the B-tree access method in BDB to compute bit interleaved addresses as part of the comparison process between composite primary keys. This further evidences the fact that MDDC does not require alterations to the host B-tree data structure. The implementation additionally contains programs to search the MDDC and MHC data structures in a skip sequential manner. To simplify the code for the implementation, the simulation only permits queries that specify at most one value for each dimension level. If the queries do not specify a value for a given dimension level, the implementation allows all values and uses wild card bits for these dimension levels. Finally, if a query does not specify a value for a dimension level then the query cannot specify any specific values for the children of this dimension level.

The query output of the simulation on MDDC and MHC consists of record counts for each query and access statistics including cache size, records scanned, total I/O problems, I/O probes to disk, processor time, and elapsed time. The simulation also includes a full scan program for MDDC and MHC to ensure query results are correct.

5.3 Simulation Parameters

This implementation only uses a small amount of RAM for MDDC and MHC so that it can demonstrate the I/O efficiency of MDDC. This implementation limits the BDB cache size parameters to 4 database blocks. It also uses large database blocks for I/O efficiency. It uses the maximum 64 kilobyte block size in BDB. Therefore the size of the cache for MDDC and MHC is 256 kilobytes.

5.4 Simulated SQL Statements

Since this implementation uses C API for the BDB embedded database management system, no SQL interpreter is available. Therefore, C programs simulate SQL statements in this implementation. These SQL statements constrain dimensions directly. The simulation does not perform join operations since join operations would be the same for both MHC and MDDC. Similarly, the simulation does not perform range queries since they are not typical in OLAP applications and MHC or MDDC are not designed for them. [19, 21, 23, 22] The SQL query in Figure 5.1 is an example of this concept.

```
SELECT COUNT(*)
FROM GROCERY_SALES_FACT WHERE PROD_CAT_ID=4 AND
PROD_ID=8
```

Figure 5.1: Simulated SQL Query

5.5 Simulated Database Tables

Typically, relational database systems use SQL to define tables. As with all other SQL statements, this simulation uses C programs to define tables.

5.6 Special Simulation Challenges

In order to take full advantage of MHC and MDDC and to compare them fairly, a query algorithm is required that visits each leaf database block only one time. This is fairly simple in a one dimensional B-tree. Once the query accesses a given leaf block, the algorithm access all the records on that block and ensures that the next key value accessed is greater than the last key in the current block [40]. It is more complicated to effect this type of skip sequential processing in MDDC and MHC where the algorithms must deal with multidimensional key encodings and bit interleavings. The algorithm must calculate the next key in the multidimensional space using wildcard bits for partial match and slice queries. Essentially, the lowest priority bit for each dimension that has an unset value less than 1 has to be identified and then the unset bit with the lowest overall priority from all the dimensions not in the query filter has to be set to 1. This algorithm proved to be quite complex and time consuming to implement for both MHC and MDDC.

Chapter 6

Experimental Results

6.1 National Grocery Chain

The "National Grocery Chain" database contains sales data for a national chain of grocery stores broken out by month, and product. The store, month, and product data were fabricated and manually entered into text files. The sales data was created with the assistance of a random number generator. Given a sparseness factor such as 10% the simulator uses the random number generator to create one randomly selected record out of every 10 possible combinations of store, month, and product. A sparseness factor of approximately 0.28% was selected for this experiment to generate the approximate $12,000,000$ sales records in the simulation.

6.1.1 Data Model

The "National Grocery Chain" database is a flat star-schema or one that contains the complete dimension hierarchy in the base dimension [19]. The dimensions are Store in table B.1, Product in table B.2, Promotion in table B.3, and Month in table B.4. This database combines these four dimension tables to establish the grain of the Sales Facts in table B.5.

6.1.2 Test Criteria

The experiment focuses on two dimensions. It tests the behavior of the product dimension as data is loaded. It also uses the month dimension as an experimental control.

The product dimension contains a hierarchy made up of two levels, product category and product. The MHC key as tables B.2 and B.4 depict in this simulation uses 8 bits for the product category level and 8 bits for the product level. The month dimension contains a hierarchy made up of three levels, year, quarter, and month. The MHC key in this simulation uses 6 bits for year, 4 bits for quarter, and 4 bits for month. MDDC has no preset number of bits for any of the dimensions.

6.1.3 Data Loading

A set of C programs make calls to the BDB database in order to load data in the "National Grocery Chain" database. The C programs load the data into the MHC database and the MDDC database. The C programs load the "Grocery Sales Fact" table in B.5 in stages to illustrate the differences in symmetry between MDDC and MHC and the need to reorganize MHC data structures when the MHC algorithm reaches preset size thresholds.

The simulation loads all simulated records with product category keys 1 and 2 in stage 1. In stages 2 through 9, the simulation loads all simulated records with product category keys 3 through 10 respectively as results tables D.1, D.2, D.3, D.4, and D.5 depict.

6.1.4 Queries

For simplicity in the "National Grocery Chain" database, the C programs that implement queries, only allow point queries and slice queries with no more than one dimension value from any hierarchical level of any dimension. The programs do not allow specification of multiple values from any level in any dimensional hierarchy in one query. This allows for simple and efficient query implementation. The implementation could include multiple dimension values at a higher implementation cost and complexity in order to maintain the same level of efficiency. The theory and principles of MDDC and its comparison to MHC are the same for either level of implementation complexity. This is true since skip sequential processing handles multiple points in key order very efficiently [40].

To further focus on the MDDC versus MHC data structures, this simulation concentrates on queries that specify dimension and hierarchy values that MDDC and MHC use to organize the fact data or those that are in the hierarchical surrogate encoding (HSE) keys. The simulation executes all the queries represented by the SQL in Table C.1 against the implementation. The simulation runs all queries for both the MHC and MDDC databases

after each stage in the loading process represented by the corresponding product category values.

6.1.5 Results

The results demonstrate the advantage that MDDC has over MHC in symmetry and real-time update capability. The first 8 queries from Table C.1 represent the differences in MHC and MDDC most starkly since each one of these queries only constrains one of the dimensions. In this experiment, the product dimension is the experimental test whereas the other three dimensions are controlled tests. Therefore query 1 and 5 from Table C.1 should illustrate the results that are sought while queries 2 through 4 and 6 through 8 should perform as expected.

All the results demonstrated that the most persistent and reliable indicator of performance is the number of I/O operations for each query, whether they are cached or not. In addition, when a ratio of the number of I/O operations per index query versus the number of I/O operations per full scan query is used as the primary metric, then the results are stable as the underlying data increases in size. This ratio also accounts for the difference in file sizes between MHC and MDDC. As a result, the ratio of query I/O to full scan I/O is the primary metric on which this experiment focuses.

Charts 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 compare the MHC results to the MDDC results for all 9 product category data runs in this experiment. Note that charts 6.1 and 6.5 represent the greatest difference between MHC and MDDC queries. Of course these correspond exactly to test queries 1 and 5 from Table C.1. In these queries, the MHC ratio of query I/O to full scan I/O or percentage of full scan actually indicates that query I/O's in MHC are sometimes larger than the number of I/O operations in a full table scan. Note also that the ratio improves as the simulation loads more of the product category records. As charts 6.2, 6.3, 6.4, 6.6, 6.7, and 6.8 depict, MDDC queries have slightly higher I/O ratios than MHC queries in the control queries but are much lower than the 100 percent ratio required to scan the full table.

