Encoded Bitmap Indexes and Their Use for Data Warehouse Optimization

Fachbereich Informatik
der Technischen Universität Darmstadt

Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurswissenschaften (Dr.-Ing.)

von
Ming-Chuan Wu

Referenten:
Prof. Alejandro P. Buchmann, Ph.D.
Prof. P. Larson, Ph.D. (Univ. of Waterloo)


Darmstädter Dissertation
D17
Gedruckt mit Unterstützung des Deutschen Akademischen Austauschdienstes
Ming-Chuan Wu

Encoded Bitmap Indexes and Their Use for Data Warehouse Optimization

D17 (Diss. TU Darmstadt)
Foreword

The requirements of query processing in a Data Warehouse are quite different from those of an OLTP system. Major differences exist in the schema, which in the case of a relational DW is usually some variant of a star schema with rather large fact tables and descriptive dimension tables linked to the fact table(s) through key/foreign key relationships. Query processing typically involves selections on the dimension tables with star joins to the fact tables. DW queries further involve aggregations and presentation in sorted order. These are batch-like operations that in conventional query processing systems require the processing of the whole data set before further processing in sorted order or presentation to the user is possible. However, in OLAP applications, the power user is the scarce resource and response time rather than total execution time should be optimized.

This work addresses these problems by first proposing a novel indexing method, encoded bitmap indexing, that complements simple bitmap indexing and bit-slice indexes and is particularly useful for large domains. Bitmap indexing is then used to solve the optimization of aggregation operations, the optimization or substitution of star-joins, and the optimization for response time rather than for execution time. The proposed optimization strategies produce incrementally exact results of aggregations in sorted order. These results can be used both for presentation to the user or for pipelining to other operations and constitute an improvement over strategies based on sampling and statistical refinement.

Simple bitmap indexes were introduced some time ago. They are powerful index structures that allow the processing of queries on the index without the need for accessing the underlying data. Simple bitmap indexes, however, suffer from sparcity when the domain of the indexed attribute becomes larger. Encoded bitmaps solve this shortcoming by encoding the domain values in such a way, that the resulting bitmap index is dense. Instead of requiring one bit vector per domain value only log(n) vectors are required, where n is the attribute’s cardinality. Furthermore, through clever encoding strategies that are presented here, frequently asked queries can be optimized by reducing the number of bit vectors that must be accessed to evaluate a certain predicate.

With the necessary extensions to bitmap indexing in place, the optimization strategies for queries involving selections, aggregations, Group By, and Order By are introduced. The proposed optimizations exploit query rewriting techniques for selections and exploit knowledge about functional dependencies to push Group By operations past joins. One of the principles developed here consists in transforming selections and/or groupings on dimension attributes into operations on dimension keys. Since these keys are part of the key of the fact table the selections and/or groupings on dimensions are transformed to selections on (parts of) keys of the fact table, thereby avoiding the costly join operations.
The feasibility of the new optimization strategies is shown through a pilot implementation of a bit-map based rudimentary DBMS that is benchmarked against three commercial DBMSs. The comparison is based on the TPC-H benchmark. Although benchmarks are highly sensitive to implementation and tuning and should be regarded with caution, the results do indicate the great potential that is inherent in the proposed indexing mechanism and the optimization strategies. A section on lessons learned nicely rounds off this work.

The ideas proposed by Ming Chuan Wu represent an important step in the improvement of query processing in relational Data Warehouses. Hopefully they will find their way into a commercial product where they should be subjected to the test of the marketplace.

Alejandro P. Buchmann, Ph.D.
Darmstadt, February 2001
The proliferation of Data Warehouses and On-Line Analytical Processing (OLAP) has renewed discussions of many issues in database technology, including query processing and query optimization. Due to the changes in query environments in Data Warehouses, some conventional query processing techniques and index structures used to speed up query execution are no longer well-suited as they were in On-Line Transaction Processing (OLTP) systems. To tackle the new challenges one promising approach is to introduce new index structures that are well-suited in the environment where queries are mostly read-only and access a large fraction of underlying data.

Bitmap indexing introduced by O’Neil in the 1960s, due to the simplicity of its data structure and efficiency of index intersection, has attracted attention in both the research community and the industry. In spite of its strength, simple bitmap indexes have restrictions on their use. Namely, the space requirement of a simple bitmap index is a linear function of the cardinality of indexed table and the cardinality of the indexed attribute domain. As the cardinality of the attribute domain grows, the escalation of space requirements not only degrades the space efficiency, but also the processing efficiency of bitmap indexing. Two orthogonal approaches are used to eliminate the restrictions by improving the space efficiency without much sacrifice of processing efficiency. The first approach is aimed at a better index design. The second approach is to compress the bitmaps. In this work, the first approach is discussed. Nevertheless, modern compression techniques that enable bitmap processing without the need of decompression will surely further improve the overall query performance. Although compression is beyond the scope of this work, it is a technique that complement the first approach.

In this work, a new bitmap indexing technique is introduced — encoded bitmap indexing. As its name indicates, an encoded bitmap index consists of an encoding function on the attribute domain, a set of bitmaps which preserve the simple data structure of its predecessor and a set of retrieval Boolean functions. The space requirements of encoded bitmap indexing are reduced to a logarithmic function of the cardinality of the attribute domain, and encoded bitmap indexes provide more opportunities to optimize range selection evaluation than simple bitmap indexes. In order to provide a global usage of bitmap indexes in query processing, algorithms that exploit bitmap indexes to evaluate selections, Group Bys, aggregate functions and Order Bys are introduced here.

The introduction of bitmap indexes into query processing changes the landscape of query processing/optimization in Data Warehouses quite a lot. It enables that large portion of query operations are directly evaluated on bitmaps, and that data of interest can be fetched in the desired order to support pipelining or to shorten the query response time. In addition, the group-set bitmap indexes — a variant of bitmapped join-index — enrich the conventional view
and usage of bitmap indexes. A group-set bitmap index is a combination of bitmapped join indexes, selection bitmaps and grouping bitmaps. It can be used to evaluate selections, Group Bys and aggregate functions, and to omit Star-Joins in typical OLAP queries. A bitmap-enabled experimental SQL engine is implemented. The performance measurements show the advantages of bitmap indexes over conventional value-list index structures in Data Warehousing and justify the use of bitmap indexes in query processing in OLAP environments. It is convincing that bitmap indexing does provide a key solution to query processing and optimization in Data Warehousing and OLAP systems.

Acknowledgments

This work wouldn’t exist if there were not so many known and unknown supports standing behind me. I couldn’t have ever made a complete list of thanks, therefore, I would like to give my first thank to all who has ever stood behind me and given me encouragement and faith in the work.

I’d like to thank my adviser, Prof. Alejandro P. Buchmann, Ph.D., for giving me the chance to work in his group. Many helpful discussions and encouragement, especially in the beginning, when the problems and the feasibility of this work were not yet clear, helped me uncovering the problems and developing the solutions. Thanks also to Prof. P. Larson Ph.D., Prof. Dr. J. Buchmann, Prof. Dr. S.A. Huss, and Prof. Dr. H.-J. Hoffmann for their kindly review of my thesis and their constructive comments on this work.

I’d like to thank Deutscher Akademischer Austauschdienst (DAAD) for the scholarship during my stay in Germany which let me be able to concentrate on the research work without worrying about financial matters. Their generosity and helpful assistances saved me, as a newcomer to a foreign land, much time and effort of dealing with bureaucratic affairs.

In addition, I’d like to thank Christof Bornhövd, Mariano Cilia, Christian Haul, Christoph Liebig, Zhu Yan, Astrid Endres, Heidemarie Knapp and Marion Braun for their patience to my sometimes insistent questions and for giving me a harmonious working atmosphere. Special thanks also to Ronald Bourret for his reviewing of the manuscripts and to Johannes Schütt and Jörg Pfister for their valuable comments regarding to the presentation of this work.

Finally, I want to thank my parents for their mental support and understanding, and I’d like to give my best thank to Kim, who showed her understanding and patience that nobody else may ever have, especially at the time of each paper due, when I neglected her.

Ming-Chuan Wu
Darmstadt, January 2001
## Contents

1 Introduction .................................................. 13
   1.1 A Brief History of Information Processing .......... 13
   1.2 The Definition of Data Warehouses ................. 14
   1.3 Impact on Database Technology and Motivation for Novel Indexing ...... 15

2 Indexing Data ................................................ 17
   2.1 To Index or Not to Index .................................. 17
      2.1.1 Paradox between index scans and table scans ...... 18
      2.1.2 Out of the bounding box: Selections ............... 20
   2.2 Conventional Indexes ...................................... 26
      2.2.1 B-trees and variants ............................... 27
      2.2.2 Restrictions of B-trees ............................ 28
   2.3 Renaissance of Bitmap Indexes ......................... 32
      2.3.1 The emergence of bitmap indexes ................. 32
      2.3.2 Encoded bitmap indexes: a first glance .......... 35
      2.3.3 Bit slices ........................................ 38
      2.3.4 Design space of bitmap indexes ................. 40
      2.3.5 Variants of bitmap indexes ....................... 41

3 Encoded Bitmap Indexes for Data Warehouses ............. 47
   3.1 The Definition of an Encoded Bitmap Index .......... 47
      3.1.1 Building and maintaining encoded bitmap indexes ........................................................................ 48
      3.1.2 The encoding ........................................ 53
   3.2 Examples of Applications ................................ 58
      3.2.1 Hierarchy encoding .................................. 58
      3.2.2 Total-order preserving encoding .................. 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 Range-based encoding</td>
<td>61</td>
</tr>
<tr>
<td>3.3 Performance Analysis: Analytical Approach</td>
<td>62</td>
</tr>
<tr>
<td>3.3.1 Performance of encoded bitmap indexing</td>
<td>62</td>
</tr>
<tr>
<td>3.3.2 Comparing encoded bitmap indexes with B-trees</td>
<td>64</td>
</tr>
<tr>
<td>3.3.3 Comparing encoded bitmap indexing with simple bitmap indexing</td>
<td>70</td>
</tr>
<tr>
<td>3.4 Bit Slices for Continuous Data Types</td>
<td>73</td>
</tr>
<tr>
<td>3.5 Bitmap Index Design: Time Optimal Index</td>
<td>76</td>
</tr>
<tr>
<td>3.5.1 Continuous range selections</td>
<td>76</td>
</tr>
<tr>
<td>3.5.2 Discrete range selections</td>
<td>77</td>
</tr>
<tr>
<td>3.5.3 Global time optimum</td>
<td>79</td>
</tr>
<tr>
<td>3.5.4 Global time optimal indexes under space constraint</td>
<td>80</td>
</tr>
<tr>
<td>3.6 Related Work</td>
<td>81</td>
</tr>
<tr>
<td>4 Query Processing and Optimization using Bitmaps</td>
<td>85</td>
</tr>
<tr>
<td>4.1 Performance Goal and Methodologies</td>
<td>85</td>
</tr>
<tr>
<td>4.1.1 Overview of query processing using bitmap indexes</td>
<td>86</td>
</tr>
<tr>
<td>4.1.2 Query rewrites</td>
<td>88</td>
</tr>
<tr>
<td>4.1.3 Pipelining</td>
<td>94</td>
</tr>
<tr>
<td>4.2 Selection Evaluation</td>
<td>95</td>
</tr>
<tr>
<td>4.2.1 Static optimization</td>
<td>95</td>
</tr>
<tr>
<td>4.2.2 Dynamic optimization</td>
<td>96</td>
</tr>
<tr>
<td>4.3 Optimizing Group Bys and Aggregation Using Bitmaps</td>
<td>103</td>
</tr>
<tr>
<td>4.3.1 Optimizing Group Bys</td>
<td>104</td>
</tr>
<tr>
<td>4.3.2 Optimizing Order Bys</td>
<td>111</td>
</tr>
<tr>
<td>4.3.3 Aggregation on dimension tables</td>
<td>116</td>
</tr>
<tr>
<td>4.3.4 Related work</td>
<td>121</td>
</tr>
<tr>
<td>4.3.5 Performance of bitmap-based grouping algorithms</td>
<td>122</td>
</tr>
<tr>
<td>5 Performance Measurements and Analysis</td>
<td>127</td>
</tr>
<tr>
<td>5.1 Preliminary Results</td>
<td>128</td>
</tr>
<tr>
<td>5.1.1 Evaluating selections</td>
<td>128</td>
</tr>
<tr>
<td>5.1.2 Evaluating Group Bys and aggregates</td>
<td>134</td>
</tr>
<tr>
<td>5.1.3 Evaluating queries involving Group Bys, aggregations and selections</td>
<td>140</td>
</tr>
<tr>
<td>5.2 Implementation and Experimental Setups</td>
<td>147</td>
</tr>
</tbody>
</table>
F Symbols, Notations Used In This Work 213

F.1 The Relational Algebra . . . . . . . . . . . . . . . . . . . . . . . . . . . 213
F.2 Operators and Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . 213
List of Figures

1.1 The logical architecture of a Data Warehouse system ......................... 14
2.1 Hit ratio subject to different selectivities and blocking factor ............. 19
2.2 Two execution plans for $\sigma T \bowtie S$ ....................................... 21
2.3 Data structures of $B^+$-trees .................................................. 27
2.4 A classical example of simple bitmap indexing ................................. 33
2.5 Schema of the experimental database .......................................... 36
2.6 Encoded bitmap indexing ......................................................... 37
2.7 Truth table and retrieval Boolean function ..................................... 37
2.8 Bit-sliced indexes ................................................................ 38
2.9 Bit-sliced indexes with decimal base ........................................... 39
2.10 Design space of bitmap indexes .................................................. 41
2.11 Group-set bitmap index ............................................................ 42
2.12 Group-set bitmap index ............................................................ 43
2.13 Bitmapped join index ............................................................... 44
3.1 Update with domain expansion ..................................................... 49
3.2 Updates with domain expansion .................................................. 50
3.3 Relationship between optimal and well-defined encodings ................. 54
3.4 Karnaugh map .................................................................. 55
3.5 Karnaugh maps ................................................................ 57
3.6 Hierarchies along dimensions .................................................... 59
3.7 Dimension hierarchy on nations and its encoding .............................. 59
3.8 Total-order preserving encoding ................................................. 61
3.9 Example of using range-based encoding ....................................... 61
3.10 Using encoded bitmap indexes for range-based indexing ................. 62
3.11 Boundary analysis with respect to table cardinality, split factor and attribute’s cardinality ........................................... 67
3.12 Boundary analysis with respect to indexed table- and attribute-cardinalities .... 68
3.13 Processing costs of B-trees and encoded bitmaps, page size= 8K, tuple length= 128 bytes (i.e., blocking factor= 32), $\epsilon_a(0) = 0.024$ sec, $\epsilon_a(1) = 0.0015$ sec ..... 70
3.14 Space requirements of encoded and simple bitmap indexes .................................................. 72
3.15 Processing costs of simple and encoded bitmap indexes, $m = 50$ and $|\log_2 m| = 6$ .... 72
3.16 Time optimal indexes, $|A| = 1024$ ................................................................. 79
3.17 Global time optimal indexes ................................................................. 80
3.18 Global time optimal indexes under space constraints .................................................. 81
4.1 System components of a query processor ................................................................. 86
4.2 Query execution flows ................................................................. 87
4.3 Group-set bitmap index ................................................................. 89
4.4 Transformation of the execution tree for $A \leq 864$, $\cup$ denotes OR and $\cap$ denotes AND ................................................................. 95
4.5 Transformation of execution tree for $A \leq 864$ ................................................................. 96
4.6 An example of encoded bitmap indexing on $A$ ................................................................. 98
4.7 Karnaugh graph of retrieval Boolean functions ................................................................. 98
4.8 Transforming tuple-ids into data-block-ids ................................................................. 99
4.9 Execution trees for $A \leq 894$ and $A \leq 869$ ................................................................. 103
4.10 Data flows of multi-pass sort-merge algorithms ................................................................. 104
4.11 Execution trees of SQL 7 ................................................................. 105
4.12 Group-set bitmaps on lineitems using customers.mktsegment, where $b_A$ is the group-set bitmap for the group AUTOMOBILE, $b_B$ for BUILDING, $b_F$ for FURNITURE, $b_H$ for HOUSEHOLD .... 107
4.13 Group By evaluation coping with the page-I/O nature ................................................................. 108
4.14 Execution of Algorithm 6 ................................................................. 110
4.15 Moving windows of size $w$ ................................................................. 110
4.16 Evaluation of ordered Group By ................................................................. 114
4.17 Scenarios concerning response time and query processing time of both alternatives .... 115
4.18 Three sub-cases, where $r < s < t$ ................................................................. 115
4.19 Aggregate on dimension table(s) ................................................................. 118
4.20 Snapshots of dimension tables customers and orders ................................................................. 120
4.21 Response time versus execution time ................................................................. 123
4.22 Effects of batch-mode data loading ................................................................. 125
4.23 Effects of batch-mode data loading, attribute with uniform distribution

5.1 Response time of \texttt{bitSQL} and CDB1 for evaluating six different queries

5.2 Response time of \texttt{bitSQL} and CDB1 for evaluating Group Bys and aggregates

5.3 Resource utilization analysis for \texttt{bitSQL}

5.4 Resource utilization analysis of GQ 3 for CDB1

5.5 Resource utilization of GQ 4 and 5 by \texttt{bitSQL}

5.6 Performance comparison between CDB1 and \texttt{bitSQL}, 250 MB test data

5.7 Performance comparison between CDB1 with \textit{optimal} indexes and \texttt{bitSQL}, 250 MB test data

5.8 Performance comparison between CDB3 and \texttt{bitSQL} (continued), 250 MB test data

5.9 Performance comparison between CDB3 and \texttt{bitSQL}, 250 MB test data

5.10 Performance comparison between CDB2 and \texttt{bitSQL}, 250 MB test data

5.11 Memory utilization of different DBMSs

5.12 Resource utilization of CDB1 and \texttt{bitSQL}

5.13 System architecture of \texttt{bitSQL}

5.14 System loads during the runs of \texttt{bitSQL}^+

5.15 Performance comparison between two versions of \texttt{bitSQL}, 250 MB test data

5.16 Performance comparison between two versions of \texttt{bitSQL}, 1 GB test data

5.17 Performance comparison between CDB1 and \texttt{bitSQL}, 1 GB test data (continued)

5.18 Performance comparison between CDB1 and \texttt{bitSQL}, 1 GB test data

5.19 Performance comparison between CDB1 and different versions of \texttt{bitSQL}, 1 GB test data

5.20 Performance comparison between CDB1 and different versions of \texttt{bitSQL}, 1 GB test data (continued)

5.21 Performance comparison between CDB1 and different versions of \texttt{bitSQL}, 1 GB test data

5.22 Performance comparison between CDB2, CDB3, \texttt{bitSQL} and \texttt{bitSQL}^{++}, 1 GB test data (continued)

5.23 Performance comparison between CDB2, CDB3, \texttt{bitSQL} and \texttt{bitSQL}^{++}, 1 GB test data

5.24 Performance comparison between CDB2 and \texttt{bitSQL}^{++}, 2 GB test data

5.25 Performance comparison between CDB2 (using up to 15 MB memory) and \texttt{bitSQL}^{++}, 2 GB test data

5.26 Performance comparison between CDB1 and \texttt{bitSQL}^{++}, 4 GB test data

5.27 Performance comparison between CDB1 and \texttt{bitSQL}^\Delta, 4 GB test data
5.28 Performance comparison between simple bitmaps (SBI) and encoded bitmaps (EBI) ................................................. 173