These results are to be expected since the HSE key for product category in the product dimension for the MHC database allocates 5 leading bits that this simulation does not use. A new database design for MHC could alleviate this problem. If the simulation uses 4 bits for the product category key in the simulation in lieu of 8 bits for the product

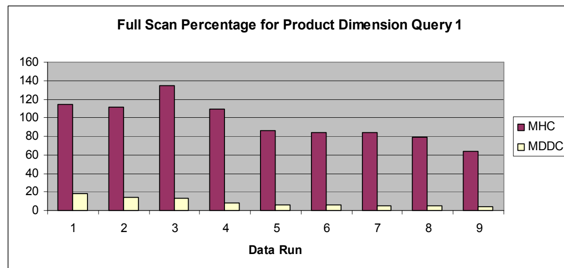

Figure 6.1: Grocery Sales Product Dimension Query 1

category in the MHC database then the above problem with the product dimension will not occur. But, this presents another problem. The MHC database would not be able to house more than 16 product categories without a complete MHC database reorganization. the slightly higher I/O ratios in the control queries are also to be expected since the product dimension in MDDC is sharing in some of the selectivity in the MDDC data structure at the expense of the other 3 dimensions while the product dimension is not doing this in the MHC data structure.

Hence, MDDC offers better symmetry than MHC and unlike MHC offers real-time updates without major data structure reorganizations. Unlike MHC, MDDC does not force a tradeoff between symmetry and intrinsic growth limits.

6.2 TPC-H

The data in this experiment originates from the TPC-H benchmark [1]. The DBGEN from TPC-H generates the database with a scale factor of 2 so that the LINEITEM fact table has approximately 12,000,000 records. See TPC-H estimated database size for a scale factor of 2 in B.6. This simulation uses the PART dimension as shown in table B.7 and the SUPPLIER dimension as shown in table B.8. This simulation uses the LINEITEM

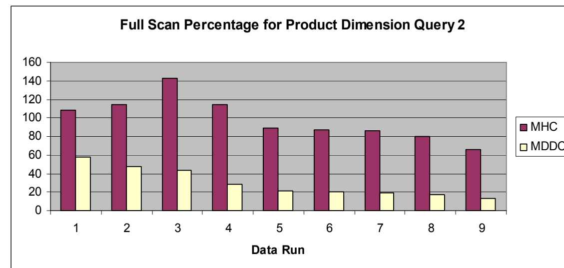

Figure 6.2: Grocery Sales Product Dimension Query 2

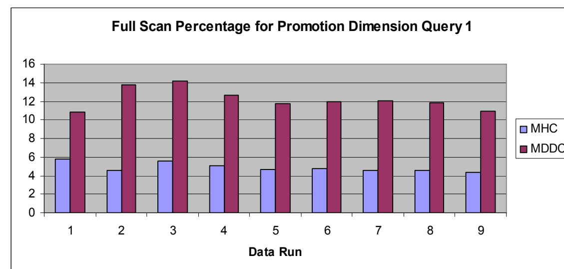

Figure 6.3: Grocery Sales Promotion Dimension Query 1

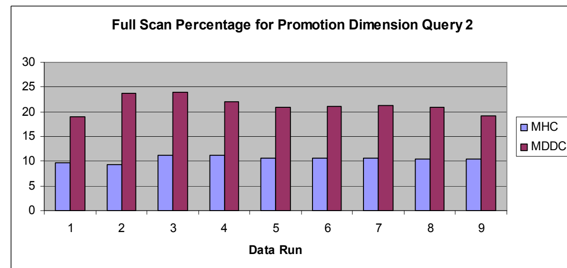

Figure 6.4: Grocery Sales Promotion Dimension Query 2

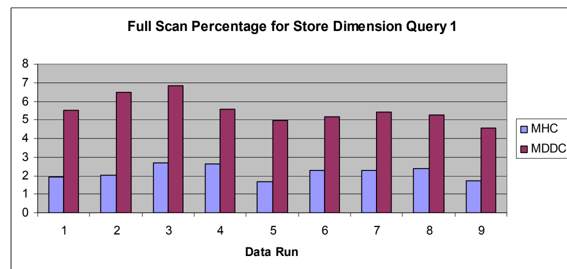

Figure 6.5: Grocery Sales Store Dimension Query 1

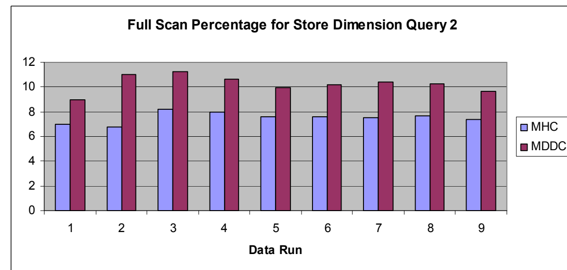

Figure 6.6: Grocery Sales Store Dimension Query 2

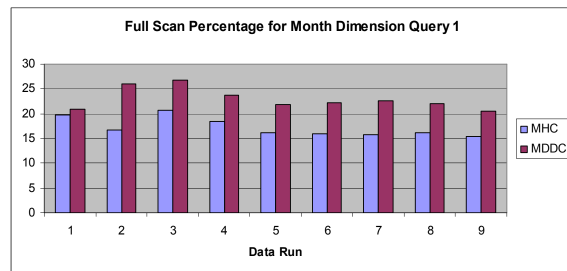

Figure 6.7: Grocery Sales Month Dimension Query 1

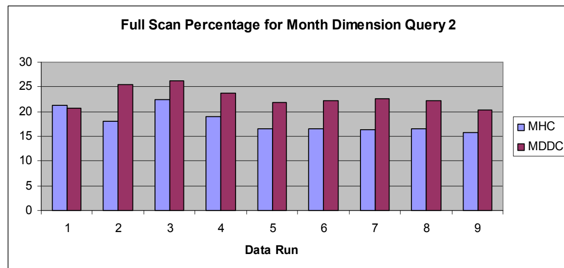

Figure 6.8: Grocery Sales Month Dimension Query 2

table as shown in table B.9 for the fact table. The MHC and MDDC data structures cluster the LINEITEM data with only the PART and SUPPLIER dimensions.

6.2.1 Data Model

The TPC-H database is a normalized schema or a schema with snow-flaked dimensions and fact tables that are normalized into more than one table [1, 19, 5]. This simulation uses PART in B.7 and SUPPLIER in B.8 as the dimensions. It combines these two dimensions to establish the granularity of the LINEITEM table in B.9.

6.2.2 Test Criteria

The experiment focuses on two dimensions. It tests the behavior of the SUPPLIER dimension as data is loaded. It also uses the PART dimension as an experimental control. The SUPPLIER and PART dimensions each contain a hierarchy made up of one level, the primary key. The MHC key as tables B.7 and B.8 depict uses 19 bits for the PARTHSE key and 16 bits for the SUPPHSE key. MDDC has no preset number of bits.

6.2.3 Data Loading

A set of C programs make calls to the BDB database in order to load and update data in the TPC-H database. The C programs load the LINEITEM table in B.9 in stages to illustrate the differences in symmetry between MHC and MDDC and the need to reorganize MHC data structures when the MHC algorithm reaches preset size thresholds.

The simulation loads all simulated records with supplier keys 1 through 4000 in stage 1. In stages 2 through 9, the simulation loads all records with supplier keys 4001 through 20000 as results Tables D.6, D.7, D.8, D.9, and D.10 depict.

6.2.4 Queries

For simplicity in the TPC-H database, the C programs that implement queries, only allow point queries and slice queries with no more than one dimension value from any hierarchical level of any dimension. The programs do not allow specification of multiple values from any level in any dimensional hierarchy in one query. This allows for simple and efficient query implementation. The implementation could include multiple dimension values at a higher implementation cost and complexity in order to maintain the same level of efficiency. The theory and principles of MDDC and its comparison to MHC are the same for either level of implementation complexity. This true since skip sequential processing handle multiple points in key order very efficiently [40].

To further focus on the MHC versus MDDC data structures, this simulation concentrates on queries that specify dimension and hierarchy values that MHC and MDDC use to organize the fact data or those that are in the HSE keys.

The simulation executes all the queries represented in Table C.2.

6.2.5 Results

Once again, the results demonstrate the advantage that MDDC has over MHC in symmetry and real-time update capability. Since all 20 queries in this experiment from Table C.2 represent a query against the PART dimension or against the SUPPLIER dimension, each is either a control or test query. Therefore, this experiment averages all the metrics of the first 10 queries in Table C.2 together to represent the PART dimension or the control and the second 10 queries in Table C.2 represent the SUPP dimension or the test in this

experiment. The average of the second 10 queries should substantiate the claims of this dissertation while the first 10 should perform as expected.

As in the previous experiment, all the results demonstrated that the most persistent and reliable indicator of performance is the number of I/O operations for each query, whether they are cached or not. In addition, when a ratio of the number of I/O operations per index query versus the number of I/O operations per full scan query is used as the primary metric, then the results are stable as the underlying data increases in size. This ratio also accounts for the difference in file sizes between MHC and MDDC. As a result, the ratio of query I/O to full scan I/O is the primary metric on which this experiment focuses.

Charts 6.9 and 6.10 compare the MHC results to the MDDC results for all 20000 SUPP_ID's in the 9 data runs in this experiment. Note that chart 6.10 represents the greatest difference between MHC and MDDC queries. Of course this corresponds exactly to test queries 10 through 20 from Table C.2. In these SUPPLIER dimension queries, the MHC ratio of query I/O to full scan I/O or average percentage of full scan is much closer to 100% of the full table than the PART dimension queries. Note also that the ratio improves as the simulation loads more of the SUPP_ID's. As chart 6.9 depicts, MDDC queries have slightly higher I/O ratios than MHC queries in the control queries but are much lower than the 100 percent ratio required to scan the full table.

These results are also to be expected since the HSE key for SUPP_ID in the SUPPLIER dimension for the MHC database allocates 16 bits which can store up to 65536 distinct SUPP_ID values. In this case, a new database design for MHC could not alleviate this problem. If the simulation uses 13 bits for the SUPP_ID in the simulation in lieu of 16 bits for the SUPP_ID, it could not store all 20000 SUPP_ID's. Even with all 16 bits, the MHC data structure can not store anymore than the 65536 SUPP_ID's. Therefore, to load any TPC-H scale factor larger than 2, the database must completely reorganize the MHC data structure. This is not the case with the MDDC data. The database can continue to load records into the current data structure. The slightly higher I/O ratios in the control queries are also to be expected since the SUPPLIER dimension in MDDC is sharing in some of the selectivity in the MDDC data structure at the expense of the PART dimension while the SUPP dimension is not doing this in the MHC data structure in the first few data runs. Notice that as the number of distinct SUPP_ID's approaches 20000, the MHC data structure behaves more like the MDDC data structure. This demonstrates the ability of the MDDC data structure to maintain more constant symmetry than the MHC

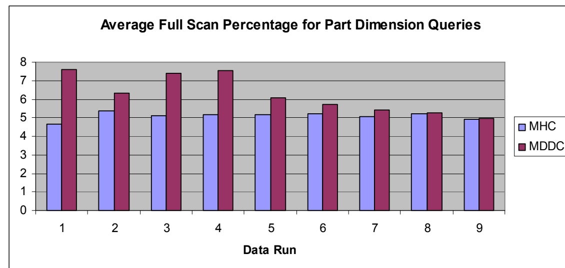

Figure 6.9: TPC-H Part Dimension Queries

data structure while also retaining real-time update capabilities. Once again, MDDC offers better symmetry than MHC and unlike MHC offers real-time updates without major data structure reorganizations. Unlike MHC, MDDC does not force a tradeoff between symmetry and intrinsic growth limits.

Hence, MDDC again in this experiment offers symmetry and dynamic or real-time updates without major data structure reorganizations whereas MHC does not.

6.3 Final Evaluation of MDDC and MHC

As the previous experiments illustrate, MDDC generally improves upon MHC in the support of OLAP queries and associated database update operations. MDDC provides symmetry that is better in some cases than MHC. MDDC also provides a real-time update capability not present in MHC. As a result, MDDC eliminates the tradeoff that exists between symmetry and the ability to update data without a reorganization of the database. Finally, MHC does not introduce any prohibitive costs.

MDDC provides better symmetry for dynamic data than MHC. When data grows and especially when the length of Hierarchical Surrogate Encodings (HSE's) are increasing and decreasing, MHC must utilize bit strings that are long enough to handle maximum

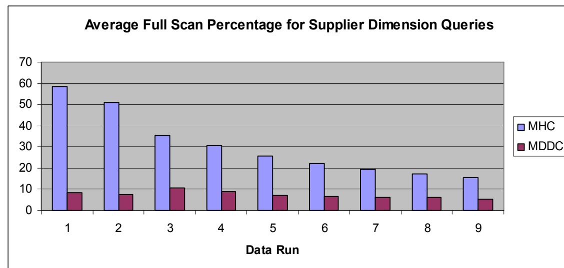

Figure 6.10: TPC-H Supplier Dimension Queries

additional growth. These preset HSE bit string lengths can destroy symmetry. If an HSE dimension level specifies too many leading bits that MHC does not use, then MHC can starve that dimension level in the collating and clustering process. MHC does not work as well when some HSE dimension levels specify too many bits.

Additionally, MHC is not as dynamic as MDDC. Even with long HSE bit strings, MHC by definition contains upper limits on HSE dimension level values. Once data exceeds these MHC limits, MHC must reorganize the data structure in its entirety. To maintain integrity, the database management systems (DBMS) must also reorganize all related dimensions. Thus, as data grows, MHC must perform data reorganizations that prohibit it from supporting real-time OLAP databases.

The resulting MDDC tradeoffs are negligible. MDDC consumes more database space than MHC and does not efficiently handle conventional range queries. By the definition of MDDC, it can consume up to one additional byte per level per dimension. In the experiments, MDDC consumes no more than 10% additional space than MHC. It is also important to consider that since MDDC like MHC does not require any additional indexes, MDDC is very space efficient. MDDC does not support conventional range queries efficiently since it does not store data in native key order. But, MHC has no advantage in

this area since it stores the data in a partial order in lieu of full native key order. MHC only stores data in order within the HSE hierarchies. In addition, range queries do not improve joins since join operations access data one tuple at the time and order does not matter as long as keys from multiple tables to be joined are in the same order. Range queries also are not typical in OLAP applications and databases [19].