6.1 An encoding that releases the minimized space requirement and takes probabilities of query occurrence into account (\(\times\) denotes don't-care conditions) ............ 180

A.1 Space Requirement analysis releasing \(\mu^* = \mu\) .................................................. 184
A.2 Best case analysis ................................................................. 185

C.1 Poor partitioning due to improper threshold .......................................... 202
List of Tables

2.1 Rules of thumb for I/O costs ................................................. 24
2.2 Quality of B-trees according to McDips properties ..................... 32
3.1 Minimized retrieval functions and the numbers of bitmaps read for $P_{nation.jd}$ ..................... 60
3.2 Notations used for analysis of encoded bitmap indexing .................. 63
3.3 Notations used for analysis of $B^{*+}$-trees ................................ 65
3.4 Notations used for analysis of $B^{*+}$-trees (continued) .................. 68
3.5 Notations used for analysis of bitmap indexing ............................ 71
4.1 Notations used in the cost analysis of the inclusion/exclusion approach ............. 99
4.2 Optimized number of bitmap scans with respect to cardinality of range selection ............ 101
4.3 Notations used in performance analysis of bitmap-based grouping algorithms .......... 122
5.1 System configurations of testbeds ........................................... 127
5.2 Statistics of the testing queries ............................................. 130
5.3 Statistics of test databases and indexed defined in both bitSQL and CDB1 ........ 130
5.4 Selectivities of individual selections ....................................... 132
5.5 Statistics of the testing queries ............................................. 136
5.6 Durations of grouping bitmap generation using simple and encoded bitmaps ............ 140
5.7 Optimal indexes for decision support workloads .......................... 141
5.8 Schema of the experimental database ..................................... 158
5.9 Selectivities of all test queries ............................................. 168
5.10 Time elapsed for index creation on 4 GB data ............................ 174
A.1 Comparison of Equations (3.9) and (A.2), Error = (Approximation − Best Case), Error in % = $\frac{\text{Error}}{\text{Best Case}}$ ................. 186
B.1 A non-binary encoded bitmap index ....................................... 199
D.1 Bitmap operations ......................................................... 203

E.1 Selected tables (✓) and Group By (✗) attributes of each query ................ 206

F.1 Relational operators ..................................................... 213
F.2 Operators and general symbols ........................................ 213
## List of Algorithms

1. Bitmap Index Maintenance .................................................. 51
2. Discrete Selection, range ($\leq$) bit-encoding ........................ 77
3. Discrete Selection, equality (=) bit-encoding .......................... 78
4. Range evaluation with tree-reduction .................................. 97
5. Finding common variables .................................................. 102
6. **bitmapGrouping** — Group By evaluation using bitmaps .......... 109
7. **bitmapGrouping** — Group By evaluation using bitmaps with limited buffers .. 111
8. **bitmapOrderedGrouping** — Bitmap grouping involving Order By aggregation .. 112
9. evalRestAgg ................................................................. 116
Chapter 1

Introduction

1.1 A Brief History of Information Processing

New information processing technologies emerge whenever data volume grows drastically and the complexity of data management increases beyond manual control. In the early 1960s, electronic data processing began to change the way data and information was managed. Computers eased the task of managing data and enriched the applications. At that time, data was created and maintained in files. Until the end of the 1960s, the growth of files exploded. Files were everywhere. Problems — arising from redundancy, inconsistency of data and complicated referential dependencies between data and programs — became so severe that without a change in the way data was managed, due to the inconsistency alone, processed data would be no longer useful.

From the late 1960s and throughout the 70s, the concept of databases and the technologies of database management brought data processing and data management into another era. Technologies, such as transaction processing, concurrency control, recovery, query processing and optimization, helped to build high performance on-line transaction processing (OLTP) systems. New data models, new algorithms and methods kept showing up and made the world of data management richer than ever.

As the volume of data kept growing, new strategic applications gained importance, like decision support systems (DSS), executive information systems (EIS), etc. The groups of users are also expanding, from terminal operators to decision makers and the use of data is evolving from transaction processing to on-line analytical processing (OLAP). Programs, like report generators, extract data from databases, perform the predefined calculations and present the results in a more understandable form. At the beginning, a single database serving both transaction and analytical processing worked well and met the users’ needs. However, data kept growing and soon the large volumes of data invalidated the solution of using a single database for both types of processing. Unlike transaction processing, analytical processing accesses large fractions of underlying data and performs complicated calculations on them. Such CPU and I/O intensive loads degrade the performance of the transaction processing applications running on the same system dramatically. Besides, to fulfill the requirements of analytical applications, the history of data changes must be kept in the database. This complicates the data management of critical transaction processing applications. It is time for a change again. However, this time it is an evolutionary change. Data Warehousing is introduced to solve the above problems.
1.2 The Definition of Data Warehouses

A Data Warehouse is a subject-oriented, integrated, non-volatile, and time variant collection of data in support of management’s decision [52]. A Data Warehouse system consists of a back-end database server, an OLAP engine and a front-end tool set. OLAP and Data Warehousing are usually used interchangeably. However, Data Warehousing includes tasks such as data extraction, data cleansing, data integration, metadata management, indexing, query processing and optimization, and other services that a DBMS provides, while OLAP denotes the services, which support decision making, built on top of those services provided by a Data Warehouse. Generally speaking, OLAP services provide a multidimensional presentation of the warehouse data, manage data cubes (summarized data of multiple dimensions) and use data cubes to answer users’ queries.

A logical architecture of a data warehouse system is depicted in Figure 1.1.

The data store component corresponds to a database server. It can be a relational, an object-oriented or a multidimensional database management system. The data management component provides services for data model conformation between the data store component and the OLAP engine, or between the data store component and the data sources. It also supports translation of queries submitted by the OLAP engine expressed in an application specific language to the language the underlying database server speaks.

The OLAP engine is a bridge between the applications and the database server. Some operations specific to analytical processing, such as slicing, dicing, drilling up and down, can be sometimes handled directly by the OLAP engine without going to the database server. If an operation is not supported by the database server, the OLAP engine provides it.

The above logical architecture can be mapped to a so-called relational OLAP system by using a relational database server at the back-end and corresponding add-on components to support OLAP applications. It can also be mapped to a multidimensional OLAP system by using a native multidimensional database server, and the OLAP engine is integrated in the database server.

There are also issues about how to map other OLAP functionalities. For example, for OLAP purposes, the CUBE operator was proposed in [42]. The question arises whether to implement...
the operator in the OLAP engine, or to extend the database server to provide a CUBE operator.

There are similar questions of how to map the following functionalities: maintenance of materialized views, answering queries using pre-computed aggregates, indexing aggregates, query language extension, data model extension, etc.

Although the mapping of the OLAP functionalities can be done in either way, in order to support Data Warehousing there are some necessary extensions at the back-end, i.e., at the database server. In the next section, the impact of Data Warehousing on database technology and the motivations for a novel indexing scheme are discussed.

1.3 Impact on Database Technology and Motivation for Novel Indexing

In [90], [87] and [19], many open issues concerning Data Warehousing are discussed. Among them, issues like modeling, optimization of CUBE computation, query language extension, materialized view maintenance and query processing, have sparked discussion in the past five years [1, 19, 27, 29, 42, 44, 48, 58, 71, 75, 76, 79, 80, 95].

A Data Warehouse maintains data with the ultimate goal to support efficient querying of the data. Query processing and optimization can be achieved by several approaches. In this work, the discussion is confined to indexing and use of indexes.

Conventionally, indexes are used to quickly locate the tuples of interest, without a costly sequential search throughout the whole table. This approach works quite well when the numbers of tuples in question are small. However, unlike OLTP queries, OLAP queries access a large fraction of underlying data and are mostly read-only. In such a scenario conventional indexes, typically B-trees, are not well-suited.

In [64] and [65], the authors proposed algorithms using bitmap indexes, which originated from the Model 204 DBMS in the 1960s [62], for processing OLAP queries. The strength of bitmap indexing is its simplicity. However, the use of bitmap indexing is restricted by the cardinalities of indexed attributes. As the cardinality increases, the index becomes sparser resulting in high processing costs due to poor space utilization. To solve these problems and at the same time keep the simplicity of bitmap indexing, a variation of bitmap indexing, called encoded bitmap indexing, is proposed in [91] and [92].

Encoded bitmap indexing and other variants of bitmap indexing that will also be discussed later enrich the landscape of query processing and optimization in Data Warehouses.

First, bitmap indexing opens new possibilities for selection evaluation. Traditionally, value-list indexes like B-trees cannot be efficiently merged to evaluate multiple selection conditions. Instead, the most selective indexed attribute in the selection conditions is chosen by the query optimizer and is used to retrieve the qualified data, followed by the evaluation of the remaining conditions directly on the selected data. This is simply because merging B-trees involves a matching operation which is as expensive as the join operation. However, different bitmap indexes can be easily and efficiently merged to evaluate complex selection conditions at the same time. Furthermore, the results are also bitmaps. The impact of these properties is two-fold: 1) Because all bitmap-indexed selection conditions are evaluated at the same time, it relieves the query optimizer from the task of looking for the most selective indexed attribute and reduces
the search space for optimal execution plans. 2) The fact that the results of bitmap-merges are also bitmaps implies that subsequent query operations can make use of these bitmaps. In other words, the intermediate results of bitmap processing are still indexed.\footnote{As a matter of fact, using bitmaps to evaluate selections does not produce intermediate results since the conditions are evaluated on bitmaps, not on data. No data is retrieved at that time.} This is what B-trees cannot achieve.

Second, bitmap indexing enriches the implementation of individual query operators. With the help of bitmaps, query operators, such as selection, aggregate and Group By, and Join, are re-implemented.

Third, bitmap indexing changes the philosophy and mechanism of query optimization. Algebra for bitmap processing is introduced into the relational algebra, and through this extension some unnecessary conventional transformation rules for query optimization are invalidated. Such changes and simplifications affect the emphasis of query optimization. The task of optimization is then to find the highest parallelism of bitmap processing and data retrieval, and to balance the load of bitmap processing and data retrieval.

Bitmap indexing thus opens a new branch of query processing in Data Warehousing. The theme of this work is to explore how encoded bitmap indexes and other variations of bitmap indexes can be applied to improve query processing in Data Warehouses. To compare the performance of the bitmap-based query operations, a bitmap-enabled query processor and optimizer — bit\textsc{SQL} — is implemented, and experiments on different sizes of data sets are conducted.

The remainder of this work is organized as follows. In Chapter 2 general issues concerning indexing are introduced and the restrictions of conventional indexes and requirements of indexes in Data Warehousing are discussed. In Chapter 3 encoded bitmap indexing is defined, and its applications in Data Warehousing are introduced. Optimal index design for encoded bitmap indexes is derived through the definition of well-defined encoding. In addition, bit-sliced indexes and issues concerning bitmap index design with respect to both continuous and discrete range queries are discussed. Query processing and optimization techniques using bitmaps are discussed in Chapter 4. The discussion covers selection, aggregate and Group By operations. To understand how the proposed bitmap-based algorithms perform, some experimental results by running 27 test queries on some commercial DBMSs and bit\textsc{SQL} in two different testbeds are presented in Chapter 5. Generally speaking, the performance results of bitmap-based algorithms are quite satisfying and convincing. Bitmap indexing does show its advantages over B-trees, and bitmap-based algorithms show their strength over conventional query processing methods.
Chapter 2

Indexing Data

Building indexes has never been a low-cost task and, in spite of its overhead, its necessity has never been doubted. Imagine that there is a 1000-page book without an index. After several futile trials to locate pages, the readers will curse the author(s) for not providing an index, or even place the book permanently on the highest shelf. The importance of an index becomes obvious as the amount of data (or, the number of pages in the example of a book) grows. Information systems today gather data in such huge volume that searching without the help of an index would be infeasible.

2.1 To Index or Not to Index

To index or not to index is not the question. The questions are: what to index, how to index it and when to use the index.

The first question refers to (in database terms) what table(s) the index should be built on. In Data Warehouses, it is especially interesting to know whether the index is to be built on the fact table or on the dimension table, since different index types will be applied to different types of tables. Simply speaking, for primary key attributes on relatively small tables, traditional value-list indexes, such as B-trees, provide efficient access paths, while for non-key attributes, including foreign-keys, on relatively large tables, bitmap indexes will be a better choice than B-trees. However, under what circumstances one index will outperform the other, is the question answered in this work.

The second question refers to the design of indexes, especially the design of bitmap indexes. There are a lot of variations of bitmap indexes, and each of them is characterized by its own time and space complexity. The design space of bitmap indexes will be introduced in Section 2.3.4, and the design issues concerning encoded bitmap indexes and bit-sliced indexes are discussed in Chapter 3.

The third question, perhaps the most important one, is when to use an index. The answer seems to be quite simple. Whenever users want to select some data of interest, indexes are used to reduce the size of the search space and to locate the desired data directly without inefficient and expensive table scans. In practice, this is not the case. Quite often, using indexes results in worse performance than table scans. We identify those paradoxical cases next.
2.1.1 Paradox between index scans and table scans

Indexes are designed for, but not restricted to, selection evaluation. Theoretically, query operations based on matching, such as Group Bys and Joins, can be carried out by a sequence of selections followed by other relational operators. That is, theoretically they can be processed using indexes. However, in practice, they are not. Even selections for which indexes are available will be evaluated sometimes using table scans instead of using indexes. This paradox is due to the characteristics of I/O subsystems of computers, and using the total processing time as the performance metrics.

To understand this paradox, let us review some query processing basics first. Without loss of generality, queries are assumed to be evaluated by one of the two plans: P1) a table scan, and P2) an index scan followed by data retrieval operations. The cost of query processing is composed of data retrieval cost and index processing cost.\(^1\) The cost of plan P1 consists of data retrieval cost only, since no index is used. On the other hand, the cost of plan P2 consists of both index processing cost and data retrieval cost. Let \(t_{P1}\) and \(t_{P2}\) denote the total costs of P1 and P2, respectively. Using the number of input/output operations (I/Os) to denote the costs of P1 and P2, \(t_{P1}\) and \(t_{P2}\) can be derived as follows.\(^2\)

In computer systems, I/Os are page I/Os. A page is the minimal unit of data which is transferable from/to disks and the main memory. That is, in order to retrieve a desired tuple, the page which contains the tuple will be read from the disk into the main memory. As a result, the number of I/Os is counted by the number of pages being read, not the number of tuples. Consider the following scenario. Suppose we have a table \(T\) which occupies \(n\) disk pages, an index \(I\) on \(T\), and a user query \(q\) on \(T\) with the selectivity, \(s\), which is defined by the ratio of the number of selected tuples over the cardinality of the operand table. In addition, the blocking factor with respect to \(T\) is defined by the number of \(T\)'s tuples resident in a physical page, which further depends on the physical page size and the width of the table \(T\). Now, we would like to know how many pages of \(T\) will be read in order to retrieve the result of the query \(q\). To answer this question, we can map this problem to the well-studied coloring problem.

The coloring problem is defined as follows. Suppose we have \(N\) balls with \(n\) different colors, and each color has \(m\) identical balls, i.e., \(N = m \times n\). If \(k\) balls are randomly selected (\(k \leq N\) without replacement, then the expected number of distinct colors that the \(k\) balls have is defined by

\[
n \times \left( 1 - \prod_{r=1}^{k} \frac{N \cdot \frac{n-1}{n} - r + 1}{N - r + 1} \right). \tag{2.1}\]

In other words, there are \(n\) pages, and each page accommodates \(m\) tuples (the blocking factor). If \(k\) tuples are selected \((k = s \times |T|, \text{ where } |T| \text{ is the cardinality of } T)\), then the expected number of pages being hit is \(n \times \left( 1 - \prod_{r=1}^{k} \frac{N(n-1)/(n) - r + 1}{N - r + 1} \right)\). That is,

\[
t_{P1} = n \quad \text{and} \quad t_{P2} = t_I + n \times \left( 1 - \prod_{r=1}^{k} \frac{N(n-1)/(n) - r + 1}{N - r + 1} \right),
\]

\(^1\)For the sake of simplicity, assume that data retrieval and index scanning are sequential. In the implementation of \(b\)TSQ\(L\), however, the phases of index scanning and data retrieval are overlapped.

\(^2\)Of course, using the number of I/Os to denote the query processing cost does not really reflect the actual cost, but it does demonstrate the existence of the paradox described above.
where \( t_I \) denotes the number of I/Os by index processing. For the sake of simplicity, but without invalidating the whole discussion, assume that \( t_I \approx 0 \), then

\[
\begin{align*}
\text{if } \quad & n = n \times \left( 1 - \prod_{r=1}^{k} \frac{N(n-1)/(n-r+1)}{N-r+1} \right) \\
\Rightarrow \quad & t_{p2} = t_{p1}
\end{align*}
\]

Our intention is to find the boundary condition such that, under that condition, the I/O cost of the data retrieval phase will equal that of a table scan, regardless of how well an index performs. The hit ratio of a table is defined by the number of accessed pages divided by the total number of pages of the table. Then, the boundary condition is the smallest selectivity where the hit ratio equals 1. Figure 2.1 depicts the relationship between the hit ratio of a table and the selectivity of a query with different blocking factors.

![Figure 2.1: Hit ratio subject to different selectivities and blocking factor](image)

In Figure 2.1, we see that for the case in which the blocking factor equals 32, if 7.5% of the tuples in the operand table are selected, then about 91% of the pages in the operand table will be hit; and if the selectivity is about 15%, then over 99% of the pages are hit. What is revealed by this figure is that using indexes for selection evaluation is only profitable if the query is very selective.

Three points need to be clarified here. First, the curves in Figure 2.1 are computed based on the assumption that the tuples are randomly distributed in the table space; this does not really reflect the reality in Data Warehouses where data loading happens in bulk-mode at night. Some degree of clustering in the table space flattens the curves slightly, and thus raises the boundary condition. Second, in the above discussion, the boundary condition is derived based upon 1) sequential query processing and 2) the total processing time as the performance measurement. However, in Data Warehousing, response time is the main performance consideration. In such environments, indexes may provide an opportunity to produce results in sorted order that not only reduces the response time (even for high selectivities), but also enables pipelining successive query operations, which could further reduce the total query processing time. Of course, the index must be able to cope with the nature of page I/Os, \( i.e., \) optimizing operations at the page-level instead of at the tuple-level. We will show that bitmap indexes will do that. Third, using the number of page I/Os to measure the processing cost results in large bias. The purpose of the above discussion is to demonstrate the effect of the page I/O nature of computer systems,
and the limitations of indexes under such systems. For performance evaluation purposes, the disparities between sequential and random I/Os should be taken into account. We revise \( t_{P1} \) and \( t_{P2} \) accordingly as follows.