Therefore, MDDC works as well as MHC but adds dynamic symmetry and real-time update capabilities.

Chapter 7

Conclusions and Future Work

The real-time OLAP problem involves fact tables with multiple dimension keys such that each dimension key represents one hierarchy from one dimension. Queries can restrict data by any combination of dimensions keys. The goal is to provide efficient query performance in proportion to the number of dimensions and number of dimension keys the query restricts regardless of which combination of dimensions that queries chooses. The real-time OLAP problem also requires that insert, update, and deletion operations be completely dynamic without the need for data structure reorganizations due to preset key limits. In short, the problem is to provide a B-tree like solution for OLAP applications such any combination of restricted dimensions perform as if they compose a prefix of a composite key in a B-tree.

MDDC provides dynamic, real-time updates and efficient symmetry for OLAP queries and is implemented on top of a standard B-tree. MDDC does not require data reorganizations, demonstrates suitability for the most common OLAP queries, and has the capability to reduce the size and number of indexes and materialized views. This dissertation outlines the basic concepts of MDDC. It also illustrates the effectiveness of MDDC with two experiments. The first experiment compares MHC and MDDC with simulated data from a “National Grocery Chain” and the second experiment compares MHC and MDDC with the TPC-H benchmark database. In both experiments this dissertation proves that MDDC always provides symmetry at least as good as MHC and better under some scenarios. Perhaps more importantly, the experiments prove that MDDC dynamically adjusts as the size and structure of the multidimensional hierarchical data changes without the data reorganizations or symmetry problems that MHC causes. Finally, the theory of

MDDC and associated experiments demonstrate that MDDC does not have any built-in key limits where as MHC does. Thus, we present MDDC as a practical solution to the problem of real-time OLAP.

Future work includes seeking a journal publication with published experimental results, improving the symmetry of MDDC, and implementing MDDC in a commercially available open source code database system such as MySQL or PostgreSQL.

Bibliography

[1] The transaction processing performance council.

[2] Rudolf Bayer. The universal B-tree for multidimensional indexing: Genernal concepts. In *World-Wide Computing and Its Applications '97*, pages 10–11, 1997.

[3] E. Bertino and W. Kim. Indexing techique for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, pages 196–214, 1989.

[4] Bishwaranjan Bhattacharjee, Leslie Cranston, Tim Malkemus, and Sriram Padmanabhan. Boosting query performance: Multidimensional clustering. *DB2 Magazine*, 2003.

[5] Pedro Bizarro and Henrique Madeira. Adding a performance-oriented perspective to data warehouse design. *International Conference on Data Warehousing and Knowledge Discovery*, pages 232–244, 2002.

[6] W.A. Burkhard. Hashing and trie algorithms for partial match retrieval. *ACM TODS*, 1(2):175–187, June 1976.

[7] S.S. Conn. OLTP and OLAP data integration: a review of feasible implementation methods and architectures for real time data analysis. In *SoutheastCon,2005.Proceedings.IEEE*, pages 515–520, April 2005.

[8] R.J. Enbody and H.C. Du. Dynamic hashing schemes. *ACM Computing Surveys (CSUR)*, 20(2):850–113, June 1988.

[9] George Evangelidis, David Lomet, and Betty Salzberg. The hB-tree: a Multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Journal*, 6:1–25, 1997.

[10] Christos Faloutsos. Multiattribute hashing using Gray codes. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 227–238, 1986.

[11] Verde Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[12] Hector Garcia-Molina, Jeffery D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, 07458, 2002.

[13] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, 1993.

[14] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM*, 6:47–57, 1984.

[15] Hinrichs and Klaus. Implementation of the grid file: Design concepts and experience. *BIT*, pages 569–592, 1985.

[16] Vinit Jain and Ben Shneiderman. Data structures for dynamic queries: An analytical and experimental evaluation. In *Advanced Visual Interfaces*, pages 1–11, 1994.

[17] James B. Rothnie Jr. and Tomas Lozano. Attribute based file organization in a paged memory environment. *Communications of the ACM*, 17(2):63–69, February 1974.

[18] Jonas S. Karlsson and Martin L. Kersten. Omega-storage: A self organizing multi-attribute storage technique for very large main memories. Technical Report INS-R9910, 30, 1999.

[19] Ralph Kimball, Laura Reeves, Margy Ross, and Warren Thornthwaite. *The Data Warehouse Lifecycle Toolkit*. John Wiley & Sons Inc., New York, Chichester, Weinheim, Brisbane, Singapore, Toronto, 1998.

[20] Jukka Kiviniemi, Antoni Wolski, Antti Pesonen, and Johannes Arminen. Lazy aggregates for real-time OLAP. In *Data Warehousing and Knowledge Discovery*, pages 165–172, 1999.

[21] Volker Markl. The Tetris-Algorithm for Sorted Reading from UB-tree. In *Grundlagen von Datenbanken*, pages 89–93, 1998.

[22] Volker Markl. Mistral - processing relational queries using a multidimensional access technique. *Datenbank Rundbrief*, 26:24–25, 2000.

[23] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *Proceedings of Ideas'99 in Montreal Canada*, 1999.

[24] Michael Martin and Rada Chirkova. Implementing real-time OLAP with MDDC (multidimensional dynamic clustering). In *17th Annual IRMA International Conference*, pages 666–667, Washington, D.C., 2006. International Research Management Association.

[25] Donald R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *JACM*, 15(4):514–534, October 1968.

[26] Jussi Myllymaki and James Kaufman. Locus: A testbed for dynamic spatial indexing. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 5(1):19–50, 1976.

[27] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984.

[28] J.A. Orenstein and T.H. Merritt. A class of data structures for associative searching. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 181–190, April 1984.

[29] Jack A. Orenstein. Spatial query processing in an object-oriented database system. *Communications of the ACM*, pages 326–333, 1986.

[30] Ekow J. Otoo. A multidimensional digital hashing scheme for files with composite keys. *ACM*, 5:214–226, 1985.

[31] Mohamed Ouksel and Peter Scheuermann. Storage mappings for multidimensional dynamic linear hashing. pages 90–105, 1983.

[32] Roland Pieringer, Volker Markl, Frank Ramsak, and Rudolf Bayer. HINTA: A linearization algorithm for physical clustering of complex OLAP hierarchies. In *Design and Management of Data Warehouses*, page 11, 2001.

[33] Pieringer Elhardt Ramsak. Transbase: A leading-edge rolap engine supporting multidimensional indexing and hierarchy clustering. 2003.

[34] Regnier and Mireille. Analysis of grid file algorithms. *BIT*, pages 335–357, 1985.

[35] Ronald L. Rivest. Partial-match retrieval algorithms. *SIAM Journal of Computing*, 5(1):19–50, 1976.

[36] Doron Rotem. Clustered multiattribute hash files. *ACM*, 3:225–, 1989.

[37] Moni Naor Russell Impagliazzo. Efficient cryptographic schemes provably secure as subset sum. *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, 1989.

[38] Hanan Samet. Data structures for quadtree approximation and compression. *Commun. ACM*, 28(9):973–993, 1985.

[39] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. Strategies for processing ad hoc queries on large data warehouses. *ACM*, 5:72–79, November 2002.

[40] Alan L. Tharp. *File Organization and Processing*. John Wiley & Sons, New York, Chichester, Brisbane, Toronto, Singapore, 1988.

[41] Volker Markl, Michael G. Bauer, and Rudolf Bayer. Variable UB-trees: An efficient way to accelerate OLAP queries. In *GI-Workshop Data Mining und Data Warehousing als Grundlage Moderner Entscheidungsunterstutzender Systeme*, pages 79–88, 1999.

[42] Carlo Zaniolo and Stefano Ceri. *Advanced Database Systems*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.

Appendix A

Introduction to Appendices

These appendices contain supplemental information pertaining to the experiments in this dissertation. These appendices contain complete table definitions, queries, and results for the experiments. These appendices also contain tables that this dissertation uses as examples.

Appendix B

Tables definitions for experiments

Table B.1: Grocery store dimension

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
ID	Integer	Store Identifier
NAME	Character	Store Name
STATE	Character	State Name
REGION	Character	Region Name

Table B.2: Grocery product dimension

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
ID	Integer	Product Identifier
NAME	Character	Product Name
CATEGORY	Character	Product Category Name

Table B.3: Grocery promotion dimension

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
ID	Integer	Promotion Identifier
NAME	Character	Promotion Name
CATEGORY	Character	Promotion Category Name

Table B.4: Grocery month dimension

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
ID	Integer	Month Identifier
MONTH	Character	Month Description
QUARTER	Character	Quarter Description
YEAR	Character	Year Description

Table B.5: Grocery sales fact

FIELD	TYPE	DESCRIPTION
STORE	Binary	STORE HSE Key (MHC or MDDC)
PRODUCT	Binary	PRODUCT HSE Key (MHC or MDDC)
PROMOTION	Binary	PROMOTION HSE Key (MHC or MDDC)
MONTH	Binary	MONTH HSE Key (MHC or MDDC)
SALES	Integer	SALES METRIC

Table B.6: TPC-H estimated database size

Table Name	Rows	Row Length (in Bytes)	Typical Table Size (in MB)
SUPPLIER	10,000	159	2
PART	200,000	155	30
LINEITEM	6,000,000	112	641

Table B.7: TPC-H PART dimension

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
P_PARTKEY	Integer	Part Identifier
P_NAME	Character	Part Name
P_MFGR	Character	Part Manufacturer
P_BRAND	Character	Part Brand
P_TYPE	Character	Part Type
P_SIZE	Integer	Part Size
P_CONTAINER	Character	Part Container
P_RETAILPRICE	Fixed Decimal	Part Retail Price
P_COMMENT	Character	Part Comment

Table B.8: TPC-H SUPPLIER dimension

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
S_SUPPKEY	Integer	Supplier Identifier
S_NAME	Character	Supplier Name
S_ADDRESS	Character	Supplier Address
S_NATIONKEY	Integer	Nation Identifier
S_PHONE	Character	Supplier Phone
S_ACCTBAL	Fixed Decimal	Account Balance
S_COMMENT	Character	Supplier Comment

Table B.9: TPC-H LINEITEM fact

FIELD	TYPE	DESCRIPTION
HSE	Binary	HSE Key (MHC or MDDC)
L_PARTHSE	Binary	Lineitem HSE
L_SUPPHSE	Binary	Supplier HSE
L_ORDERKEY	Integer	Order Identifier
L_PARTKEY	Integer	Part Identifier
L_SUPPKEY	Integer	Supplier Identifier
L_LINENUMBER	Integer	Line Number
L_QUANTITY	Fixed Decimal	Quantity
L_EXTENDEDPRICE	Fixed Decimal	Extended Price
L_DISCOUNT	Fixed Decimal	Discount
L_TAX	Fixed Decimal	Tax
L_RETURNFLAG	Character	Return Flag
L_LINESTATUS	Character	Line Status
L_SHIPDATE	Date	Ship Date
L_COMMITDATE	Date	Commit Date
L_RECEIPTDATE	Date	Receipt Date
L_SHIPINSTRUCT	Character	Shipping Instructions
L_SHIPMODE	Character	Ship Mode
L_COMMENT	Character	Line Item Comment

Appendix C

Queries for experiments

Table C.1: Grocery sales experiment - queries

Query	SQL
1	$select\ count(*)\ from\ sales$ $where\ prd_cat_id = 1\ and\ prd_id = 4$
2	$select\ count(*)\ from\ sales$ $where\ yr_id = 5\ and\ qtr_id = 1$
3	$select\ count(*)\ from\ sales$ $where\ pro_cat_id = 4\ and\ pro_id = 4$
4	$select\ count(*)\ from\ sales$ $where\ reg_id = 3\ and\ st_id = 6$
5	$select\ count(*)\ from\ sales$ $where\ prd_cat_id = 2\ and\ prd_id = 8$
6	$select\ count(*)\ from\ sales$ $where\ yr_id = 4\ and\ qtr_id = 2$
7	$select\ count(*)\ from\ sales$ $where\ pro_cat_id = 5\ and\ pro_id = 3$
8	$select\ count(*)\ from\ sales$ $where\ reg_id = 7\ and\ st_id = 4$
9	$select\ count(*)\ from\ sales$ $where\ prd_cat = 2\ and\ prd_id = 1\ and\ yr_id = 10\ and\ qtr_id = 4$
10	$select\ count(*)\ from\ sales$ $where\ prd_cat = 2andprd_id = 1\ and\ reg_id = 3\ and\ st_id = 2$
11	$select\ count(*)\ from\ sales$ $where\ prd_cat = 2\ and\ prd_id = 1\ and\ pro_cat_id = 3\ and\ pro_id = 5$
12	$select\ count(*)\ from\ sales$ $where\ yr_id = 10\ and\ qtr_id = 4\ and\ reg_id = 3\ and\ st_id = 12$
13	$select\ count(*)\ from\ sales$ $where\ yr_id = 10\ and\ qtr_id = 4\ and\ pro_cat_id = 3\ and\ pro_id = 5$
14	$select\ count(*)\ from\ sales$ $where\ reg_id = 3\ and\ st_id = 6\ and\ pro_cat_id = 3\ and\ pro_id = 5$
15	$select\ count(*)\ from\ sales$ $where\ prd_cat = 2\ and\ prd_id = 24\ and\ yr_id = 6\ and\ qtr_id = 2$
16	$select\ count(*)\ from\ sales$ $where\ prd_cat = 2\ and\ prd_id = 24\ and\ reg_id = 3\ and\ st_id = 1$
17	$select\ count(*)\ from\ sales$ $where\ prd_cat = 2\ and\ prd_id = 24\ and\ pro_cat_id = 1\ and\ pro_id = 4$
18	$select\ count(*)\ from\ sales$ $where\ yr_id = 6\ and\ qtr_id = 2\ and\ reg_id = 3\ and\ st_id = 1$
19	$select\ count(*)\ from\ sales$ $where\ yr_id = 6\ and\ qtr_id = 2\ and\ pro_cat_id = 1\ and\ pro_id = 4$
20	$select\ count(*)\ from\ sales$ $where\ reg_id = 3\ and\ st_id = 1\ and\ pro_cat_id = 1\ and\ pro_id = 4$