The time of an I/O operation consists of three parts — the seek time, the rotational latency and the transfer time. When reading \( n \) sequential pages \((n \in \mathbb{N})\), the seek time and the rotational latency occur only once, and the transfer time is proportional to \( n \). In contrast, reading \( n \) random pages requires \( n \) times of all the seek time, the rotational latency and the transfer time. The seek time, the rotational latency and the transfer time are denoted by \( t_s \), \( t_r \) and \( t_x \), respectively. Thus, the solution to the following equation gives the time for which reading \( n \) sequential pages is equal to the time required to read \( m \) random pages.

\[
t_s + t_r + n \cdot t_x = m(t_s + t_r + t_x).
\]

Another factor in this scenario is the autonomy of page allocation of the I/O subsystem. All pages in a logical table space are logically continuous; in practice, they might be scattered across the whole disk. As a result, the actual time required to read \( n \) logically continuous pages depends on the number of blocks which contains physically continuous pages. Let us denote this number by \( b \). The above equation is altered to

\[
b(t_s + t_r) + n \cdot t_x = m(t_s + t_r + t_x).
\]

Taking the effects of sequential and random I/Os into account, the cost functions, \( t_{P1} \) and \( t_{P2} \), are rewritten as

\[
t_{P1} = b(t_s + t_r) + n \cdot t_x
\]

\[
t_{P2} = t_I + m(t_s + t_r + t_x), \quad \text{where } m = n \times \left(1 - \prod_{r=1}^{k} \frac{N(n - 1)/(n - r + 1)}{N - r + 1}\right).
\]

Using indexes increases performance only if

\[
t_{P1} > t_{P2}.
\]

### 2.1.2 Out of the bounding box: Selections

This section discusses how to find a way out of the selection-bounding box for indexes. As shown in the last section, indexes do not have a very large playground in the field of query processing when processing time is the main performance metric and indexes are used for selections only. This is because, no matter how efficient the index is, the whole table will be read to retrieve the results when the selectivity is from medium to high. So, why bother using indexes?

Indexes have a favorable side-effect: they can produce results in sorted order. The effect of this feature is two-fold. First, for expensive matching operations, such as Joins and Group Bys, a low-cost, efficient algorithm can be applied if the inputs are sorted. That means that the

---

Some commercial systems provide further optimization of reading blocks of pages by scheduling the disk arm movements so that it chooses the most efficient sequence of movements to perform successive reads, e.g., the list prefetch mode of DB2. In such cases, the right hand side of the above equation is overestimated.
overhead of using indexes may be amortized by the benefits of a low-cost Join or Group By algorithm. Second, feeding sorted intermediate results enables pipelining successive operations. Pipelining not only reduces the total processing time by overlapping successive operations, but also reduces the response time by producing results stepwise for early response.

Let us illustrate the ideas by the following two examples. The first example shows that even for sequential processing, indexes still provide the potential to reduce the total processing time. The second example shows the favorable effects of enabling pipelining through indexes. Note that the physical data structures of indexes is not defined, since the ideas here are conceptually applicable to any kind of index. At the end of this section, we will go into the issues about how the page I/O nature of computers affects the feasibility of the ideas in practice and its impact on data structures of indexes.

Consider the following scenario. Suppose we have a user query which selects some tuples from a table $T$, followed by a Join to another table $S$, denoted by $\sigma T \bowtie S$ (in the relational algebra). Assume that the selectivity of the query is higher than the boundary condition, i.e., all the pages of $T$ are hit. To evaluate the query, two plans are generated. One employs table scans with binary hash join, and the other invokes index scans with merge-join. For the binary hash join, recursive partitioning is applied on both tables if overflow occurs by hashing. Similarly, if the table to be sorted does not fit into the memory, it is written into several sort runs and then merged into larger and larger runs until only one run is left, i.e., the sorted results.

Figure 2.2(a) depicts the first plan. A table scan is performed on $T$ to retrieve all the desired tuples, and the intermediate results are written to a temporary file\(^4\) which is further hashed into partitions as the probe inputs for the subsequent join. The other table $S$ (the build input of the

\(^4\)The results of the table scan can be direct hashed into partitions, i.e., saving one time of writing $n'$ pages to disks and one time of reading $n'$ pages from disks for the first run of partitioning. We will consider this saving in the cost analysis.
Join) is hashed and partitioned using the same hashing and partitioning functions. Since the recursion depth of partitioning is determined by the size of the build input and is independent of the size of the probe input, the smaller input is chosen as the build input. After hashing on both inputs is finished, the pairs of partition files (from both build and probe inputs) are concatenated to form the join results. The total number of page I/Os of this execution plan, including a factor 2 for writing and reading at the partitioning phase, is

\[ \text{table scan & partitioning } T = n + 2n' \left( \lceil \log_k \frac{m}{M} \rceil - 1 \right) + 2m \cdot \left\lfloor \log_k \frac{m}{M} \right\rfloor + (n' + m), \]

where \( k \) is the fan-out of the partitioning and \( M \) is the memory size in pages. The partitioning fan-out is the number of partition files created (the partitioning outputs) by one run. The maximal partitioning fan-out, \( k \), is determined by the memory size, \( M \), minus one unit of buffer which is reserved for the partitioning input, i.e., \( k = \left\lfloor \frac{M}{c} - 1 \right\rfloor \), where \( c \) is the buffer size in pages [38].

The first two terms of the above cost expression denote the number of I/Os of scanning the table \( T \) and recursively partitioning the selected results. The recursion depth equals \( \lceil \log_k \frac{m}{M} \rceil \), and the subtraction by 1 reflects the fact that the results of table scan are passed directly for partitioning without being materialized. The third term denotes the number of I/Os of recursively partitioning \( S \) until the build input fits into memory. The last term denotes the cost of joining all pairs of build and probe partitions, assuming that the partitioned files fit into memory.\(^5\) Details about how to tune the method, e.g., the effect of data skew, or the choice of fan-out [38], will not be discussed further, since our intention is to show the possibility of better performance by employing indexes in the query evaluation than a traditional hash method.\(^6\)

Figure 2.2(b) depicts the second plan, which employs index scans and a variation of the merge join. First, table \( S \) is sorted on the join attribute using an external sort and the sorted results are written to a temporary file. In general, external sorting algorithms contain two phases — dividing and combining. At the first phase, the inputs are read into the memory, \( M \) pages at a time. An internal sorting algorithm, such as quick-sort, is used to sort the in-memory data, and the sorted data are written to disks, named sort runs. The number of sort runs is determined by \( \lceil \frac{M}{c} \rceil \). In the second phase, the sort runs are merged until only one run is left, i.e., the sorted results. The number of runs being merged at one time is defined as the fan-in, analogous to fan-out in hashing. The fan-in is denoted by \( k \). Next, table \( T \) is selected using multiple index scans on both selection attribute(s) and the join attribute to produce results in the sort order of the join key. The results are not written to disk; instead, they are joined immediately with the sorted inner input. The total number of page I/Os of the second execution plan, including a factor 2 for writing and reading at the sorting phase, is

\[ \text{index scan & selection } T + \text{external sort on } S = \lceil \frac{T}{n} \rceil + n' + m = \left( 2m \cdot \left\lfloor \log_k \frac{m}{M} \right\rfloor + \frac{m}{m} \right). \]

The first two terms of the cost expression denote the cost of index scans and selection evaluation. Since if indexes are used to perform table scans, with caching mechanisms the number

---

\(^5\)The same assumption applies to the second plan, too.

\(^6\)Hash joins and merge joins are two advanced join methods. The main difference of the I/O costs between them is: the recursion depth of the hash join depends only on one input (the smaller one), while that of the merge join depends on both inputs [39, 41]. The reason why using hash join in the first plan is that with known sizes of build and probe inputs, hash join outperforms sort-merge join.
of I/Os is \( n \) in the best case. To reflect the number of I/Os in the average cases, the number of I/Os of scanning \( T \) using an index is a linear function of \( n \), denoted by \( \alpha n \), where \( \alpha \geq 1 \). The third term denotes the number of I/Os of sorting \( S \). For the sake of simplicity, instead of using indexes to select \( S \), an external sort is applied to sort \( S \), such that the following analysis is concentrated on the effects of using indexes on one table only. The last term denotes the number of I/Os needed to join the outer inputs to the inner inputs. The outer inputs are selected using indexes and are directly joined to the inner inputs. Therefore, no I/O is required for the materialization of the outer inputs.\(^7\) In this plan I/Os are random I/Os, while in the first plan the table scan results in sequential I/Os. In the next we will consider the effects of random and sequential I/Os.

Obviously, if \( (t_I + \alpha n) < (n + 2n'([\log_k \frac{M}{2}] - 1) + n') \), then the second plan outperforms the first one. Suppose we have a B*-tree defined on the join attribute of table \( T \) with a list of page ids at the leaf nodes instead of tuple ids. The cost of index scan, \( t_I \), is therefore the number of leaf nodes of the B*-tree. (The non-leaf nodes of the B*-tree are memory-resident.) The number of the leaf nodes is estimated by \( \frac{N}{D} \mu' \), where \( D \) is the degree of the leaf nodes, \( \mu \) is the storage utilization of the leaf nodes and \( N \) is the cardinality of \( T \) [57, 23]. The page ids in the leaf nodes are sorted and the access of the pages of \( T \) is optimized by some prefetch mode, i.e., an optimization of disk arm movements such that the most efficient sequence of performing successive reads is chosen. In addition, the data which have been read but are not yet needed are buffered such that no page in \( T \) is accessed twice, i.e., \( \alpha = 1 \). To reflect the differences between different I/O modes, the estimates from [63] are adopted as rules of thumb; these are shown in Table 2.1.\(^8\)

We can now rewrite the cost expressions of the numbers of I/Os using these estimated I/O times. \( t_I + n \) is rewritten as

\[
\frac{1}{100} \left( \frac{N}{D} \mu + n \right)
\]

(2.2)

As for the first plan, the scanning on \( T \) results in sequential I/Os. However, in the partitioning phase, reads from the temporary files can be sequential, but writes to the temporary files are random I/Os. Therefore, \( n + 2n' \left([\log_k \frac{M}{M}] - 1\right) + n' \) is rewritten as

\[
\frac{1}{400} \left( n + n'\left([\log_k \frac{m}{M}] - 1\right) + n' \right) + \frac{1}{100} \cdot n' \left([\log_k \frac{m}{M}] - 1\right)
\]

(2.3)

with the same assumption that random I/Os are optimized with prefetching.

As a concrete example, consider a table \( S \) of 1000 \times 2^{20} \) pages, and a system with 1 gigabyte of memory. Each physical page has 8192 bytes, and a page id consists of 4 bytes, i.e., \( D = 2048 \). Assuming that the leaf nodes of the B*-tree are \( \frac{3}{4} \) full (i.e., \( \mu = 86\% \)), the blocking factors for

\(^7\)For \( [\log_k \frac{M}{2}] \leq 1 \), i.e., \( m \leq kM \), the total number of page I/Os for the first plan will be \( n + 2n' + 2m + (n' + m) \). The total I/Os for the second plan is \( t_I + \alpha n + 2m + m \).

\(^8\)The figures shown in Table 2.1 were obtained in the mid 90s. The performance of modern I/O subsystems has been improved rapidly. For example, according to a white paper concerning the performance of the IBM FC-RSS (Fibre Channel RAID Storage Server) systems [51], the measurements of the FC-RSS storage server are: 180 MB/second for sequential reads and 130 MB/second for random reads. The performance difference between sequential reads and random reads is greatly reduced. However, to avoid favoring the index-based method more than the hash method the figures from [63] are still applied to find the boundary condition.
Table 2.1: Rules of thumb for I/O costs

<table>
<thead>
<tr>
<th>Random I/O</th>
<th>Sequential I/O</th>
<th>List Prefetch I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{100}$ sec/page</td>
<td>$\frac{1}{100}$ sec/page</td>
<td>$\frac{1}{100}$ sec/page</td>
</tr>
</tbody>
</table>

both $T$ and $S$ are 64, and the fan-out of partitioning (likewise, the fan-in of merge sort) is also 64. Now the boundary condition which satisfies the following inequality can be derived.

\[
t_1 + n < n + 2n' \left( \log_6 \frac{m}{M} \right) - 1 + n' \\
\implies \frac{1}{100} \left( \frac{64n}{2048 \times 0.86} + n \right) < \frac{1}{400} \left( n + n' \left( \log_{64} 1,000 \times 2^{20} - 128 \times 2^{10} \right) + \frac{n'}{100} \left( \log_{64} 1,000 \times 2^{20} - 128 \times 2^{10} \right) - 1 \right) \\
\implies 0.15n + 4n < n + 3n' + 8n' \implies n' > \frac{3.15}{11} n \\
\implies \frac{n'}{n} > 29\% 
\]

This means that under the current assumptions if the selectivity of the query is over 29%, using indexes will improve the performance of query processing. It is a very interesting result. If only selections are considered, indexes improve performance only when the selectivities are very low. However, considering selections in conjunction with joins, indexes improve performance when the selectivities are much higher. This is because the cost of the first plan depends on the size of the selection results, while that of the second plan does not. This result breaks through the bounding box for indexes, which held that they are only applicable to very selective selections.

As mentioned at the beginning of this section, the advantages of feeding sorted results are two-fold. First, with sorted inputs we can use efficient and easy algorithms for costly operations, such as Joins and Group Bys, and the results produced by the algorithms are still sorted for successive operations. Second, since the results are produced in sorted manner, successive operations can be pipelined to reduce the response time by making partial results available earlier and also reduce the processing time by overlapping the successive operations.

Taking the above example, suppose that table $S$ is also selected using index scan, and that the selection and the join operations are pipelined. Then, the total number of I/Os is

\[
t_1 + \alpha n + t_1 + \beta m, 
\]

where $I'$ is an index on the join attribute of $S$, and $t_1$ is the cost of scanning index $I'$. Comparing the cost expression with that of its sequential version, the difference is that, for the join operation, no I/O is needed since both inputs are produced in sort order and directly joined without being materialized. The response time depends on the distribution of the first key value selected. Suppose that the cardinality of the join attribute is $\kappa$ and the attribute values are evenly distributed in both tables, $T$ and $S$. That is, for each key value, there are on average $\frac{|T|}{\kappa}$ and $\frac{|S|}{\kappa}$ tuples selected from $T$ and $S$, respectively. By applying Equation (2.1), the expected numbers of pages being hit by selecting a key value on $T$ and $S$ are

\[
n \times \left( 1 - \prod_{r=1}^{k} \frac{|T|/n - r + 1}{|T| - r + 1} \right), \quad k = \frac{|T|}{\kappa}, \text{ and} \\
m \times \left( 1 - \prod_{r=1}^{l} \frac{|S|/m - r + 1}{|S| - r + 1} \right), \quad l = \frac{|S|}{\kappa}, 
\]
respectively. The number of I/Os of index processing by selecting one key value using \( I \) is \( \frac{t_1}{\kappa} \), and that of using \( I' \) is \( \frac{t_1'}{\kappa} \). The response time, in terms of the number of I/Os, is

\[
\frac{t_1}{\kappa} + \alpha \times \left( 1 - \prod_{r=1}^{k} \frac{|T_n| - r + 1}{|T| - r + 1} \right) + \frac{t_1'}{\kappa} + \beta \times \left( 1 - \prod_{r=1}^{l} \frac{|S_m| - r + 1}{|S| - r + 1} \right).
\]

From the curves in Figure 2.1, we can see that as the selectivity grows, the number of pages hit rapidly approaches the total number of pages of the selected table. Consequently, the method of using indexes to retrieve data in some desired order for pipelining successive Joins or Group Bys is suitable for high cardinality attributes. Nonetheless, if we consider the cases of exploiting interesting orderings [78], the method of using indexes is advantageous, even if the cardinality of the first join attribute is low, since the high cost of selection at the first join will be amortized by successive low-cost Joins or Group Bys. An ordering is interesting if there are multiple joins on the same attribute. In [78], the interesting ordering is introduced in the context of joins on the same attribute. It can be extended to include cases of different query operations, such as selection, Group Bys and Joins, on the same attribute. The issues about optimizing Group Bys and Joins will be further discussed in Section 4.3 (Chapter 4).

New requirements on indexes

As we have seen in the above discussion, we need to index data at both the tuple-level and the page-level because of the nature of page I/O. A tuple-level index should be easily and efficiently transformed to its page-level counterpart. In addition, multiple index scans should be performed efficiently, especially in OLAP applications, where queries usually involve complex selection conditions along multiple dimensions. Traditionally, multiple selections are evaluated by a single index scan using the most selective attribute, followed by evaluation of the remaining conditions on the selected data. Therefore, the temporary results can be produced in the sort order of the join attribute only if the join attribute happens to be the most selective indexed attribute. Although multiple index scans can be performed, they are done by expensive set operations, such as intersection and union, which are as costly as the join operator itself. As a result, although many indexes are built, they are only applied exclusively in practice. Another solution to this problem is to build indexes on multiple attributes (composite indexes). However, the order of the attributes defined in a composite index is significant. As a result, the number of composite indexes will become unmanageable as the number of attributes in question grows, due to combinatorial explosion. Aggregation is another frequent query operation in OLAP systems. If any index is defined on an attribute that does not lose information about the indexed keys, the aggregate on that attribute could have been evaluated on the index itself, instead of reading the base data. To sum up, an index in the Data Warehousing environment should meet the following properties (McDips for short, the initials of all properties).

---

9 In practice, the benefits brought by multiple index scans are determined by the overhead of index intersection (used in Microsoft SQL Server [40]) and the selectivity reduced by applying the second, or further, indexes.

10 Due to immense space requirements and maintenance overheads of large number of composite B-trees, proposals that balance the tradeoffs between resource requirements and performance have been made. Index selection [45, 33, 20] aims to minimize the cost of query processing under a given storage constraint, by choosing a set of optimal indexes, while index merging [21] aims to minimize storage under a given cost constraint.

11 Some indexing techniques, such as Prefix B-trees, use partial keys [5, 32]. These contain only enough information to distinguish between the indexed values as a way of reducing the space needed by the index. Because these do not contain full attribute values, aggregation may not be directly evaluated in such indexes.
• **Multidimensionality**  Multidimensionality means that an index is capable of locating data by conditions along multiple dimensions. This can be achieved by two alternatives. The first is to build a *native* multidimensional index, *i.e.*, the index keys are concatenations of multiple dimension keys. The second is to evaluate multidimensional conditions by operations on all individual one-dimensional indexes.

• **Cooperativity**  An index is cooperative if it can be combined with another index to evaluate multiple selections on both indexed attributes.

• **Duality**  Duality means that an index should contain information about both the physical page addresses where the indexed data resides and the offsets of individual tuples within the pages.