Table C.2: TPC-H experiment - queries

Query	SQL
1	$select\ count(*)\ from\ lineitems\ where\ part_id = 21$
2	$select\ count(*)\ from\ lineitems\ where\ part_id = 55$
3	$select\ count(*)\ from\ lineitems\ where\ part_id = 71$
4	$select\ count(*)\ from\ lineitems\ where\ part_id = 98$
5	$select\ count(*)\ from\ lineitems\ where\ part_id = 108$
6	$select\ count(*)\ from\ lineitems\ where\ part_id = 299$
7	$select\ count(*)\ from\ lineitems\ where\ part_id = 407$
8	$select\ count(*)\ from\ lineitems\ where\ part_id = 511$
9	$select\ count(*)\ from\ lineitems\ where\ part_id = 604$
10	$select\ count(*)\ from\ lineitems\ where\ part_id = 1011$
11	$select\ count(*)\ from\ lineitems\ where\ supp_id = 1$
12	$select\ count(*)\ from\ lineitems\ where\ supp_id = 9$
13	$select\ count(*)\ from\ lineitems\ where\ supp_id = 14$
14	$select\ count(*)\ from\ lineitems\ where\ supp_id = 67$
15	$select\ count(*)\ from\ lineitems\ where\ supp_id = 201$
16	$select\ count(*)\ from\ lineitems\ where\ supp_id = 311$
17	$select\ count(*)\ from\ lineitems\ where\ supp_id = 401$
18	$select\ count(*)\ from\ lineitems\ where\ supp_id = 509$
19	$select\ count(*)\ from\ lineitems\ where\ supp_id = 799$
20	$select\ count(*)\ from\ lineitems\ where\ supp_id = 2100$

Appendix D

Detailed experimental results

Table D.1: Grocery sales results - part 1

Query	Prod Lvls	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	2	5993	6837	625	6744	1222	78
2	2	5993	1186	203	6744	1405	125
3	2	5993	348	47	6744	728	125
4	2	5993	115	15	6744	372	47
5	2	5993	6511	625	6744	3906	297
6	2	5993	1277	203	6744	1393	156
7	2	5993	583	94	6744	1286	203
8	2	5993	419	78	6744	603	93
9	2	5993	960	140	6744	547	79
10	2	5993	161	31	6744	158	16
11	2	5993	651	110	6744	444	78
12	2	5993	45	0	6744	108	16
13	2	5993	115	16	6744	181	31
14	2	5993	26	0	6744	75	15
15	2	5993	991	156	6744	572	78
16	2	5993	165	16	6744	164	31
17	2	5993	650	109	6744	326	62
18	2	5993	61	16	6744	111	15
19	2	5993	122	15	6744	166	16
20	2	5993	31	16	6744	97	16
1	3	9467	10516	844	10633	1516	94
2	3	9467	1586	250	10633	2765	187
3	3	9467	428	63	10633	1460	187
4	3	9467	194	47	10633	691	78
5	3	9467	10869	1000	10633	5022	328
6	3	9467	1713	266	10633	2711	187
7	3	9467	880	156	10633	2519	219
8	3	9467	642	125	10633	1168	140
9	3	9467	1259	188	10633	704	78
10	3	9467	221	31	10633	204	32
11	3	9467	909	141	10633	574	78
12	3	9467	60	15	10633	191	16
13	3	9467	142	31	10633	350	31
14	3	9467	29	0	10633	133	16
15	3	9467	1262	203	10633	737	78
16	3	9467	233	46	10633	214	32
17	3	9467	931	156	10633	420	47
18	3	9467	76	15	10633	126	16
19	3	9467	136	16	10633	298	31
20	3	9467	40	0	10633	168	32

Table D.2: Grocery sales results - part 2

Query	Prod Lvls	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	4	10407	14032	1016	11626	1516	94
2	4	10407	2150	265	11626	3104	235
3	4	10407	584	78	11626	1653	203
4	4	10407	277	47	11626	795	94
5	4	10407	14888	1031	11626	5064	297
6	4	10407	2322	282	11626	3054	281
7	4	10407	1172	156	11626	2791	219
8	4	10407	850	141	11626	1305	125
9	4	10407	1697	204	11626	707	93
10	4	10407	289	46	11626	205	16
11	4	10407	1256	172	11626	573	62
12	4	10407	82	16	11626	213	31
13	4	10407	200	32	11626	386	47
14	4	10407	41	0	11626	155	16
15	4	10407	1705	203	11626	739	94
16	4	10407	311	47	11626	215	31
17	4	10407	1300	157	11626	442	62
18	4	10407	108	16	11626	224	15
19	4	10407	196	16	11626	331	31
20	4	10407	55	0	11626	189	31
1	5	16395	17951	1140	18103	1516	94
2	5	16395	3016	297	18103	4299	265
3	5	16395	832	109	18103	2288	188
4	5	16395	433	47	18103	1008	141
5	5	16395	18681	1204	18103	5064	296
6	5	16395	3126	391	18103	4286	281
7	5	16395	1838	250	18103	3971	250
8	5	16395	1310	203	18103	1924	203
9	5	16395	2095	250	18103	708	62
10	5	16395	423	62	18103	205	32
11	5	16395	1436	171	18103	573	79
12	5	16395	112	43	18103	252	31
13	5	16395	269	31	18103	483	78
14	5	16395	57	0	18103	175	31
15	5	16395	2117	234	18103	739	63
16	5	16395	447	63	18103	215	15
17	5	16395	1474	172	18103	443	63
18	5	16395	125	15	18103	267	47
19	5	16395	251	32	18103	420	46
20	5	16395	74	16	18103	220	16

Table D.3: Grocery sales results - part 3

Query	Prod Lvls	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	6	23200	20024	1141	25371	1516	94
2	6	23200	3728	359	25371	5552	344
3	6	23200	1084	156	25371	2982	266
4	6	23200	392	62	25371	1253	141
5	6	23200	20708	1296	25371	5383	313
6	6	23200	3822	406	25371	5559	328
7	6	23200	2460	344	25371	5286	328
8	6	23200	1759	266	25371	2524	187
9	6	23200	2322	250	25371	740	78
10	6	23200	545	78	25371	238	31
11	6	23200	1558	187	25371	611	78
12	6	23200	107	0	25371	290	31
13	6	23200	306	31	25371	572	63
14	6	23200	63	16	25371	217	31
15	6	23200	2299	234	25371	776	47
16	6	23200	609	78	25371	249	16
17	6	23200	1594	187	25371	470	47
18	6	23200	137	16	25371	300	31
19	6	23200	283	47	25371	503	47
20	6	23200	97	16	25371	252	15
1	7	23964	20193	1188	26359	1516	94
2	7	23964	3825	422	26359	5877	359
3	7	23964	1130	156	26359	3141	313
4	7	23964	543	78	26359	1365	172
5	7	23964	20903	1453	26359	5383	344
6	7	23964	3946	406	26359	5871	391
7	7	23964	2559	328	26359	5572	391
8	7	23964	1813	297	26359	2691	219
9	7	23964	2356	266	26359	740	78
10	7	23964	576	78	26359	238	16
11	7	23964	1544	203	26359	611	62
12	7	23964	108	15	26359	302	47
13	7	23964	321	47	26359	589	78
14	7	23964	67	0	26359	230	32
15	7	23964	2331	265	26359	776	94
16	7	23964	643	78	26359	249	32
17	7	23964	1600	187	26359	471	47
18	7	23964	136	16	26359	306	31
19	7	23964	287	31	26359	517	63
20	7	23964	109	15	26359	266	32

Table D.4: Grocery sales results - part 4

Query	Prod Lvls	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	8	25089	21142	1406	27558	1516	94
2	8	25089	3958	406	27558	6215	422
3	8	25089	1153	390	27558	3327	375
4	8	25089	569	79	27558	1494	140
5	8	25089	21686	1484	27558	5383	359
6	8	25089	4085	468	27558	6206	422
7	8	25089	2666	344	27558	5873	406
8	8	25089	1887	296	27558	2872	219
9	8	25089	2373	266	27558	740	78
10	8	25089	603	79	27558	238	15
11	8	25089	1564	203	27558	611	78
12	8	25089	104	78	27558	327	125
13	8	25089	328	31	27558	647	94
14	8	25089	77	0	27558	256	32
15	8	25089	2381	281	27558	776	94
16	8	25089	663	78	27558	249	31
17	8	25089	1655	219	27558	471	63
18	8	25089	137	15	27558	326	47
19	8	25089	311	47	27558	558	62
20	8	25089	105	15	27558	292	47
1	9	28849	22817	1328	31727	1516	93
2	9	28849	4660	500	31727	6988	454
3	9	28849	1306	187	31727	3747	297
4	9	28849	683	94	31727	1674	203
5	9	28849	23213	1390	31727	5383	328
6	9	28849	4742	484	31727	7049	454
7	9	28849	3033	297	31727	6650	406
8	9	28849	2219	547	31727	3264	625
9	9	28849	2435	265	31727	740	78
10	9	28849	671	78	31727	238	31
11	9	28849	1644	203	31727	611	63
12	9	28849	108	16	31727	363	47
13	9	28849	387	47	31727	720	94
14	9	28849	78	0	31727	290	31
15	9	28849	2426	282	31727	776	93
16	9	28849	743	79	31727	249	31
17	9	28849	1745	203	31727	471	47
18	9	28849	159	16	31727	369	47
19	9	28849	395	47	31727	627	63
20	9	28849	113	16	31727	326	31

Table D.5: Grocery sales results - part 5

Query	Prod Lvls	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	10	37510	23946	1453	41171	1516	94
2	10	37510	5768	610	41171	8410	562
3	10	37510	1637	235	41171	4516	406
4	10	37510	641	78	41171	1881	219
5	10	37510	24595	1641	41171	5383	329
6	10	37510	5881	609	41171	8389	578
7	10	37510	3913	375	41171	7892	485
8	10	37510	2756	453	41171	3977	297
9	10	37510	2468	281	41171	740	79
10	10	37510	795	94	41171	238	14
11	10	37510	1815	219	41171	611	79
12	10	37510	151	16	41171	377	62
13	10	37510	451	63	41171	803	109
14	10	37510	90	15	41171	321	32
15	10	37510	2513	422	41171	776	94
16	10	37510	856	110	41171	249	31
17	10	37510	1844	219	41171	471	46
18	10	37510	187	32	41171	452	62
19	10	37510	468	47	41171	704	93
20	10	37510	144	16	41171	342	47

Table D.6: TPC-H Lineitems results - part 1

Query	Supplier ID's	MHC Full Scan I/O	MHC I/O	MHC CPU msecs	MDDC Full Scan I/O	MDDC I/O	MDDC CPU msecs
1	4000	7173	334	62	7177	495	94
2	4000	7173	333	31	7177	546	94
3	4000	7173	333	47	7177	544	62
4	4000	7173	333	31	7177	532	78
5	4000	7173	333	46	7177	573	63
6	4000	7173	333	31	7177	546	94
7	4000	7173	333	31	7177	564	78
8	4000	7173	333	47	7177	521	62
9	4000	7173	333	32	7177	605	78
10	4000	7173	333	47	7177	517	63
11	4000	7173	4176	516	7177	25	0
12	4000	7173	4178	359	7177	641	47
13	4000	7173	4176	360	7177	699	47
14	4000	7173	4186	469	7177	673	47
15	4000	7173	4187	375	7177	732	47
16	4000	7173	4181	422	7177	699	63
17	4000	7173	4184	453	7177	662	32
18	4000	7173	4172	375	7177	614	31
19	4000	7173	4139	391	7177	744	47
20	4000	7173	4364	484	7177	689	47
1	6000	10690	577	79	9842	620	125
2	6000	10690	576	47	9842	632	140
3	6000	10690	576	47	9842	650	109
4	6000	10690	576	62	9842	575	125
5	6000	10690	576	62	9842	641	125
6	6000	10690	576	62	9842	648	125
7	6000	10690	576	47	9842	631	109
8	6000	10690	576	63	9842	614	94
9	6000	10690	576	62	9842	664	94
10	6000	10690	577	63	9842	565	93
11	6000	10690	5395	547	9842	34	0
12	6000	10690	5399	344	9842	781	78
13	6000	10690	5397	359	9842	877	73
14	6000	10690	5403	406	9842	776	63
15	6000	10690	5410	375	9842	846	63
16	6000	10690	5409	360	9842	809	64
17	6000	10690	5412	359	9842	776	63
18	6000	10690	5400	406	9842	796	78
19	6000	10690	5355	422	9842	844	63
20	6000	10690	5665	469	9842	758	78

Table D.7: TPC-H Lineitems results - part 2

Query	Supp ID's	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	8000	14240	729	94	14368	990	140
2	8000	14240	728	78	14368	993	140
3	8000	14240	728	78	14368	1102	125
4	8000	14240	728	63	14368	1085	156
5	8000	14240	728	63	14368	1110	156
6	8000	14240	728	63	14368	1093	156
7	8000	14240	728	78	14368	1097	109
8	8000	14240	728	63	14368	1008	109
9	8000	14240	728	78	14368	1147	109
10	8000	14240	729	62	14368	1013	125
11	8000	14240	5387	516	14368	37	0
12	8000	14240	5390	407	14368	1157	62
13	8000	14240	5388	421	14368	1211	79
14	8000	14240	5394	390	14368	1457	79
15	8000	14240	5402	454	14368	1492	78
16	8000	14240	5402	453	14368	4308	93
17	8000	14240	1390	79	14368	1390	79
18	8000	14240	5392	375	14368	1428	94
19	8000	14240	5351	468	14368	1489	79
20	8000	14240	5664	469	14368	1406	94
1	10000	17775	918	94	16498	1176	109
2	10000	17775	918	78	16498	1190	125
3	10000	17775	918	94	16498	1277	125
4	10000	17775	918	94	16498	1245	125
5	10000	17775	918	78	16498	1239	125
6	10000	17775	918	78	16498	1287	125
7	10000	17775	918	94	16498	1246	125
8	10000	17775	918	78	16498	1212	125
9	10000	17775	918	93	16498	1266	125
10	10000	17775	916	93	16498	1335	125
11	10000	17775	5424	516	16498	45	15
12	10000	17775	5427	421	16498	1461	79
13	10000	17775	5425	516	16498	1449	62
14	10000	17775	5431	391	16498	1646	93
15	10000	17775	5439	516	16498	1769	94
16	10000	17775	5437	468	16498	1723	94
17	10000	17775	5440	484	16498	1648	78
18	10000	17775	5428	453	16498	1657	78
19	10000	17775	5372	438	16498	1711	93
20	10000	17775	5675	453	16498	1607	94