• **Isomorphism**  Suppose that we define a view $\mathcal{V}$ on a table by the projection of an attribute and the tuple-id. An index on that attribute can be treated as a mapping from key values to tuple-ids, $M : \text{key} \rightarrow \text{tuple-id}$. The index is said to be isomorphic to $\mathcal{V}$, if

$$M(d) = k, \text{ iff } (d, k) \in \mathcal{V}.$$  

Simply speaking, an index is isomorphic to the indexed table if the index does not lose any information about the keys and the relationships between the keys and the tuple-ids. Intuitively, without lost information, queries that contain the indexed attribute only can be directly evaluated on the index without accessing the underlying data.

In the literature, *dense*, or *sparse*, indexes are used to refer to primary access paths which record all, or partial, key values in the indexes [35]. This is analogous to the distinction between isomorphic and non-isomorphic indexes, but not identical. Isomorphism of an index refers to the fact that there is a one-to-one and onto mapping between the index and the indexed table. A non-isomorphic index may still be *dense*. In addition, isomorphism is not restricted to describing primary access paths.

• **Polymorphism**  As discussed in the last section, the utility of an index designed for selection evaluation only is very restricted. Polymorphism means that an index can be used to evaluate different query operations, such as selections, grouping, joins and aggregation.

• **Space efficiency**  The space requirement of an index directly and indirectly determines the costs of index construction, index processing and index maintenance. Therefore, the space utilization of an index should be kept high, and the space requirement should be kept low.

To fulfill the above requirements, an indexing technique, first introduced in the 1960s by O’Neil and implemented in the Model 204 DBMS, reveals its strength. It is *bitmap indexing*. Before introducing bitmap indexing, let us first take a brief review of the most widely used indexing technique in database management systems — B-trees and their variations.

### 2.2 Conventional Indexes

Since their introduction by Bayer and McCreight in 1972 [4], B-trees and variants have become the standard secondary access paths in database management systems (DBMSs). B-trees are
well-suited in on-line transaction processing (OLTP) systems because of their dynamics for updates and the balance property. In this section, characteristics of B-trees and their restrictions are briefly presented, to justify the necessity of new index techniques in Data Warehousing.

### 2.2.1 B-trees and variants

A B-tree of degree (order) $M (M \geq 2)$ is a directed tree with the following properties:

1. Each node of the tree can accommodate $M$ data items (keys) and $M+1$ pointers, and each node except the root has at least $\left\lceil \frac{M}{2} \right\rceil$ data items. The keys in a node are sorted, without loss of generality, in ascending order.

2. Each node except the root and the leaves has at least $\left\lceil \frac{M}{2} \right\rceil$ children. The root is a leaf or has at least two children. Each node has at most $M + 1$ children.

3. The height of the tree is the number of nodes visited from the root to the leaf. All leaves are of the same height.

4. For any non-leaf node holding $n$ keys, $k_1, k_2, \ldots, k_n$, $k_1$ is greater than or equal to all keys in its first child, all the keys in the $i$-th child ($1 < i \leq n$) are greater than $k_{i-1}$ and less than or equal to $k_i$, and $k_n$ is less than all keys in the $(n+1)$-th son.

One fundamental operation on B-trees is tree traversal, which is used by operations such as retrieval, insertion and deletion. The cost of tree traversals depends highly on the height of the tree, which in turn depends on the space utilization of the tree. Simply speaking, the space utilization of a B-tree is defined by the ratio of the number of stored data items to the number of data slots allocated in the B-tree. In the literature, many proposals have been made to improve the space utilization.

One variation, named B⁺-tree originated from [56], increases the space efficiency and is defined as follows. Instead of storing entire tuples in the non-leaf nodes, only key values are stored in the non-leaf nodes; data are all stored in the leaf nodes. Through this modification, the degree of the non-leaf nodes is increased (denoted by $M^*$) and the height of the tree will be eventually decreased. The data structures of a B⁺-tree non-leaf and leaf node are depicted in Figure 2.3(a) and (b), where $P_i (0 \leq i \leq M)$ are pointers to subtrees, $K_j (1 \leq j \leq M, M^*)$ are keys, and $D_j (1 \leq j \leq M)$ are data items, which can be either the actual data or the physical disk addresses pointing to the data.

![Figure 2.3: Data structures of B⁺-trees](image)

Another variation of B-trees, named B*-trees, defers splitting by redistributing the keys, and increases the storage utilization. A B-tree is a balanced tree with all leaves at the same level. If an insertion causes an overflow of a node, the node is split in two, and a new key is inserted into the parent-node to keep the tree balanced. In addition, if the insertion causes a further
node-overflow at the upper level, splitting is performed recursively up to the root. Splitting the root increases the height of the tree by 1. The redistribution technique is used to defer splitting. If an overflow occurs, up to \( t - 1 \) sibling nodes are scanned for empty space, instead of splitting. If free space is found, keys (along with pointers) are shifted to gain space for the new key; otherwise, the data of the \( t \) nodes are redistributed uniformly over \( t + 1 \) nodes. As a result, if the tree is large enough, it guarantees the minimal space utilization will be \( \frac{\frac{t}{M+1}}{\frac{t+1}{M}} \).

B\(^+\)-trees that use redistribution techniques are called B\(^++\)-trees.\(^{12}\) Since it is not the main topic of this work to discuss the differences among all these B-tree variants, and in Data Warehousing we are more concerned with retrievals than insertions and deletions, the following discussion is confined to retrieval operations that use B\(^++\)-trees and examine the feasibility of B\(^++\)-trees in Data Warehouses according to the McDips properties.

### 2.2.2 Restrictions of B-trees

Since the daily operations in Data Warehouses involve read-only queries exclusively, and data is loaded and indexes are maintained in batch mode, insertion, deletion and other maintenance operations of B-trees are not the main concerns here. Instead, we discuss how to use B-trees for individual query operations, including selection, Group Bys and Joins. Then the cooperativity of B-trees for multiple query operations within a single query is examined. The discussion is carried out in two parts. The first reviews the aspects concerning the application of B-trees to individual query operations, and the second reviews the aspects concerning the application of B-trees to different query types. As a summary of this section, the discussion is summed up in Table 2.2 showing how useful B-trees are according to the McDips properties.

### Aspects concerning individual query operators

First of all, assume that the B-tree being considered is isomorphic to the indexed table. In simple words, the B-tree completely and exactly models the relationship between the indexed attribute(s) and their holding tuples. Second, tuple-ids are conceptually stored at the leaf nodes of B-trees. (In practice, physical disk addresses are stored.) Thus, the duality between page and tuple levels of B-trees exists.

To examine the feasibility of B-trees for different query operators, i.e., the polymorphism of B-trees, let us consider the minimal set of basic relational operations — \{union, difference, Cartesian-product, selection, projection\}. If it is shown that B-trees can be applied to all the operations in the basic relational operation set, it implies that B-trees are fully functional for all relational operations and are thus polymorphic.

Suppose we have a B-tree on attribute \( A \) of table \( R \), which can be treated as a 2-tuple relation from \( R.A \) to the set of \( R \)'s tuple ids, denoted by \( \{(a, \ t.id) | a \in A, \ t \in R, \ \text{and} \ t.A = a\} \). Obviously, since a B-tree is a relation, relational operations can be performed on it. For example, we can join a B-tree to a table, or we can perform a grouping on a B-tree, etc. As a result, all the

---

\(^{12}\)The naming of B-tree variants is sometimes confusing. Following the convention used by Comer [24], and Chu and Knott [23], B-trees with data on leaves are called B\(^+\)-trees and B-trees using the redistribution overflow technique are called B\(^*\)-trees. In addition, B-trees that apply both improvements are called B\(^++\)-trees. Later in this text, the term “B-trees” denotes the B\(^*\)-trees, except as explicitly stated.
relational operations can be performed on B-trees, not just the selections for which B-trees were originally designed. Thus, B-trees are polymorphic.

Although theoretically all relational operations can be performed on B-trees, in practice B-trees are not used for all query operations. Even for selections, B-trees are applied only when the query is very selective and hits very few data items in the base table, due to the characteristics of page I/Os (as discussed in Section 2.1.1). Exploiting B-trees for Joins (such as the zig-zag join, and indexed nested-loop join) and for Group Bys is rare and restricted [38, 28].

Other than the effects of page I/Os, the reason why B-trees are not used for Joins or Group Bys is that set operators like union and intersection are themselves as costly as the Join or Group By operators. (They are basically matching and duplicate elimination operations.) Intuitively, it will not improve performance if we rewrite one relational operator with another relational operator that has the same computational complexity. As the following shows, $R \bowtie S$ is rewritten as a sequence of unions, intersections, selections and Cartesian Products. (The symbols of relational operators are listed in Appendix F.1.)

$$R \bowtie_{\alpha=\beta} S \implies \bigcup_{\forall v \in J} \pi_{\mid\alpha}(\sigma_{\alpha=v} B_{\alpha}) \times \pi_{\mid\beta}(\sigma_{\beta=v} B_{\beta}), \ J = \alpha \cap \beta.$$  

Suppose that we have B-trees defined on both attributes $\alpha$ and $\beta$, denoted by $B_{\alpha}$ and $B_{\beta}$, respectively. For each value $v$ in $J$, where $J$ is the intersection of $\alpha$ and $\beta$, tuple-ids are retrieved from $B_{\alpha}$ and $B_{\beta}$ using $v$, followed by a Cartesian Product on the two sets of tuple-ids. The results are the joining paths for $\alpha = \beta = v$. Although the results provide exact access paths indicating what tuples of $R$ join to what tuples of $S$, implementing this execution plan will result in poor performance due to the nature of page I/O and the high overhead of intersections and Cartesian Products. As described in Section 2.1.2, the cost of index processing and the cost of data retrieval using the index determines whether or not using the index improves query performance. If index processing itself is as complex as the original query operator, no matter how well the page access sequence is optimized, using the index will perform worse.

A solution to this problem is to precompute and materialize the following relation —

$\bigcup_{\forall v \in J} \pi_{\mid\alpha}(\sigma_{\alpha=v} B_{\alpha}) \times \pi_{\mid\beta}(\sigma_{\beta=v} B_{\beta}),$

which is also named a join-index. We can define a B-tree, denoted by $B_\gamma$, by a relation from $\gamma$ to $\pi_{\mid\gamma} R \times \pi_{\mid\gamma} S$, where $\gamma = \alpha \cap \beta$, and

$$B_\gamma (\gamma, id_1, id_2) = \{(v, tid, tid) \mid v \in \gamma, (tid, tid) \in \pi_{\mid\gamma}(\sigma_{\alpha=v} B_{\alpha}) \times \pi_{\mid\gamma}(\sigma_{\beta=v} B_{\beta})\}.$$

$B_\gamma$ is a 3-tuple relation, and $(\gamma, id_1, id_2)$ is its schema. The B-tree, $B_\gamma$, is isomorphic to $R \bowtie S$. However, it is neither isomorphic to $R$ nor isomorphic to $S$. Precisely speaking,

$$B_\gamma = \pi_{\mid\gamma,R.tid,S.tid} (R \bowtie_{\alpha=\beta} S), \text{ but } (2.4)$$

$$\exists w \in \alpha, \pi_{\mid\alpha,tid}(\sigma_{\alpha=w} R) \neq \pi_{\mid\gamma,tid}(\sigma_{\gamma=w} B_\gamma), \text{ and } (2.5)$$

$$\exists w \in \beta, \pi_{\mid\beta,tid}(\sigma_{\beta=w} S) \neq \pi_{\mid\gamma,tid}(\sigma_{\gamma=w} B_\gamma). \text{ } (2.6)$$

For example, the indexed nested-loop join can be a very efficient join method only if one of the inputs is so small and the other indexed input is so large that the number of page accesses of the index and the smaller input is smaller than the number of pages in the larger input [28].

Although we can first sort the results on either of the tuple-ids and apply techniques of optimizing page access [77] to optimize the I/O time, sorting and other optimization techniques are not done without cost.
In simple words, the join-index can be used to evaluate join operations only. Using the join-index to evaluate selections produces incomplete results. That is, although B-trees are theoretically polymorphic, in practice we design B-trees for specific query operations for performance considerations. This is a consequence of incooperativity of B-trees (which will be discussed next).

As for Group Bys, conventional methods, such as indexed nested-loop\textsuperscript{15}, sort-based or hash-based algorithms, do not exploit any B-tree defined on the grouping attribute(s), since using B-trees for aggregation encounters similar problems to using them for Joins.\textsuperscript{16}

**Aspects concerning the interact of multiple query operations**

In the last section, the applications of B-trees to each individual query operator are discussed. Although B-trees are theoretically polymorphic, dedicated B-trees are designed for different operations for the sake of performance. In this section, it reveals that even if dedicated B-trees are defined, B-trees designed for different purposes do not work together well. Under such circumstances, the benefits of defining dedicated B-trees are diminished, since in most cases B-trees are applied exclusively.

The unsuitability of B-trees in Data Warehousing is demonstrated in three query types.

**Multiple Selections.** Consider the following query with multiple selections — \( \sigma_{a=v \land d=w} R \). Suppose that B-trees are defined on attributes \( a \) and \( d \), denoted by \( B_a \) and \( B_d \). Then, the above query can be evaluated using \( B_a \) and \( B_d \). The execution plan is expressed by the relational algebra as follows.

\[
\sigma_{\text{id} \in \tau} R, \text{ where } \tau = \pi_{\text{id}}(\sigma_{a=v} B_a) \cap \pi_{\text{id}}(\sigma_{d=w} B_d)
\]

Although it shows the cooperativity of B-trees at the algebraic level, in practice, the above plan will not be used because of the high overhead of the intersection operation, which performs costly matching and duplicate elimination.

In Data Warehousing, such multidimensional selection queries are quite common. One solution to the incooperativity of single-dimensional B-trees is to define a multidimensional B-tree. As for the above example, if a B-tree on the composite key \((a, d)\) is defined, denoted by \( B_{a,d} \), the query can be directly evaluated on \( B_{a,d} \).

\[
\sigma_{\text{id} \in \tau} R, \text{ where } \tau = \pi_{\text{id}}(\sigma_{a=d} B_{a,d}) \]

Note that the ordering of the composite keys of B-trees is significant, and partial key search can be evaluated only if the leading keys are selected in the queries. For example, the expres-\textsuperscript{15}Nested-loop aggregation loops for each input item over the outputs accumulated so far. It aggregates the input item either into the appropriate output slot or creates a new output slot and appends it to the outputs. Indexed nested-loop algorithms speed the inner loop by an index on the output. No index on the input is used.

\textsuperscript{16}In spite of the restricted uses of B-trees, there were counter-examples and examples of exploiting them to improve the join operation in the past. The manipulation of tuple identifiers (physical addresses) was considered to improve the join performance [12, 13]. However, they concluded that either merge-join or nested-loops join performs better than manipulating the tuple ids. Another hybrid join combining index nested-loops join and merge-join was used in IBM’s DB2 [22]. This technique sorts the outer input on the join attribute. The sorted outer input is then merge-joined to the leaf nodes of a B-tree on the join attribute of the inner input. The joined tuples contain entire tuples of the outer input and the physical locations of the inner input. The intermediate join results are sorted on the physical locations, and the final results are obtained by merging sorted intermediate join results and the inner input. We see that the method may suffer from the high costs of two sortings.
sion \( (\sigma_{\alpha = v} R) \) can still be evaluated on \( B_{\alpha} \), but \( (\sigma_{\delta = w} R) \). An unfavorable consequence of this solution is the large number of B-tree indexes. For \( n \) dimensions, there will be \( n! \) composite indexes.

**Range Selections.** Another example in Data Warehouse applications is a range-selection of the form \( \sigma_{a \in V} R \), where \( V = \{v_0, v_1, \ldots, v_m\} \). One special case of range selections that is typical in Data Warehouses is selections along dimensional hierarchies. For example, there are six products, \( \{a, b, c, d, e, f\} \), and two categories \( \{c_1, c_2\} \). Products \( a \) and \( f \) make up category \( c_1 \), and products \( b, c, d \) and \( e \) make up category \( c_2 \). Selections on \( c_1 \) are equivalent to selections on \( \{a, f\} \), and selections on \( c_2 \) are equivalent to selections on \( \{b, c, d, e\} \). Usually, when using a B-tree on products to locate \( c_1 \), a traversal from the root down to the leaf node which contains \( a \) is performed, followed by a sequential search on the leaf nodes to find \( f \), instead of activating each individual search from the root. In the worst case, all the leaf nodes will be scanned once. For the above example, to select \( \{a, f\} \) using a B-tree on products with keys sorted in ascending order, all leaf nodes will be read. Another alternative is to begin each individual search with the root. How to optimize the search using B-trees is beyond the scope of this work, and will not be discussed further.

Another straightforward solution is to build a B-tree on category. To build B-trees on all combinations of multiple dimensions along different levels of dimension hierarchies will soon become infeasible, due to the explosively large number of indexes that must be created and the high cost of maintenance as the number of dimensions and/or the number of hierarchy levels increase.

**Involvement of Group Bys, Joins and Selections.** In OLAP applications, most queries involve selections, Joins and Group Bys. (c.f. TPC-H [85]) The fact table is joined to multiple dimensions, named Star-Joins, and selections are issued on both the fact table and dimensions, followed by Group Bys and/or Order Bys to produce sorted, aggregated results. For the sake of performance, dedicated B-trees are defined for each type of query operation. However, different B-trees are either incooperative or cooperative with high overheads. Due to this characteristic of B-trees, conventional query processors or query optimizers are forced to choose a single B-tree and then carry out the rest of the queries on the non-indexed intermediate results.

For example, suppose that we have the query \( \sigma_{a \in V} R \) \( \Join_{\alpha = \beta} S \). Suppose also that there is one B-tree defined on attribute \( \alpha \) and one join-index defined on \( R \Join_{\alpha = \beta} S \). As discussed above, since the join-index is not isomorphic to \( R \), it cannot be used to evaluate selections on \( R \). A conventional query optimizer may use the B-tree on \( \alpha \) to select \( R \) followed by joining the selected data to \( S \) without using the join-index. The reason is simply, as discussed above, the high overhead of performing set intersections on different B-trees. The execution plan (following the notation used above)

\[
\bigcup_{x \in V} \sigma_{\text{tid} \in R(x)} R \times \sigma_{\text{tid} \in S(x)} S, \quad \text{where} \quad R(x) = \pi_{\text{tid}} (\sigma_{\alpha = x} B_\alpha)
\]

\[
S(x) = \pi_{\text{d}_x} (\sigma_{\text{tid} \in R(x)} B_\gamma)
\]

\[
B_\gamma (\alpha, \text{id}_1, \text{id}_2) = \pi_{\alpha, \text{R.tid.S.ud}} (R \Join_{\alpha = \beta} S)
\]

will not even be generated for cost-evaluation by conventional query optimizers.

Consequently, indexes have very restricted utilization in conventional query processing. Although indexes are built and maintained, they are used individually and for most cases exclusively for selections. In addition, since indexes for different query operations do not work
together, conventional query optimization optimizes individual operators, not considering all
the query operations in a query as a whole.

The characteristics of B-trees are summarized in Table 2.2.

<table>
<thead>
<tr>
<th>Multidimensionality</th>
<th>Cooperativity</th>
<th>Duality</th>
<th>Isomorphism</th>
<th>Polymorphism</th>
<th>Space Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieved through multidimensional B-trees, but very high maintenance costs due to combinatorial explosion</td>
<td>very low and achieved by high-overhead set operations</td>
<td>yes</td>
<td>yes</td>
<td>theoretically yes, but in practice infeasible due to high costs of set operations</td>
<td>good</td>
</tr>
</tbody>
</table>

Table 2.2: Quality of B-trees according to McDips properties

2.3 Renaissance of Bitmap Indexes

Bitmap indexing was first introduced in the Model 204 DBMS in the 1960s [62]. The basic idea behind bitmap indexes is to use a string of binary numbers (0 or 1) to indicate whether the indexed attribute in a table is equal to a specific value or not. There are two main differences between value-list indexes, such as B-trees, and bitmap indexes. The first is that in value-list indexes the index keys are modeled and stored as data in the index structures, while they are modeled as metadata in bitmap indexes, i.e., the index keys are not directly stored in the bitmaps, instead they are encapsulated in the bitmaps’ identifiers. The second is that in value-list indexes the address of an indexed tuple is explicitly modeled and stored together with the corresponding index key, whereas in bitmap indexes the addresses of tuples are implicitly modeled by the bit position in the bitmaps.

Due to the simplicity and compact representation of information being indexed, bitmap indexes provide potential for better query performance in Data Warehouses than value-list indexes [65]. In the following section, a brief introduction to bitmap indexes is given. As introduced by O’Neil, these indexes have some restrictions in their application. As part of eliminating these restrictions, which is the motivation of this work, let us take a first look at encoded bitmap indexing in Section 2.3.2 and other variations of bitmap indexing in Sections 2.3.3 and 2.3.5.

2.3.1 The emergence of bitmap indexes

The very first member of the bitmap indexing family, named simple bitmap indexing, uses one bitmap for each index key and the number of bits in each bitmap is equal to the cardinality of the indexed table, i.e., one bit is used to index one tuple. To retrieve data by some specific key value, the corresponding bitmap is scanned and those 1 bits indicate the desired tuples.

To illustrate the idea more concretely, let us look at the “hello, world!” example of the simple bitmap index. Suppose we have an attribute Gender with the domain \{M, F\}, which denotes male and female, respectively. A simple bitmap index on Gender consists of two bit vectors (bitmaps), one for M and the other for F, denoted by \(b_M\) and \(b_F\). The numbers of bits in both bit vectors are equal to the cardinality of the indexed table. The bits of \(b_M\) are set to 1 if the tuples
have \texttt{Gender}=M; otherwise they are set to 0. Similarly, the bits of \texttt{b}_F are set to 1 if the associated tuples have \texttt{Gender}=F; otherwise they are set to 0. To retrieve tuples with \texttt{Gender}=M, the bitmap \texttt{b}_M is used, and the offsets of all the 1 bits within the bitmap are the desired tuples’ identifiers (ids) in the continuous logical table space.\footnote{Without loss of generality, we can view a table space as a continuous memory space, and within a table, the tuple ids as ordinal numbers beginning with 0, such that the physical address of a tuple can be directly derived from its id.}

There are two important points to notice here:

- \textit{The Allowance Of The Value NULL} If, for example, NULL is allowed by the attribute \texttt{Gender}, then \texttt{b}_M \neq \texttt{b}_F, where \texttt{\neg b} is the negation of \texttt{b}. In such cases, we can not use \texttt{\neg b}_M to replace \texttt{b}_F.

- \textit{The Treatment Of Deleted Tuples} For performance reasons, deleted tuples are often tagged as deleted and still reside in the table space, instead of being actually removed from disk. There is a need to maintain an extra bitmap in bitmap indexing to indicate undeleted, or existing tuples. It is called the existence bitmap and is denoted by \texttt{b}_E. Thus, retrieving tuples with \texttt{Gender}=M is slightly different from the procedure described above. Instead of using the bitmap \texttt{b}_M only, the resulting bitmap of \texttt{(b}_M \cdot \texttt{b}_E) is used, where \cdot is the bitwise AND operator. As Figure 2.4 shows, those 1 bits in \texttt{(b}_M \cdot \texttt{b}_E) indicate the result set of the selection, \texttt{Gender}=M.

\begin{figure}[h]
\centering
\begin{tabular}{c|c|c|c|c|c|c|c}
\hline
\multicolumn{2}{c|}{\texttt{Gender}} & \multicolumn{3}{c|}{\texttt{b}_E} & \multicolumn{2}{c}{\texttt{b}_M \cdot \texttt{b}_E} \\
\hline
\texttt{deleted} & & \texttt{b}_F & \texttt{b}_M & \texttt{b}_E & \texttt{b}_M & \texttt{b}_E \\
\hline
M & & 0 & 1 & 0 & 0 & 1 \\
F & & 1 & 0 & 1 & 0 & 0 \\
M & & 0 & 1 & 0 & 0 & 1 \\
M & & 1 & 0 & 1 & 0 & 0 \\
\hline
\end{tabular}
\caption{A classical example of simple bitmap indexing}
\end{figure}

The simple bitmap index is formally defined as follows.

\textbf{Definition 1 (Simple Bitmap Index)} Given a table \(\mathcal{T} = \{t_1, \ldots, t_n\}\), where \(t_j\) is a tuple of \(\mathcal{T}\) \((j = 1, \ldots, n)\), let \(A\) be an attribute of \(\mathcal{T}\), denoted by \(\mathcal{T}.A\), and the domain of \(A\) be \(\{a_1, \ldots, a_m\}\). Then, a simple bitmap index on \(\mathcal{T}.A\), \(\mathbb{B}^A\), is a set of bitmap vectors \(\{\texttt{b}_E, \texttt{b}_1, \ldots, \texttt{b}_m\}\), such that \(\forall \texttt{b}_i\) \((i = 1, \ldots, m)\), \(t_j\) \((j = 1, \ldots, n)\), \(\exists \texttt{b}_i[j] = 1\), if \(t_j.A = a_i\), else \(\texttt{b}_i[j] = 0\), where \(\texttt{b}_i[j]\) denotes the \(j\)-th bit of \(\texttt{b}_i\), and \(\texttt{b}_E\) denotes the existence bitmap.

\textit{Strengths and restrictions of simple bitmap indexing}

For decades, B-trees and their variants (B\textsuperscript{+}-tree, B\textsuperscript{++}-tree, etc.) have been widely used in database systems for external indexing. They provide efficient mechanisms for searching and require time and space only logarithmic to the size of the indexed files. Their strength is their...
dynamic nature, performance and stability under update — properties that are not required in a Data Warehouse. In the Data Warehousing environment, simple bitmap indexing shows its strength over B-trees in two aspects. First, due to its simplicity and the compactness of its data structures, bitmap processing involves low-cost bitwise logical operations only. Second, bitmap indexes provide more potential for query optimization by combining multiple selections, Group Bys and aggregations, and omitting unnecessary Joins. However, since the space requirement of a simple bitmap index is a linear function of the cardinality of the indexed attribute, as the cardinality grows, the efficiency of simple bitmap indexes degrades.

**COOPERATIVITY OF BITMAPS.** If two or more selection conditions are given in a query, say \( \sigma_{\alpha = v} \text{ AND } \beta = w \). separate simple bitmap indexes on \( \alpha \) and \( \beta \) can efficiently work together to fetch the desired data by simply performing a logical operation, \( \text{AND} \), on the corresponding bitmap vectors. In contrast, separate B-trees on attribute \( \alpha \) and attribute \( \beta \) cannot efficiently cooperate with each other to fetch the data. We need to build another B-tree on the composite key \( (\alpha, \beta) \).

The impact of the cooperativity of simple bitmap indexes is that if the top \( n \) attributes with the highest referenced rate in users’ queries are chosen, and indexes are to be built on them, we only need \( n \) simple bitmap indexes. Any combination of selection conditions involving any subset of the \( n \) attributes can be efficiently evaluated by applying corresponding logical operations on the bitmap vectors. The multidimensionality is simply a byproduct of the high cooperativity of bitmap indexing. On the other hand, if B-trees on composite keys are built, in order to cover all possible permutations of selection conditions among these \( n \) attributes, we need at least \( n! \) B-trees. The cost of maintaining so many B-trees would be unacceptable. Therefore, with respect to index cooperativity, simple bitmap indexes have dominating advantages over B-trees.

In addition, the result of bitmap processing is also a bitmap that preserves the address information encapsulated in the bit positions. In other words, the result of bitmap processing is still a bitmap index which can be further used by successive query operations, such as Group Bys and aggregations. In contrast, value-list indexes are used by query processors mostly for selection evaluation. Successive query operations must be evaluated on the non-indexed intermediate result of selections. Details about using bitmaps for query operations other than selections and how bitmaps change query processing and query optimization in Data Warehouses will be discussed in Chapter 4.

**DUALITY, ISOMORPHISM AND POLYMORPHISM.** Since the physical addresses of tuples are implicitly encapsulated in the bit positions and they are automatically stored in ordinal order, the address information can be easily converted to physical page numbers. In Section 4.2.2, page-level bitmaps will be formally defined.

A bitmap index that includes all the attribute values is isomorphic to the indexed table, and a bitmap index is polymorphic, since they are not restricted to use of some specific query operations. In Chapter 4 how bitmap indexes can be used to evaluate different query operations will be discussed.

**PRELIMINARY SPACE REQUIREMENT ANALYSIS.** Let \( \mathcal{T} \) be a table and \( \mathcal{T} = \{t_1, \ldots, t_n\} \). The cardinality of \( \mathcal{T} \) is the number of distinct tuples in \( \mathcal{T} \), denoted by \( |\mathcal{T}| \). Then, building a simple index...
bitmap index on an attribute \( A (A \in \{a_1, \ldots, a_m\}) \) of table \( T \) requires

\[
\frac{|T| \times |A|}{8} = \frac{n \times m}{8} \text{ bytes,}
\]  

(2.7)

where \( m \) is the cardinality of \( A \), denoted by \( |A| \), defined by the number of distinct values in the domain of \( A \). On the other hand, building a B-tree on attribute \( A \) requires about

\[
\frac{1.44 \times n}{M} \times \pi \text{ bytes,}
\]  

(2.8)

where \( \pi \) is the page size, and \( M \) is the degree of the B-tree [24, 23].\(^1\) As for the time complexity, the complexity of building a simple bitmap index on \( A \) is

\[
O(n \times m).
\]  

(2.9)

On the other hand, the complexity of building a B-tree on \( A \) is

\[
O(n \times \log_M (m)) + O(n \times \log_2 (\frac{\pi}{4})),
\]  

(2.10)

where the size of a tuple-id equals four bytes. The first term denotes the cost of traversing from the root to the leaf nodes, and the second term denotes the cost of inserting the tuple-ids into the corresponding leaf nodes.

**Restrictions.** From the requirement analysis above, it is obvious that if \( m > \frac{11.52\pi}{M} \), the space required by simple bitmap indexes is larger than that required by B-trees. For example, assume that \( \pi = 8K \) and \( M = 512 \), then if \( m \geq 185 \), a simple bitmap index requires more space than a B-tree does.

Another problem that comes with high cardinality is the sparsity in bitmaps. The sparsity of a bitmap is on average \( \frac{m-1}{m} \), where \( m \) is the cardinality of the attribute. As \( m \) increases, the space utilization degrades. Large space requirement and poor space utilization also lead to long building and processing times of the indexes.

As a result, simple bitmap indexing is only suitable for very low cardinality attributes. For moderate to high cardinality attributes, a new member of the bitmap indexing family — encoded bitmap indexing — is introduced in this work [91]. Encoded bitmap indexes do not have the restrictions of simple bitmap indexing, but at the same time retain the advantageous properties of bitmap indexing.

### 2.3.2 Encoded bitmap indexes: a first glance

In this section, the basic ideas and usage of encoded bitmap indexing are illustrated by examples. Formal definitions and other design issues, such as well-defined encoding, will be discussed in more detail in Chapter 3. The following working example, a simplification of the schema defined in the TPC-H Benchmark, will be used throughout the rest of the text. The experiments and tests described in Chapter 5 are also carried out on such a database.

\(^1\)Here we just want to give a quick comparison between B-trees and simple bitmap indexes. A detailed analysis of both B-trees and recent bitmap indexes can be found in Chapter 3 (Section 3.3).
Example 1  Consider a data warehouse which models the shipment of orders submitted by some customers, in which each order contains some parts that are supplied by some suppliers. The main concern of the business are the items being shipped for each order, which is modeled as the fact table, lineitems. Around the fact table are five dimension tables: parts, suppliers, orders, customers and nations. The schema of the data warehouse is depicted in Figure 2.5. Note that it is a Snow-flake Schema, and the arrows denote the key to foreign-key relationships.

Figure 2.5: Schema of the experimental database

Suppose that we want to build an encoded bitmap index on the attribute part_id of the fact table lineitems. Assume that the domain of part_id is \{P00001, P00002, \ldots, P01000\}, and the cardinality of part_id is equal to 1000. We first define a mapping function which maps the attribute domain to a set of binary numbers, as Figure 2.6(a) shows. Since $|\text{part_id}| = 1000$, we need at least 10 bits to represent all possible key values. (\(\because \lceil \log_2 1000 \rceil = 10\)) Now, bitmaps are defined based on the encoded values as follows.

First of all, the number of bitmaps is determined by \(\lceil \log_2 |\text{part_id}| \rceil\), which is equal to 10 in our example. For each tuple in lineitems, the key value is mapped to its encoded value and the bitmaps are set bit by bit according to the encoded value. For example, in Figure 2.6(b) the key value of the first tuple in lineitems is $P00002$, which is mapped to $000000001_2$, then the first bits in bitmaps, $b_1, \ldots, b_1$, are set to $0, 0, 0, 0, 0, 0, 0, 0, 0, 1$, respectively. The second bits of $b_2, \ldots, b_2$ are set to $0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0$, respectively, since the encoded key value of the second tuple is $0000000001_2$, and so on. The resulting bitmaps are shown in Figure 2.6(b).

To select data using an encoded bitmap index, the selection predicates are converted into a retrieval Boolean function. To do this, the encoded values are used first to construct the truth table for the Boolean function. For example, suppose that we want to select tuples with (part_id=P00002 OR part_id=P00003). As Figure 2.7(a) shows, the entries in the truth table where the value is equal to the encoded values of either P00002 or P00003 are set to 1; other-
Figure 2.6: Encoded bitmap indexing

wise they are set to 0. Now, we can obtain the Boolean function which is logically equivalent to the truth table by summing up the fundamental conjunctions\(^\text{20}\) whose values in the truth table are 1. Such a Boolean function is called the retrieval Boolean function for the selection. In Figure 2.7(b) the retrieval Boolean function is expressed as a sum of fundamental conjunctions (or, disjunctive normal form) which can be further reduced to

$$
\bar{b}_9 \cdot \bar{b}_8 \cdot \bar{b}_7 \cdot \bar{b}_6 \cdot \bar{b}_5 \cdot \bar{b}_4 \cdot b_3 \cdot b_2 \cdot b_1
$$

The reduced retrieval Boolean function is used to select the desired tuples. In our example, the negations of \(b_9, b_8, \ldots, b_2\) are \(ANDed\) to \(b_1\), and the 1 bits in the resulting bitmap indicate the result set. The retrieval Boolean functions for each attribute value are listed in Figure 2.6(c).

\(^{20}\)Suppose that \(f\) is a Boolean function on the \(n\) variables, \(x_1, x_2, \ldots, x_n\), \((n \in \mathbb{N})\), then a term of the form \(y_1 \cdot y_2 \cdot \ldots \cdot y_n\), where each \(y_i = x_i\) or \(\overline{x_i}, 1 \leq i \leq n\), is a fundamental conjunction. A representation of \(f\) as a sum of fundamental conjunctions is a disjunctive normal form of \(f\) [43].
In the mapping table in Figure 2.6(a), two artificial key values are included — deleted and NULL. The inclusion of NULL can be revoked if NULL is not allowed by the indexed attribute. As for the deleted tuples, it is not a coincidence that the encoded value 0 is chosen. Such a choice is based on performance considerations and will be further discussed in Chapter 3.

To summarize, encoded bitmap indexes reduce the space requirements of bitmaps to a logarithmic function of the indexed attributes’ cardinalities, and at the same time retain the cooperativity of bitmap indexes.

2.3.3 Bit slices

Before introducing bit slices, let us define what a projection index is. Simply speaking, a projection index is a materialized projection on the indexed attribute with duplicates preserved. For example, the projection index on $f_{b fBD a}$ of $f_{b}$ is the materialized projection on $f_{b}$. Figure 2.8(a) shows. A bit-sliced index is a bitwise vertical partition of a projection index. For example, a bit-sliced index on $f_{b fBD a}$ consists of $n$ bitmaps, $b_{n-1}, \ldots, b_{0}$, where $n = \text{sizeof}(price)$ (in terms of the C language), and the bitmap $b_{i}$ ($0 \leq i < n$) is the vertical partition of the $i$-th bit of the projection index on $f_{b}$. Figure 2.8(b) shows the bit-sliced index on $f_{b}$. Figure 2.8 is an example of a binary bit-sliced index. It is binary because the key values are represented in their binary forms. A bit-sliced index can also have a non-binary or even non-uniform base [16, 17]. Before defining the bit slices, key values are first decomposed by the bases. For example, a value $V$ can be expressed by $d_{3}d_{2}d_{1}$ in decimal base, such that $V = d_{3} \times 10^{2} + d_{2} \times 10^{1} + d_{1} \times 10^{0}$; or by $h_{2}h_{1}$ in hexadecimal base, such that $V = h_{2} \times 16^{1} + h_{1} \times 16^{0}$. That is, $V$ can be decomposed into $d_{3}d_{2}d_{1}$ in decimal base, or into $h_{2}h_{1}$ in hexadecimal base. In addition, $d_{3}$ is said to be the digit at position $10^{2}$, $d_{2}$ is the digit at position $10^{1}$, and so on.

For the above example, if we use a decimal base instead of a binary base, the value of $f_{b}$.price is first decomposed into two components, expressed by $p_{2}p_{1}$, as Figure 2.9(a) shows. For each component, a simple bitmap index is defined on it. As Figure 2.9(b) shows, 10 bit vectors are used for each component. $b_{j}^{i}$ denotes the $i$-th bitmap of the $j$-th component. $b_{j}^{i}$ is set to 1 if the key value of the $j$-th component, i.e., $p_{j}$, is equal to $i$; otherwise it is set to 0. Such an assignment of bits is called the equality bit-encoding [16]. The notation $\mathbb{B}_{<b_{N}, \ldots, b_{1}>}$ is used to denote a bit-sliced index with $N$ components and the bases $< b_{N}, \ldots, b_{1}>$. Figure 2.9(c) shows
another example of a bit-sliced index on `lineitems.price` with decimal-decomposition and range bit-encoding [16].