Table D.8: TPC-H Lineitems results - part 3

Query	Supp ID's	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	12000	21223	1094	110	19556	1106	109
2	12000	21223	1094	109	19556	1106	125
3	12000	21223	1094	93	19556	1250	125
4	12000	21223	1094	93	19556	1172	141
5	12000	21223	1094	93	19556	1172	125
6	12000	21223	1094	109	19556	1233	141
7	12000	21223	1094	110	19556	1309	140
8	12000	21223	1094	110	19556	1167	140
9	12000	21223	1093	94	19556	1263	109
10	12000	21223	1094	94	19556	1118	125
11	12000	21223	5424	437	19556	41	0
12	12000	21223	5427	391	19556	1340	62
13	12000	21223	5425	391	19556	1366	62
14	12000	21223	5431	390	19556	1618	141
15	12000	21223	5434	406	19556	1701	110
16	12000	21223	5432	375	19556	1605	93
17	12000	21223	5434	375	19556	1589	109
18	12000	21223	5422	406	19556	1558	94
19	12000	21223	5366	375	19556	1626	109
20	12000	21223	5677	422	19556	1562	94
1	14000	24667	1290	125	22582	1126	125
2	14000	24667	1290	125	22582	1198	125
3	14000	24667	1290	125	22582	1362	109
4	14000	24667	1290	110	22582	1368	125
5	14000	24667	1290	125	22582	1355	140
6	14000	24667	1290	125	22582	1332	140
7	14000	24667	1290	125	22582	1356	156
8	14000	24667	1290	125	22582	1226	140
9	14000	24667	1289	125	22582	1394	140
10	14000	24667	1290	109	22582	1168	125
11	14000	24667	5419	453	22582	35	0
12	14000	24667	5422	469	22582	1282	63
13	14000	24667	5420	438	22582	1330	62
14	14000	24667	5426	437	22582	1840	94
15	14000	24667	5433	406	22582	1855	110
16	14000	24667	5431	390	22582	1717	94
17	14000	24667	5433	375	22582	1748	94
18	14000	24667	5421	406	22582	1665	78
19	14000	24667	5367	453	22582	1801	94
20	14000	24667	5679	453	22582	1735	94

Table D.9: TPC-H Lineitems results - part 4

Query	Supp ID's	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	16000	27981	1418	125	28493	1268	78
2	16000	27981	1418	141	28493	1442	125
3	16000	27981	1418	125	28493	1653	141
4	16000	27981	1418	125	28493	1616	156
5	16000	27981	1418	125	28493	1626	109
6	16000	27981	1418	125	28493	1636	109
7	16000	27981	1418	125	28493	1674	125
8	16000	27981	1418	125	28493	1480	125
9	16000	27981	1418	125	28493	1670	141
10	16000	27981	1418	125	28493	1442	156
11	16000	27981	5421	437	28493	32	16
12	16000	27981	5424	407	28493	1389	62
13	16000	27981	5422	438	28493	1463	62
14	16000	27981	5428	421	28493	2165	94
15	16000	27981	5435	391	28493	1734	92
16	16000	27981	5433	375	28493	2310	94
17	16000	27981	5436	422	28493	2176	94
18	16000	27981	5424	438	28493	2085	94
19	16000	27981	5367	391	28493	2404	94
20	16000	27981	5678	375	28493	2080	94
1	18000	31469	1643	156	32037	1464	110
2	18000	31469	1643	156	32037	1271	140
3	18000	31469	1643	157	32037	1826	125
4	18000	31469	1643	156	32037	1756	172
5	18000	31469	1643	140	32037	1764	172
6	18000	31469	1643	141	32037	1839	171
7	18000	31469	1644	157	32037	1848	172
8	18000	31469	1644	156	32037	1680	157
9	18000	31469	1644	156	32037	1833	172
10	18000	31469	1644	156	32037	1632	156
11	18000	31469	5440	485	32037	40	0
12	18000	31469	5443	406	32037	1588	63
13	18000	31469	5441	422	32037	1597	62
14	18000	31469	5447	391	32037	2421	109
15	18000	31469	5455	344	32037	2474	172
16	18000	31469	5453	453	32037	2551	172
17	18000	31469	5455	422	32037	2422	156
18	18000	31469	5443	406	32037	2338	109
19	18000	31469	5386	407	32037	2493	109
20	18000	31469	5680	422	32037	2344	110

Table D.10: TPC-H Lineitems results - part 5

Query	Supp ID's	MHC Full Scan I/O's	MHC I/O's	MHC CPU msecs	MDDC Full Scan I/O's	MDDC I/O's	MDDC CPU msecs
1	20000	35704	1750	172	36308	1538	125
2	20000	35704	1750	156	36308	1691	157
3	20000	35704	1750	156	36308	1881	110
4	20000	35704	1750	156	36308	1823	187
5	20000	35704	1750	172	36308	1833	172
6	20000	35704	1750	156	36308	1867	188
7	20000	35704	1751	156	36308	1887	141
8	20000	35704	1751	172	36308	1738	172
9	20000	35704	1751	157	36308	1860	156
10	20000	35704	1742	172	36308	1961	156
11	20000	35704	5444	453	36308	49	0
12	20000	35704	5447	422	36308	1669	63
13	20000	35704	5445	469	36308	1675	62
14	20000	35704	5448	454	36308	1701	68
15	20000	35704	5459	422	36308	2586	125
16	20000	35704	5457	422	36308	2445	109
17	20000	35704	5459	453	36308	2436	109
18	20000	35704	5447	422	36308	2398	109
19	20000	35704	5387	484	36308	2516	110
20	20000	35704	5681	516	36308	2392	109

Appendix E

Examples

Table E.1: Product dimension

COLUMN	DESCRIPTION	PRIMARY KEY
PROD_ID	Product Identifier	Y
PROD_DESC	Product Description	N
PROD_CAT_ID	Product Category Identifier	N
PROD_CAT_NM	Product Category Name	N

Table E.2: Month dimension

COLUMN	DESCRIPTION	PRIMARY KEY
MO_ID	Month Identifier	Y
MO_NM	Month Name	N
QTR_ID	Quarter Identifier	N
QTR_NM	Quarter Name	N
YR_ID	Year Identifier	N
YR_NM	Year Name	N

Table E.3: Store dimension

COLUMN	DESCRIPTION	PRIMARY KEY
STR_ID	Store Identifier	Y
STR_NM	Store Name	N
ZIPCODE_CD	Zip Code	N
ST_CD	State Code	N
ST_NM	State Name	N
RG_ID	Region Identifier	N
RG_NM	Region Name	N

Table E.4: Sales fact

COLUMN	DESCRIPTION	PRIMARY KEY
PROD_ID	Product Identifier	Y
MO_ID	Month Identifier	Y
STR_ID	Store Identifier	Y
SLS_DLR	Sales Dollars	N