![Diagram of bit-sliced indexes](image)

**Figure 2.9: Bit-sliced indexes with decimal base**

This is called range bit-encoding because the assignment of bits is made by testing a range-predicate. In the above case, \( b_j^i \) is set to 1 if \( p_j \) is smaller than or equal to \( i \); otherwise it is set to 0. In other words, the 1 bits in \( b_j^i \) denote those tuples whose key value at position \( 10^{i-1} \) is smaller than or equal to \( i \). Note that \( b_3^0 \) and \( b_4^0 \) are not needed, since all decimal numbers are smaller than or equal to 9. Therefore, \( b_3^0 \) and \( b_4^0 \) are 1-vectors. The design issues concerning how to choose bases and how different bit-encodings affect the index performance are thoroughly studied in [16] and [17] and will also be discussed in Chapter 3 for the sake of self-containment and completeness of this work.

Bit-sliced indexes are suitable for numeric (fixed-point) or ordinal attributes, and are especially good for wide-range selections and aggregate evaluation. The space requirement of a bit-sliced index is a function of the size of the indexed attribute’s data type and the bases chosen to decompose the key values. In the example above, if an integer consists of four bytes, then the binary bit-sliced index on `price` consists of 32 bitmaps. In [65] and [16], algorithms for evaluating range selection and aggregates using bit-sliced indexes are proposed. The experimental results shown in both [65] and [16] reconfirm the strength of bitmap indexing in the Data Warehousing environment.

In short, encoded bitmap indexes are most suitable for discrete data types, such as alphanumeric data, and bit-sliced indexes are most suitable for continuous data types, such as numeric data. Both are used to optimize selection evaluation. In addition, aggregation can be directly evaluated on bit slices, while encoded bitmap indexes can be used to process query operations other than selections, such as Group Bys, to meet the performance requirements of Data Warehousing. To optimize query processing, both bit slices and encoded bitmap indexes have not
only their own playgrounds, but they also cooperate with each other (the nice feature of bitmap indexing), since queries are likely to involve both types of data. Details of comparing bit-sliced indexes and encoded bitmap indexes will be discussed in Chapter 3.

2.3.4 Design space of bitmap indexes

Since the proliferation of Data Warehouses, bitmap indexing has sparked recent discussions in the area of query processing and optimization. Many variations of bitmap indexes and algorithms using them have been proposed [74, 93, 64, 65, 16, 17, 91, 92, 15]. However, most of the work is concerned with the application of bitmap indexes. The only two streams of work concerning the design of bitmap indexes originate from [91] and [65, 16].

Before introducing different applications of bitmap indexes, let us first take a look at the design space of bitmap indexes, so that we have a clear overview of current development. The design space of bitmap indexes is defined by three dimensions: *value decomposition*, *bit-encoding* and *domain-encoding*. The first two dimensions have been defined in [16]. Here the design space is expanded by adding the third dimension, domain encoding, and by generalizing the second dimension with the definition of *relation bit-encoding*.

The first dimension — value decomposition — runs the spectrum from *minimal decomposition* to *maximal decomposition*. As these terms indicate, minimal decomposition treats the set of key values as a single component, while maximal decomposition decomposes the set of key values into a maximal number of components, i.e., using the binary base to decompose key values. Among others, two important intermediate cases are *time-optimal decomposition* and *space-optimal decomposition* [16]. Time-optimal decomposition is a decomposition of key values such that the number of bitmaps being accessed while processing the index is minimized. In contrast, space-optimal decomposition is a decomposition of key values such that the space requirements of the index are minimized. Simply speaking, time-optimal decomposition saves time in processing the index, while space-optimal decomposition saves space in building the index.

The second dimension — bit-encoding — describes the method of bit assignment. It can be briefly classified into three cases: *equality bit-encoding*, *range bit-encoding* and *relation bit-encoding*. The equality and the range bit-encoding are as described in the previous section. To explain what a relation bit-encoding is, let us define a *relation* as follows.

**Definition 2 (Relation)** For sets $A, B \subseteq U$ (the universal set), $A \times B$ denotes the Cartesian Product of $A$ and $B$ and equals $\{(a, b) | a \in A, b \in B\}$. Then, any subset of $A \times B$ is called a relation from $A$ to $B$.

For example, let $i, k \in \mathbb{Z}$, $\{(0,0), (1,1), (2,2), \ldots\}$ be a relation of $i \times k$. Let $i$ be the subscripts of the bitmaps, and $k$ be the key values, then $b_i$ is set to 1 if $(i,k) \in \{(n,n) | n \in \mathbb{Z}\}$. This results in equality bit-encoding. For range bit-encoding with the predicate $m \leq n$, $b_i$ is set to 1 if $(i, k) \in \{(m,n) | m \leq n \text{ and } m, n \in \mathbb{Z}\}$. We can generalize this bit assignment by accepting any arbitrary relations from $i$ to $k$, and call it relation bit-encoding. Equality and range bit-encoding are two special cases of relation bit-encoding.

Note that the number of components is not introduced in the discussion, since the methods of bit assignment are independent of value decomposition. Nonetheless, for cases where the
The number of components is larger than one — for example, the key value $k$ is decomposed into $N$ components, $k = k_N \cdots k_j \cdots k_1$ — the variable $k$ described above is replaced with $k_j$ for assigning bitmaps in the $j$-th component ($1 \leq j \leq N$).

**Figure 2.10: Design space of bitmap indexes**

The third dimension — domain encoding — describes how to map the attribute domains to another set of values. In Chapter 3 (Section 3.1.2) domain encoding will be discussed in more detail. In Figure 2.10, some models of bitmap indexes are located in the design space. Different classes of bitmap indexes are illustrated in different grey-scale. At the origin is the simple bitmap index, which involves minimal value decomposition, equality bit-encoding and no domain-encoding. To its right, a large area of darker grey covers the class of bit-sliced indexes, and to its right is the binary bit-sliced index. (Simple bitmap indexes and binary bit-sliced indexes are two special cases of generic bit-sliced indexes.) Above binary bit-sliced indexes is the class of encoded bitmap indexes, which employ maximal value decomposition, equality bit-encoding and involve some domain encodings. At the front is the bitmap index class employing arbitrary relation bit-encodings, and on the top are simple bitmap indexes on surrogate keys.

It is not intended to depict all possible classes of bitmap indexes in Figure 2.10. It just serves as a roadmap, to locate an application of bitmap indexing in the space later on. Some holes in the figure are to be filled by upcoming work.

### 2.3.5 Variants of bitmap indexes

In this section, we will see two variants of bitmap indexes that are useful for improving two heavily-used query operators in the Data Warehousing environment — Group By and Join. The discussion here emphasizes the applications of indexes, that is, how the indexes are built and where they are applied. The choice of designs, e.g., which decomposition scheme or bit-encoding scheme should be applied, is another issue at the design time of a Data Warehouse, once the Data Warehouse administrator/designer determines to build the index.
Group-set bitmap indexes

Group By is one of the key operations in Decision Support Systems. (Seventeen of twenty-two queries in the TPC-H Benchmark involve Group By clauses.) Traditionally, Group Bys, like other matching operators such as Join, are performed by two major approaches — sorting or hashing. Both sorting and hashing are costly, especially for large group sets: recursive partitioning on the operand table is performed and that involves extra disk I/Os. In addition, in Data Warehouses the volume of data involved in the aggregation and the Group By clauses may be tera-bytes and no precise result can be given to end users until the whole data set is processed. As a result, the response time is often unacceptable.

Bitmap indexing provides an opportunity to change the method of Group By processing and improve its performance through group-set bitmap indexes. Group-set indexes change the semantics of indexes. Traditionally, we use an index to locate tuples by some key values, and the indexed attribute resides in the indexed table. In contrast, we use a group-set index to locate tuples of one table by some key values of another table. The indexed attribute does not reside in the indexed table.

To explain this more concretely, let us begin with an example. Suppose that we want to sum the qty of lineitems by brand of parts. Conceptually, to evaluate this query, we can choose all tuples from lineitems which belong to one brand and sum their qty(s), then choose tuples of next brand, sum their qty(s), and so on. In other words, we want to locate tuples of the fact table, lineitems, by key values of brand which resides in the dimension table, parts.

To achieve this, a group-set index is built on the fact table using the grouping predicate on the dimension table(s). Figure 2.11 shows a group-set (simple) bitmap index on lineitems grouped by parts.brand.

![Figure 2.11: Group-set bitmap index](image_url)

The bitmaps are constructed as follows. First, define a relation \( R \) from brand to part.id by the SQL query below.

**SQL 1**

```
SELECT brand, part.id
FROM parts
GROUP BY brand, part.id
```

For each distinct value, \( \beta \), of brand, define a simple bitmap \( b_{\beta} \) on lineitems with bits assigned to 1 for all \( t \in \text{lineitems} \), where \( (\beta, t, \text{part.id}) \in R \).

The building of a group-set index depends on key/foreign-key reference(s) and the existence of functional dependencies between the key and the grouping attribute. If the dimension is
normalized (resulting in a Snow-flake Schema), group-set indexes on fact table(s) can also be built using any attribute along the dimension hierarchy, as long as the key/foreign-key references and functional dependencies between the keys and the grouping attribute exist. For example, a group-set bitmap index on \textit{lineitems} grouped by \textit{customers.mktsegment} is shown in Figure 2.12.

To build the above index, the following SQL query is used to find out which orders are submitted by customers who belong to the same \textit{market segment}. The result of the SQL query is assigned to the relation $\mathcal{R}'$.

\textbf{SQL 2}

\begin{verbatim}
SELECT c.mktsegment, o.order_id
FROM orders o, customers c
WHERE o.cust_id = c.cust_id
GROUP BY c.mktsegment, o.order_id
\end{verbatim}

For each distinct value $\mu$ of mktsegment, a simple bitmap $b_\mu$ is built on \textit{lineitems} with bits assigned to 1 for all $t \in \textit{lineitems}$, where $(\mu, t.order_id) \in \mathcal{R}'$.

The usage of group-set bitmaps is quite straightforward. The 1 bits in each bitmap locate those tuples in the fact table that belong to one group. The group-set bitmap can be treated as a selection bitmap which selects tuples belonging to that group. Logically, the grouping using group-set bitmaps is performed by a sequence of selections.

Three points to notice here are: first, in the above examples, simple bitmap indexes are used to build the group-set indexes. Nonetheless, group-set indexes can be implemented by encoded bitmap indexes, or other bit-sliced indexes as well, or even traditional value-list indexes. Second, the group-set index is not only good for grouping, but also good for selection. The most significant feature of group-set indexes in Data Warehouses is that they skip joins of dimension tables to the huge fact table. In \cite{55}, \textit{hash teams} for Joins and Group Bys using bitmaps are built on the fly, instead of precomputation. The shortcomings of hash teams are long processing time (at run time) and the possibility of \textit{false drops}, while group-set indexes require preprocessing, prior to query processing. However, the return on investment of building group-set indexes is high and the cost of building them is low. In Chapter 5 (Section 5.3), the time of building group-set indexes is measured at the time of loading warehouse data. Third, it is a common scenario in Data Warehouses that data are grouped by multiple dimensions. Grouping by multiple attributes can be handled by logical combinations of single group-set indexes. If value-list indexes are used to implement group-set indexes, the number of indexes explodes as the number of dimensions grows. In contrast, due to the cooperativity of bitmaps, bitmap indexes handle multiple groupings by combining bitmaps of each individual grouping at query
processing time. The number of group-set bitmap indexes grows linearly to the number of dimensions.

**Bitmapped join-indexes**

Join is a fundamental operation in the relational model that allows relations to be combined. Because Joins are expensive, many efforts have been made to implement joins, for both *ad hoc* and predefined queries, in the most efficient way possible. One of the approaches is to precompute the join operation and maintain pairs of identifiers of tuples that match in the join. This precomputed access path is called a *join index*, and algorithms using join indexes have been proven to be faster than *ad hoc* methods, such as hybrid hash joins [86]. Naturally, these algorithms require the join indexes to be precomputed.

Data Warehouses are characterized by complex, *ad hoc* queries over huge data volume. In Data Warehouses, Joins usually occur in the form of a Star-Join, which joins the fact table to multiple dimension tables. To be able to cover all possible Star-Joins, join indexes would have to be built on all possible combinations of join attributes in all dimension tables. The resulting number of join indexes would be a combinatorial explosion [64].

A bitmapped join index stores the same information, except that it uses bitmaps to locate tuples of the fact table which match the join to the dimension table. Since bitmaps can be easily and efficiently combined with logical operators, and combinations of bitmaps are still bitmaps, which preserve the information about the indexed tuples’ locations, we need to build bitmapped join indexes on single join only. Star-Joins can then be handled by combining individual bitmapped join indexes. Obviously, the number of bitmapped join indexes grows linearly to the number of dimension attributes which are likely to be used to join the fact table.

Figure 2.13 shows one bitmapped join index between the fact table `lineitems` and the dimension table `orders` over the attribute `order_id`. To join the first tuple of `orders` to `lineitems`, the bitmap `b_01` is used to select tuples from `lineitems` which match the join, and so on.

![Figure 2.13: Bitmapped join index](image)

We can see that a bitmapped join index is not different from any other bitmap index. In the above example, the bitmap index is a join index, and at the same time it is also a selection bitmap index as well.

Last, but not least, although many techniques have been proposed to speed up join operations, the best way is to omit Joins. In Chapter 4 we will discuss how to use bitmap indexes to rewrite queries, such that join operations need not be executed.
SUMMARY  In this chapter, we have discussed what role an index plays in query processing. We see that it is not always best to evaluate queries using indexes instead of using table scans. Furthermore, traditional value-list indexes lack efficient cooperativity among indexes on different attributes, such that they are usually applied exclusively for only one query operation (usually the selection), and the remainder of the query will be evaluated on the non-indexed intermediate results. Bitmap indexes, on the other hand, can be efficiently combined by logical operators, such as AND, OR, NOT, and what is even better, the result is still a bitmap which preserves the information about the tuples’ addresses in the table space. In addition, no matter what kind of bitmaps the indexes are (i.e., any instance in the design space), and what purpose the bitmaps have (i.e., for selection, grouping, or joins), they all share their simple data structure — bit string. That is, different query operations, including selections, grouping, aggregation and joins, can be processed and optimized in the same run using bitmaps assuming, of course, that the necessary bitmap indexes are defined. This property of bitmap indexes has great impact on the landscape of query processing and optimization. It changes not only the mechanism of query processing, but also the heuristics used for query optimization. In later chapters, we will discuss how bitmaps affect query processing and optimization in Data Warehouses, and an experimental implementation of a bitmap-enabled SQL engine is used to compare their performance against some commercial database systems.
Chapter 3

Encoded Bitmap Indexes for Data Warehouses

As discussed in the last chapter, the strength of bitmap indexing in Data Warehousing environments is its wide and global application in query processing that will change the mechanism and philosophy of query processing and optimization. However, the original simple bitmap index also shows its limitations in the case of high cardinality domains. In this chapter, we formalize a new indexing technique, encoded bitmap indexing, discuss its applications in Data Warehouses and derive theorems for optimal index design. The space and time complexities of encoded bitmap indexes will be compared with those of B-trees.

3.1 The Definition of an Encoded Bitmap Index

Following the example demonstrated in Section 2.3.2, an encoded bitmap index on the attribute \( \text{part.id} \) of the fact table \( \text{lineitems} \) consists of a mapping table, a set of bit vectors and a set of retrieval Boolean functions, as Figure 2.6 shows. The mapping table stores the pairs of key values and encoded values. Based upon the encoded values, bitmaps of lengths equal to the cardinality of \( \text{lineitems} \) are defined; and also based upon the encoded values, the retrieval Boolean functions for each key value are derived.

The encoded bitmap index is formally defined as follows:

**Definition 3 (Encoded Bitmap Index)** Given a table \( T = \{ t_1, \ldots, t_n \} \), where \( t_j \) is a tuple of \( T \) \((j = 1, \ldots, n)\), let \( A \) be an attribute of \( T \), denoted by \( T.A \), and the domain of \( A \) be \( \{ a_1, \ldots, a_m \} \). Then, an encoded bitmap index, \( B^A \), on \( T.A \) is a set of bitmap vectors \( \{ b_{k-1}, \ldots, b_0 \} \), a one-to-one mapping \( (M^A : A \rightarrow \{ b_{k-1} \cdots b_0 \} | b_i \in \{0, 1\}, i = 0, \ldots, k-1, k = \lceil \log_2 m \rceil \), and \( m > 1 \) \) and a set of retrieval Boolean functions \((\{ f_{a_1}, \ldots, f_{a_m} \})\). The bitmap vectors are defined as follows. \( \forall a_i \) \((i = 0, \ldots, k-1)\), \( t_j \) \((j = 1, \ldots, n)\), \( \exists b_i[j] = 1 \), if \( M^A(t_j.A)[i] = 1 \), else \( b_i[j] = 0 \), where \( b_i[j] \) denotes the \( j \)-th bit of \( b_i \) and \( M^A(t_j.A)[i] \) the \( i \)-th bit (from the least significant bit, LSB, to the most significant bit, MSB) of \( M^A(t_j.A) \). In addition, \( \forall \alpha \in \{ a_1, \ldots, a_m \} \), the retrieval Boolean function for \( \alpha \), \( f_\alpha \), is a \( k \)-variable min-term (fundamental conjunction) \( x_{k-1} \cdots x_0 \), where \( x_i = b_i \), if \( M^A(\alpha)[i] = 1 \), otherwise \( x_i = \overline{b}_i \) \((i = 0, \ldots, k-1)\).

The retrieval Boolean function for the value \( \alpha \) is satisfiable, i.e., an assignment which makes
the Boolean function \( f_{\alpha} \), true, only at the encoded value of \( \alpha \). This property ensures that any retrieval function will retrieve all the tuples, and nothing but the tuples, which contain the value represented by the function.

The mapping function, or the encoding function, as the name of the indexing originated, plays an important role in defining an encoded bitmap index. The one-to-one property of the mapping function is critical. While evaluating queries using encoded bitmaps, encoded values are used to perform the reverse mapping from the encoded value set to the attribute domain, i.e., the mapping function should be reversible in order to correctly retrieve the desired data. In terms of McDips properties, the one-to-one property makes an encoded bitmap index isomorphic to the indexed table. This property is important, since queries can be evaluated on the bitmaps without loss of data in the result set. Encoding will be discussed later in Section 3.1.2. Assuming that the encoding function is given, the creation and maintenance of encoded bitmap indexes is discussed.

### 3.1.1 Building and maintaining encoded bitmap indexes

After the initial loading of the warehouse data, bitmaps are created in bulk mode on the indexed attributes, i.e., multiple indexes on the same table can be simultaneously created in a single run. Simply speaking, building an encoded bitmap index requires two phases. The first one is the construction of the mapping table, and the second one is the creation of the encoded bitmaps.

To construct the mapping table, the cardinality of the indexed attribute should be known. This determines not only the size of the mapping table, but also the number of bit vectors defined in the index. The cardinality of the attribute domain can be counted by a table scan, and the counting can be carried out on the fly while loading the data. In practice, for the sake of performance, the mapping table can be implemented as a hash table, or be physically sorted on the indexed attribute.\(^1\)

After the mapping table is built, the table to be indexed is scanned once for the attribute values which are used to look up the mapping table for the encoded values. The bitmap vectors are set according to the encoded values. The set of encoded bitmaps can be viewed as a bitwise vertical partitioning on the attribute whose values have been replaced with their encoded versions. For the example illustrated in Figure 2.6(b), if we retrieve all the \( i \)-th bits from all encoded bitmaps, the binary number, \( b_0[i]b_8[i] \ldots b_0[i] \), equals the encoded value of \( \text{part}_{-i} \) of the \( i \)-th tuple. The retrieval Boolean function for each value is defined based upon the encoded value. It is so defined that the Boolean function equals 1 (TRUE) at the point of the corresponding encoded value only. For the example shown in Figure 2.6(c), the retrieval Boolean function for \( P_{00001} \) — \( b_0b_8b_7b_6b_5b_4b_3b_2b_1b_0 \) — equals to 1, if and only if we apply the encoded value of \( P_{00001} \) to the function.

As data are updated, the encoded bitmap indexes need to be maintained to reflect the changes. We discuss the maintenance of encoded bitmap indexes in two cases — updates without domain expansion and updates with domain expansion. Updates without domain expansion mean that

\(^1\)In the current implementation of \texttt{bitSQL}, the mapping tables are physically sorted on the attribute values. In addition, statistics about the indexed attribute, such as the selectivity of each value, are maintained in the mapping table. Such statistics can be gathered on the fly while constructing the mapping table without much extra overhead. However, the experimental version of \texttt{bitSQL} does not apply this statistical information yet for query optimization.
the updates do not affect the domain of the indexed attribute. In contrast, updates with domain expansion mean that the new data contain attribute values which do not have a reference entry in the mapping table. As a result, the mapping table needs to be altered, which causes a cascade of changes on bitmaps and retrieval functions.

**Updates Without Domain Expansion** Following the example in Figure 2.6, if a tuple with \( \text{part} \_\text{id} = \text{P00001} \) is appended to table \textit{lineitems}, then the lengths of all bitmaps are increased by 1 bit, and \( \textbf{b}_j[j] = 0, \textbf{b}_8[j] = 0, \cdots, \textbf{b}_1[j] = 0 \) and \( \textbf{b}_0[j] = 1 \), where \( j \) is the offset of the new inserted tuple in table \textit{lineitems}. In this case, the mapping table and the retrieval Boolean functions remain unaffected.

**Updates With Domain Expansion** If a new tuple with \( \text{part} \_\text{id} = \text{P01001} \) is appended to \textit{lineitems}, \textit{i.e.}, the domain of \( \text{part} \_\text{id} \) is now expanded to \{P00001, P00002, \cdots, P01000, P01001\}, then the following equation should be first tested

\[
\lceil \log_2 |A^{(m-1)}| \rceil = \lceil \log_2 |A^{(m)}| \rceil, \tag{3.1}
\]

where \( A \) is the indexed attribute, \( |A^{(m-1)}| \) denotes the cardinality of \( A \) before insertion, and \( |A^{(m)}| \) denotes the cardinality of \( A \) after insertion. In our case, \( |A^{(m-1)}| = 1000 \) and \( |A^{(m)}| = 1001 \), thus \( \lceil \log_2 1000 \rceil = \lceil \log_2 1001 \rceil \). If Equation (3.1) is true, as is the case in our example, then add an entry for the new attribute value into the mapping table, say \( M^A(\text{P01001}) = 1111101000_2 \), and set \( \textbf{b}_i[j] = M^A(\text{P01001})[i] \) (where \( i = 0, \ldots, k-1, j \) is the offset of the new inserted tuple in table \textit{lineitems}, \( \textbf{b}_i[j] \) denotes the \( j \)-th bit of the bit vector \( \textbf{b}_i \), and \( M^A(\text{P01001})[i] \) the \( i \)-th bit of the encoded value (from LSB to MSB) of \( \text{P01001} \)), and let the retrieval function for \( \text{P01001} \) be \( \textbf{b}_9 \textbf{b}_8 \textbf{b}_7 \textbf{b}_6 \textbf{b}_5 \textbf{b}_4 \textbf{b}_3 \textbf{b}_2 \textbf{b}_1 \textbf{b}_0 \). The changes are depicted in Figure 3.1.

![Figure 3.1: Update with domain expansion](image-url)

Although the above update inserts a new entry in the mapping table, it does not affect the number of bitmaps in the index. As time goes on, and updates take place periodically with some
new attribute values, there will be some time that Equation (3.1) is no longer true. Suppose
that at some time in the past attribute values, \{P01002, P01003, \ldots, P01022\}, have been added
to the mapping table, and another tuple with \text{part\_id} = \text{P01023} is to be appended into \text{lineitems}.
At this point in time, \[\log_2 1024 \prec \log_2 1025\]. Thus, the following actions need to be taken to
reflect the changes on data to the encoded bitmap index.

1. Expand the mapping function from \(M^A : A^{(m-1)} \rightarrow \{0, \ldots, b \mid b \in \{0, 1\}, 0 \leq i \leq 0\}\)
to \(M^A : A^m \rightarrow \{0, \ldots, b \mid b \in \{0, 1\}, 0 \leq i \leq 10\}\), where \(A^{(m-1)}\) denotes the attribute
domain before insertion, and \(A^m\) the one after insertion.

2. Add a new entry in the mapping table for the new value – \text{P01023}, say \(M^A(\text{P01023}) = 10000000000(2)\).

3. Add a bitmap vector \(b_{10}\) to the index, and set \(b_{10}\) to 0.

4. Set \(b_{i}[j] = M^A(\text{P01023})[i]\), where \(i = 0, \ldots, 10\) and \(j\) is the offset of the new inserted tuple
in \text{lineitems}. \(b_{i}[j]\) denotes the \(j\)-th bit of the bit vector \(b_i\), and \(M^A(\text{P01023})[i]\) the \(i\)-th bit
of the encoded value (from LSB to MSB) of \text{P01023}.

5. Define the retrieval Boolean function for the new value (based on the encoded value) by
\(b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0\), and revise the existing retrieval functions by \(\text{ANDing} b_{10}\) to
them, as Figure 3.2(c) shows.

\[
\begin{array}{|c|c|c|}
\hline
\text{part\_id} & \text{encoded value} & \text{lineitems} \\
\hline
\text{deleted} & 00000000000(2) & \text{null} \\
\text{P00001} & 00000000001(2) & \text{P00002} \\
\text{P00002} & 00000000010(2) & \text{P00001} \\
\text{P00003} & 00000000011(2) & \text{P00002} \\
\text{P00004} & 00000000100(2) & \text{P00001} \\
\text{P00005} & 00000000101(2) & \text{P01000} \\
\ldots & \ldots & \ldots \\
\text{P01000} & 01111111000(2) & \text{P01001} \\
\text{P01001} & 01111111010(2) & \text{P01023} \\
\ldots & \ldots & \\
\text{NULL} & 01111111111(2) & \text{P01023} \\
\hline
\end{array}
\]

(a) The revised mapping table

\[
\begin{array}{|c|c|c|}
\hline
\text{attribute value} & \text{Retrieval Boolean function} & \text{attribute value} & \text{Retrieval Boolean function} \\
\hline
\text{deleted} & b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0 & \text{NULL} & b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0 \\
\text{P00001} & \ldots & \text{P00002} & \ldots \\
\text{P01000} & \ldots & \text{P01001} & \ldots \\
\hline
\end{array}
\]

(b) Encoded bitmaps, adding a new bitmap \(b_{10}\)

(c) Revised retrieval Boolean functions

Figure 3.2: Updates with domain expansion
Algorithm 1 Bitmap Index Maintenance

Remark: Maintaining encoded bitmap indexes with respect to updates
Input: A new tuple \( t \), an encoded bitmap index \( \mathbb{B}_A \) on attribute \( A \), including the mapping table, denoted by \( M^A \), and a set of bitmaps, \( \{ \ldots, b_1, b_0 \} \), an integer \( j \) indicating the offset in the table space where the new tuple \( t \) is inserted

1) Begin
2) \( \text{if } t.A \text{ is defined in } M^A \)
3) \( \text{for } i = 0 \text{ to } \lceil \log_2 |A| \rceil - 1 \)
4) \( b_i[j] = M^A(t.A)[i]; \)
5) \( \text{next } i \)
6) \( \text{else } /* \text{ update with domain expansions } */ \)
7) \( \text{assign an encoded value for } t.A \text{ and insert it into } M^A \)
8) \( \text{if } \lceil \log_2 |A| \rceil = \lceil \log_2 (|A| - 1) \rceil \)
9) \( \text{for } i = 0 \text{ to } \lceil \log_2 |A| \rceil - 1 \)
10) \( b_i[j] = M^A(t.A)[i]; \)
11) \( \text{next } i \)
12) \( \text{define the retrieval Boolean function } f_{t.A}; \)
13) \( \text{else } /* \text{ insert a new bit vector } */ \)
14) \( \text{add } b_{\lceil \log_2 |A| \rceil - 1} \text{ to } \mathbb{B}_A \text{ and reset } b_{\lceil \log_2 |A| \rceil - 1}; \)
15) \( \text{for } i = 0 \text{ to } \lceil \log_2 |A| \rceil - 1 \)
16) \( b_i[j] = M^A(t.A)[i]; \)
17) \( \text{next } i \)
18) \( \text{for each existing retrieval Boolean function } f_A \)
19) \( f_A = \overline{b}_{\lceil \log_2 |A| \rceil - 1} f_A; \)
20) \( \text{define the retrieval Boolean function } f_{t.A}; \)
21) \( \text{endif} \)
22) \( \text{endif} \)
23) End

A general algorithm for maintaining the encoded bitmap indexes with respect to both types of updates is listed below.

As discussed in Section 2.3.1, two issues — the allowance of the NULL value and the treatment of deleted tuples — also need extra attention in defining an encoded bitmap index. A simple way to deal with these two issues is to define an existence bitmap and an extra bitmap indicating tuples with the indexed attribute equal to NULL. A better way is to include two artificial attribute values — NULL and deleted — in the attribute domain, where NULL indicates those tuples with the indexed attribute equal to NULL, and deleted indicates those deleted tuples and non-existent tuples.2 The mapping function is then defined on the revised domain, including NULL and deleted. This is better than the first method because it reduces the total number of bitmaps defined in the index. In addition, in query processing, the existence bitmap does not need extra processing if the deleted and non-existent tuples are encoded as 0.

For example, consider the following query, which is evaluated against the database instance illustrated in Figure 2.6.

---

2 For some reason, the space in disk pages is not 100% occupied. The space is reserved for later insertions. In such cases, the artificial key “deleted” is used to represent both deleted and non-existent tuples. The terms, deleted tuples and non-existent tuples, are used interchangeably throughout this text.
As Figure 2.7 shows, the retrieval Boolean function for the above selection condition is constructed by summing up the fundamental conjunctions representing each selected value, which can be further reduced to \( \overline{b_0b_5b_7b_0b_5b_4b_2b_1} \). Although it is not explicitly expressed in the above query that only existing tuples are to be selected, it is commonly agreed that the semantics of the query is to exclude those deleted and non-existent tuples. That is, precisely speaking, the retrieval Boolean function for the above selection should be expressed as

\[
(b_0b_8b_7b_6b_5b_4b_2b_1)\overline{b_0b_7b_6b_5b_4b_2b_1b_0},
\]

where the last term restricts the selections to the existing tuples only. By applying rules in the Boolean algebra, we can further reduce the above expression as follows.

\[
(b_0b_8b_7b_6b_5b_4b_2b_1)\overline{b_0b_7b_6b_5b_4b_2b_1b_0} = (b_0b_8b_7b_6b_5b_4b_2b_1) (b_0 + b_8 + b_7 + b_6 + b_5 + b_4 + b_3 + b_2 + b_1 + b_0) = b_0b_8b_7b_6b_5b_4b_2b_1b_0
\]

It is not surprising that the resulting expression is identical to the original one before we consider the exclusion of non-existent tuples. In other words, while evaluating selections using encoded bitmaps, we only need to consider the values that appear in the selection predicate; the derived retrieval function will implicitly exclude deleted and non-existent tuples if the artificial attribute value — deleted (or non-existent) — is mapped to 1 in the mapping table. The following theorem certifies the suggestion to reserve 0 for deleted or non-existent tuples.

**Theorem 1** Let deleted and non-existent tuples of a table, \( \mathcal{T} \), be encoded as 0. Given any selection on attribute \( A \) of \( \mathcal{T} \) on any subset of existing tuples, the corresponding retrieval Boolean expression, \( f_{\sigma(A)} \land \neg f_{\text{deleted}} \), can be reduced to \( f_{\sigma(A)} \), i.e., ignoring the selection on the existing tuples.

**Proof:** Since deleted and non-existent tuples are encoded as 0, then the retrieval function for deleted and non-existent tuples is \( f_{\text{deleted}} = \overline{b_{k-1} \cdots b_0} \), where \( k \) is the number of bitmap vectors and \( k = \lfloor \log_2 |A| \rfloor \). Then, the retrieval Boolean expression for the selection is

\[
f_{\sigma(A)} \land \neg f_{\text{deleted}} = f_{\sigma(A)}(\overline{b_{k-1} \cdots b_0})
= f_{\sigma(A)}(b_{k-1} + \cdots + b_0) \quad (\text{DeMorgan’s Laws})
= f_{\sigma(A)} \quad (\text{Identity Laws})
\]

since \( b_{k-1} + \cdots + b_0 \) is a tautology, or a universal set.

As a result, such an encoding saves time in query processing since any selection on any subset of existing tuples can be evaluated without explicitly taking the retrieval Boolean function for deleted tuples, \( f_{\text{deleted}} \), into consideration, while in other bitmap indexes the existence bitmap must be always ANDed to the resulting bit vector to exclude non-existent tuples.
3.1.2 The encoding

So far, we have not applied any requirement on the mapping (encoding) functions of the encoded bitmap indexes except for the artificial value “deleted”. It is now time to further discuss the effects of encoding to the performance of encoded bitmap indexes. Let us begin with an example.

**SQL 4**

```sql
SELECT part_id, qty
FROM lineitems
WHERE part_id IN ('P00002', 'P00003', 'P00004', 'P00005')
```

Following the mapping table illustrated in Figure 3.2, to evaluate SQL 4, the retrieval Boolean functions of all the selected values are *ORed* to form the retrieval function for the selection, which can be reduced as follows.

\[
\overline{b_{10}} \overline{b_{6}} \overline{b_{7}} \overline{b_{6}} \overline{b_{5}} \overline{b_{3}} \overline{b_{2}} b_{0} + \overline{b_{10}} \overline{b_{6}} b_{7} \overline{b_{6}} b_{5} b_{4} b_{3} b_{2} b_{1} b_{0} + \\
\overline{b_{10}} \overline{b_{5}} b_{7} \overline{b_{6}} b_{5} b_{4} b_{3} b_{2} b_{1} b_{0} + \overline{b_{10}} \overline{b_{5}} b_{6} b_{5} b_{4} b_{3} b_{2} b_{1} b_{0} + \\
\overline{b_{10}} \overline{b_{5}} b_{6} b_{5} b_{4} b_{3} b_{2} b_{1} b_{0}
\]

Two variables are reduced in the final Boolean function. A reducible function is desirable, since a reduction of any Boolean variable means reading one bitmap less during query processing. Obviously, not all selection conditions can result in retrieval Boolean functions which can be further reduced; this depends on the encoding.

In Data Warehousing environments, although users’ queries are ad hoc, there exist some patterns of selections due to the nature of data classifications, e.g., geographical data is classified into continents, regions, nations, cities, and so on. A sales-analysis might query data by region or continent and, once interesting information is found, the user may decide to go into details and query data by nations, or vice versa. Therefore, in spite of the “ad-hoc” nature, selections may follow some specific patterns, e.g., selections along the hierarchies defined on the data. In the above example, we have a hierarchy on the geographical data, namely cities \(\rightarrow\) nations \(\rightarrow\) regions \(\rightarrow\) continents. Hierarchical classification is just one example of possible selection patterns in Data Warehouses. The reason why predefinable selection patterns are interesting to encoded bitmap indexes is that with respect to some specific selections, we can find an *optimal* encoding such that the number of bitmaps being accessed during query processing is minimized.

For the above example, the encoding defined in Figure 3.2 is an optimal encoding with respect to SQL 4, since there does not exist any other encoding such that the retrieval Boolean function for the selection can be reduced by more than two variables. Note that an optimal encoding provides means of static query optimization for those predefinable query types. However, the lack of such an encoding (or the lack of predefinable query types) does not degenerate the usage and the performance of encoded bitmap indexing since the major strength of such indexing is the global applicability to all kinds of query operations, including aggregations, Group Bys and Joins, not just selections. Query processing and optimization techniques using bitmaps will be further discussed in Chapter 4, and the performance of bitmaps in query processing will be presented in Chapter 5 using real cases.

Nevertheless, let us examine the issue of encoding next, since having an optimal encoding is a favorable feature of an encoded bitmap index.
Definition 4 (Optimal Encoding) An encoding of an encoded bitmap index is a function which maps the attribute domain to a set of binary numbers and, for each value in the domain, the function is defined. Such an encoding is said to be optimal with respect to a selection on the attribute if the number of bitmaps being accessed during selection evaluation is minimized.

Now let us come back to the above example. As claimed above, the encoding listed in Figure 3.2 is optimal with respect to SQL 4 since there exists no other encoding such that the selection can be evaluated by reading less than nine bitmaps. In other words, no encoding can result in a more simplified retrieval function for the selection in SQL 4. The reason is quite straightforward. Consider the identity laws of the Boolean algebra.

\[
x \cdot y = x, \quad \text{iff} \quad y = 1
\]

Let \( \overline{b_{10}} \overline{b_0} b_7 \overline{b_0} \overline{b_3} \overline{b_3} b_1 = \overline{b_{10}} b_0 b_7 b_0 b_3 b_4 b_3 b_1 \cdot g(b_2, b_0) \)

\[\implies g(b_2, b_0) = 1, \quad \text{for all } b_2, b_0 \]

\[\implies g(b_2, b_0) = \overline{b_2}b_0 + \overline{b_2}b_0 + b_2 \overline{b_0} + b_2 b_0 \]

That is, a Boolean function (expressed by sum of products) of \( n \) variables can only form a tautology if it sums up all the \( 2^n \) fundamental conjunctions. Using Reductio ad Absurdum, we know that it is impossible for a Boolean function of three variables to form a tautology if the function contains less than \( 2^3 \) fundamental conjunctions. As a result, since there are only four members in the IN-list in SQL 4, \( i.e., \) the retrieval function consists of four fundamental conjunctions) for the best case, two variables can be reduced. Therefore, the encoding listed in Figure 3.2 is optimal.

As we see, the optimality of the encoding is a nice feature to have. However, Definition 4 does not give us further hints about how to find one. With the help of the following concepts, we will define a set of strict rules to examine whether or not an encoding is well-defined. Theorem 2 states that a well-defined encoding is also an optimal encoding. The relationship between the set of optimal encodings and that of well-defined ones is depicted in Figure 3.3. The set defined by Theorem 2 is only a subset of the optimal encodings, \( i.e., \) if an encoding is not well-defined, this does not necessarily imply that it is not optimal.

![Figure 3.3: Relationship between optimal and well-defined encodings](image)

Let us introduce first the concepts of binary distances, chains and prime chains.

Definition 5 (Binary Distance) Given two binary numbers, \( x \) and \( y \), the binary distance between \( x \) and \( y \) is a function, \( \lambda() \), defined by \( \lambda(x, y) = \text{Count}(x \oplus y) \), where \( \text{Count}(z) \) is a function which returns the number of 1 bits in \( z \), and \( \oplus \) is the bitwise XOR operation.

---

Footnote: For the sake of clarity, we only describe the concept verbally here. We will formally discuss the optimality of an encoding, \( i.e., \) how to decide whether the number of bitmaps being accessed during query processing is minimized or not, in Section 3.3.
For example, if \( x = 01010011_2 \) and \( y = 01110111_2 \), then the binary distance of \( x \) and \( y \) is two, denoted by \( \lambda(x, y) = 2 \). Obviously, the binary distance is commutative, i.e., \( \lambda(x, y) = \lambda(y, x) \).

**Definition 6 (Chain)** Suppose we have a set of distinct binary numbers, \( s = \{c_0, \ldots, c_{n-1}\} \) \((n \geq 2)\). A chain in \( s \) is defined as a sequence on \( s \), say \( <c_{s_0}, \ldots, c_{s_{n-1}}> \), such that \( \lambda(c_{s_i}, c_{s_{i+1}}) = 1 \) \((i = 0, \ldots, n-2)\) and \( \lambda(c_{s_{n-1}}, c_{s_0}) = 1 \). The sequence \( <s_0, \ldots, s_{n-1}> \) is a permutation on \( \{0, \ldots, n-1\} \).

**Definition 7 (Prime Chain of Order \( p \))** Given a set of distinct binary numbers, \( s = \{c_0, \ldots, c_{n-1}\} \) and \( |s| = 2^p \) \((p \geq 1, p \in \mathbb{Z}^+)\). A chain on \( s \) is said to be a prime chain of order \( p \), if \( \forall c_i, c_j \ (0 \leq i, j \leq n-1), \exists \lambda(c_i, c_j) \leq p \).

For example, \(<0000_2, 0100_2, 1100_2, 1110_2, 0110_2, 0010_2>\) is a chain on \( \{0000_2, 0010_2, 0100_2, 0110_2, 1100_2, 1110_2\} \), but not a prime chain. \(<0001_2, 1001_2, 1101_2, 0101_2>\) is a prime chain on \( \{0001_2, 1001_2, 1101_2, 0101_2\} \), while no chain can be defined on \( \{011_2, 0111_2, 111_2\} \). A chain can be expressed by a sum of min-terms.\(^4\) For example, the chain \(<6, 7, 15, 14>\) on 4-digit binary numbers is expressed by \( f(w, x, y, z) = \overline{w}xy \overline{z} + \overline{w}xyz + wxy \overline{z} + wxyz \). If there exists a prime chain in a Boolean function, the function is always reducible, e.g., \( f(w, x, y, z) = \overline{w}xy \overline{z} + \overline{w}yz + wxz + wxyz = xy \). (c.f. Property 1)

The concepts of chains come from the observation of Karnaugh maps.\(^5\) A Karnaugh map is a hyper-dimensional graph projected in a two-dimensional plane. As Figure 3.4 shows, the east edge is actually adjacent to the west one, and the north edge is adjacent to the south one. For example, starting with the cell numbered 0 and heading to its west, we arrive at the cell numbered 2, and starting with the cell numbered 2 and heading to its north, we arrive at the cell numbered 10, and so forth.

![Figure 3.4: Karnaugh map](image)

In a Karnaugh map, the cells are numbered so that the binary distance between neighboring cells is one. For example, the cell numbered 0 is neighbored by cells numbered 1, 2, 4 and 8 (every cell has 4 neighbors), and \( \lambda(0, 1) = \lambda(0, 2) = \lambda(0, 4) = \lambda(0, 8) = 1 \). A path in a Karnaugh map is a traversal from a starting cell to an ending cell, where the binary distance from one step to the next is equal to one. A circuit, i.e., a path where the starting cell is the same as the ending cell, in a Karnaugh map is a chain. A prime chain always contains \( 2^p \) cells \((p \in \mathbb{Z}^+)\).

\(^4\)Suppose we have a 4-variable Boolean function \( f(w, x, y, z) \). A min-term of 7, denoted by \( m(7) \), refers to the fact that the fundamental conjunction \( \overline{w}yz \) has value 1. (Derived from the binary number of \( 7 = 0111_2 \).) And if a Boolean function is expressed by a sum of min-terms, denoted by \( f(w, x, y, z) = \sum m(6, 7) \), it is logically equivalent to a sum of the corresponding fundamental conjunctions. For the above example, \( \sum m(6, 7) = \overline{w}yz + \overline{w}yz \).

\(^5\)The pictorial method for finding minimal sums of products of Boolean functions was developed by M. Karnaugh in 1953.
and the binary distance between any pair of cells in the chain is no larger than \( p \). For example, \( <5,7,15,13> \) is a prime chain. The idea behind finding a prime chain is that, for an \( n \)-variable Boolean function, a prime chain of order \( p \) can be expressed by a conjunction of \( (n-p) \) variables, i.e., \( p \) variables are reduced. As remarked above, a reduction of any variable means reading one bitmap less during query processing. A prime chain has the following property.

**Property 1 (Prime Chains)** Given a prime chain of order \( k \) (\( 0 \leq k \leq n \), and \( k \in \mathbb{Z}^+ \)) in an \( n \)-variable Karnaugh graph, the Boolean function representing the chain can be minimized to an \( (n-k) \)-variable term.

Proof: Let \( <v_0, \ldots, v_{2^k-1}> \) be a prime chain and \( v \in \{v_0, \ldots, v_{2^k-1}\} \) be \( n \)-digit binary numbers. We claim that among all the binary numbers in \( \{v_0, \ldots, v_{2^k-1}\} \), there are exactly \( (n-k) \)-common common bits whose values are identical for all \( v \in \{v_0, \ldots, v_{2^k-1}\} \). That is, if we express the binary numbers by \( n \)-variable Boolean conjunctions, and express the prime chain by summing up the conjunctions, then we can minimized the Boolean expression by distributing the \( (n-k) \) common variables over the sum of the \( 2^k \) \( k \)-variable min-terms. This is true because, for \( k \) Boolean variables, there are exactly \( 2^k \) fundamental conjunctions, and the sum of all of them forms a tautology. As a result, by applying the identity laws, the Boolean function representing the prime chain is minimized to an \( (n-k) \)-variable conjunction.

Now, we need to prove that among the binary numbers in \( \{v_0, \ldots, v_{2^k-1}\} \), there are exactly \( (n-k) \)-common common bits whose values are identical for all \( v \in \{v_0, \ldots, v_{2^k-1}\} \). Using Reductio ad Absurdum, assume that only \( (n-k-1) \) common bits among the binary numbers exist. That is, there exist at least two numbers, say \( v_i \) and \( v_j \), such that \( \lambda(v_i, v_j) = k + 1 \). For example, without loss of generality, let

\[
\lambda(v_0, v_1) = \lambda(v_1, v_2) = \cdots = \lambda(v_k, v_{k+1}) = 1,
\]

and \( v_0 \) differs from \( v_1 \) at the first digit; \( v_1 \) differs from \( v_2 \) at the second digit, and so on. Finally, \( v_k \) differs from \( v_{k+1} \) at the \( (k+1) \)-th digit. As a result, \( v_0 \) differs from \( v_{k+1} \) at \( (k+1) \) digits.

\[
\implies \lambda(v_0, v_{k+1}) = k + 1
\]

However, since \( <v_0, \ldots, v_{2^k-1}> \) is a prime chain, \( \forall w, v \in \{v_0, \ldots, v_{2^k-1}\}, \lambda(w, v) \leq k \). The results are contradictory. Therefore, the number of common bits will not be less than \( (n-k) \).

To prove that the number of common bits will not be more than \( (n-k) \), assume that there are \( (n-k+1) \) common bits. That is, \( \forall w, v \in \{v_0, \ldots, v_{2^k-1}\}, \lambda(w, v) \leq k - 1 \). Varying \( (k-1) \) bits results in at most \( 2^{k-1} \) distinct binary numbers, i.e., there must exist some numbers in \( \{v_0, \ldots, v_{2^k-1}\} \), say \( w \) and \( v \), such that \( w = v \). This contradicts the definition of a prime chain. Therefore, the number of common bits will not be more than \( (n-k) \). That is, the number of common bits is exactly \( (n-k) \).

For example, the Boolean function for the prime chain \( <4,6,12,14> \) on 4-digit binary numbers, denoted by \( f(w, x, y, z) = \sum m(4, 6, 12, 14) \), can be minimized to \( xz \).

Now, we are ready to define a well-defined encoding.

**Definition 8 (Well-Defined Encoding)** Suppose we have a subdomain, \( s = \{v_0, \ldots, v_{p-1}\} \) (\(|s| = n, \text{ and } n \geq 2\) ), of an attribute \( A \), and let \( p = \lfloor \log_2 n \rfloor \). A mapping on attribute \( A \),
\( \mathcal{M}^A : A \to \{b_{k-1} \cdots b_0 | b_i \in \{0, 1\}, 0 \leq i \leq k-1, k = \lfloor \log_2 |A| \rfloor \} \), is said to be well-defined with respect to the selection “\( A \ \text{IN} \ s \)” if one of the following conditions is satisfied:

1) if \( n = 2^p \), then there exists a prime chain in \( \{\mathcal{M}^A(v) | v \in s\} \);  
2) if \( 2^p < n < 2^{p+1} \), and \( |s| \) is even, then
   (i) \( \exists s' \subseteq s, |s'| = 2^p, \exists \) there exists a prime chain in \( \{\mathcal{M}^A(v) | v \in s'\} \),
   (ii) there exists a chain in \( \{\mathcal{M}^A(v) | v \in \varnothing \} \) and \( \forall v, v' \in s, \exists \lambda (\mathcal{M}^A(v), \mathcal{M}^A(v')) \leq p + 1 \), and
   (iii) \( \mathcal{M}^A(\cdot) \) is well-defined with respect to “\( A \ \text{IN} \ (s - s') \)”;
3) if \( 2^p < n < 2^{p+1} \), and \( n \) is odd, then
   (i) \( \exists s' \subseteq s, |s'| = 2^p, \exists \) there exists a prime chain in \( \{\mathcal{M}^A(v) | v \in s'\} \),
   (ii) \( \exists w \notin s, \text{ but } w \in A \), \( \exists \) there exists a chain in \( \{\mathcal{M}^A(v) | v \in s \cup \{w\}\} \), and
      \( \forall v, v' \in s \cup \{w\}, \lambda (\mathcal{M}^A(v), \mathcal{M}^A(v')) \leq p + 1 \), and
   (iii) \( \mathcal{M}^A(\cdot) \) is well-defined with respect to “\( A \ \text{IN} \ (s - s') \cup \{w\} \)”.

Definition 8 is recursive. The recursion stops, when \( |s| = 2^p \), where \( p \geq 1 \). Otherwise it divides the given set into two subsets, one containing \( 2^p \) members, the other containing \( |s| - 2^p \). For the first subset, it tests whether or not there exists a prime chain in it; and for the second subset, the definition is applied recursively to test its well-definedness.

**Theorem 2** A well-defined encoding is an optimal encoding.

The proof to Theorem 2 is included in Appendix A.4. Here the correctness of Theorem 2 is shown by using Karnaugh maps. The discussion is divided into two parts. In the first part, the cases where \( |s| \) is even are considered. In the second part, the cases where \( |s| \) is odd are considered.

![Karnaugh maps](image)

**Figure 3.5**: Karnaugh maps

**Case I** If \( |s| = 2^p \) (\( p \in \mathbb{Z}^+ \)), and there exists a prime chain in \( \{\mathcal{M}^A(v) | v \in s\} \), then the retrieval Boolean function for the selection, “\( A \ \text{IN} \ s \)” can be reduced by \( p \) variables (according to Property 1). It is optimal since no more than \( p \) variables can be reduced. For the examples in Figure 3.5(a), for any Boolean function consisting of 8 min-terms, no more than 3 variables can be reduced. The diagram on the left hand side of Figure 3.5(a) depicts an optimal case, while the one on the right hand side results in a Boolean function which is not reducible. Although there does exist a chain, it is not a prime chain. Thus, the encoding for that diagram is not well-defined.

If \( |s| \neq 2^p \), but \( |s| \) is even, as Figure 3.5(b) shows then, if there exists an isolated cell, i.e., there does not exist a chain, the resulting Boolean function will not be reducible. The Boolean function for the diagram on the left hand side of Figure 3.5(b) contains three variables (one variable is reduced), while the one for the right hand side contains all four variables. A well-defined encoding does not result in any isolated cell in a Karnaugh map.
CASE II  If $|s|$ is odd then, no matter what encoding is defined, the retrieval Boolean function will not be reducible. Nevertheless, an isolated cell in a Karnaugh map should always be avoided since, if no cells are isolated, adding a single cell can construct a chain, as the diagram on the left hand side of Figure 3.5(c) shows. Later, in selection evaluation, don't care conditions or dynamic optimization techniques are applied to artificially construct a chain by adding only one cell, such that the retrieval functions can be further reduced. Therefore, a well-defined encoding avoids any isolated cell.

Definition 9 is a revision of Definition 8 for describing the optimal encoding with respect to a set of selection conditions.

Definition 9 (Well-Defined Encoding) Suppose we have a set of (range) selection predicates on attribute $A$, denoted by $P_A = \{p_1, \ldots, p_n\}$, where each $p_i$ (1 ≤ $i$ ≤ n) corresponds to one subdomain of $A$, denoted by $s_1, \ldots, s_n$. The encoding is well-defined if it is well-defined with respect to all $s_i$ (1 ≤ $i$ ≤ n).

In Definition 8, only one subdomain of the indexed attribute was considered. In practice, an encoding may be well-defined with respect to multiple sets that are not necessarily disjoint to each other. As a matter of fact, an encoding which is only well-defined to a single subdomain of an attribute will not result in better performance. In the next section, some typical applications of encoded bitmap indexing in Data Warehouse environments are introduced [91], and examples of well-defined encodings which cover a set of predefined selections are given.

3.2 Examples of Applications

3.2.1 Hierarchy encoding

OLAP data are usually modeled as a Star Schema. In the center of a Star Schema, the fact table models the core data of the business. Around the fact table, multiple dimensions are defined to model the properties of the core data. As the example illustrated in Figure 2.5 shows, the line-items of each order are stored in the fact table and other entities which participate in an "order" activity are stored in dimension tables, such as customers and suppliers. It is common that there exist hierarchies on dimensions. For example, for each customer, the information about her/his nationality is modeled as nation_id in the table customers. Some nations form a region, which in turn belongs to some continent. As a result, a hierarchy, nations → regions → continents, is defined on the customer-dimension, and nations, regions and continents are the hierarchy elements. Typical sale-analysis queries may select data by varying conditions on dimension hierarchies. As Figure 3.6 shows, hierarchies are defined along TIME, GEOGRAPHIC and SALESPOINT dimensions. Sales data of different time-spans for different geographical areas at different granularities are likely to be investigated before any decision is made to promote some products.

Therefore, in spite of the ad hoc nature of OLAP queries, there do exist some selection patterns. For example, consider the geographic dimension. In addition to selecting data by nation, selections by region or by continent are also likely to support drill-ups and drill-downs. Suppose that an international corporation has business in fourteen nations — {1, 2, \ldots, 14}, which reside in six regions and on three continents. As Figure 3.7(a) shows, nations with ids {1, 2} reside
in Western Europe, the ones with ids \{3, 4\} in Eastern Europe, and so on. For the sake of clarity, the dimension hierarchy defined here has a tree structure, i.e., no nation resides across regions, and no region resides across continents. In real world cases, the hierarchy may be a lattice. That is, the relationship between hierarchy elements are \textit{many-to-many}, instead of \textit{one-to-many}. Nevertheless, the many-to-many relationship will not invalidate the following discussion. The hierarchy encoding is applicable to any classification on data.

The mapping table shown in Figure 3.7(b) defines a well-defined encoding with respect to all selections by any region or continent. That is, for any selection in \(P_{\text{nation}_j}\), the encoding is well-defined, where

\[
P_{\text{nation}_j} = \{\sigma_{\text{region}=\rho} | \rho \in \{\text{Western Europe, Eastern Europe, Middle East, Far East, Indochina, North America}\}\} \cup \{\sigma_{\text{continent}=\gamma} | \gamma \in \{\text{Europe, Asia, America}\}\}
\]

For example, to evaluate the selection — \(\sigma_{\text{continent}=\text{Asia}}\), the retrieval Boolean function is ex-
pressed by the sum of min-terms and can be minimized as follows.

\[
\sum m(1100_{(2)}, 1101_{(2)}, 1000_{(2)}, 1001_{(2)}, 1011_{(2)}, 1010_{(2)}, 1111_{(2)}, 1110_{(2)})
\]

\[
= b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0 + b_3 b_2 b_1 b_0
\]

\[
= b_3
\]

The number of bitmaps being read for evaluating \(\sigma_{\text{continent}=\text{Asia}}\) is one. For another example, to evaluate the selection — \(\sigma_{\text{continent}=\text{Europe}}\) — two bitmaps need to be read. The minimized retrieval Boolean function is \(b_0 b_1\). For evaluating any selection in \(P_{\text{nation}_{\text{jd}}}\), the encoding defined in Figure 3.7(b) is optimal, since the number of bitmaps needed to be accessed during query processing is minimized. For other selections in \(P_{\text{nation}_{\text{jd}}}\), the minimized retrieval Boolean functions and the numbers of bitmaps accessed during query processing are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Selection predicate</th>
<th>Retrieval function</th>
<th>Number of bitmap read</th>
</tr>
</thead>
<tbody>
<tr>
<td>region = Western Europe</td>
<td>(b_3 b_2 b_1)</td>
<td>3</td>
</tr>
<tr>
<td>region = Eastern Europe</td>
<td>(b_3 b_2 b_1)</td>
<td>3</td>
</tr>
<tr>
<td>region = Middle East</td>
<td>(b_3 b_2 b_1)</td>
<td>3</td>
</tr>
<tr>
<td>region = Far East</td>
<td>(b_3 b_1)</td>
<td>2</td>
</tr>
<tr>
<td>region = Indochina</td>
<td>(b_3 b_2 b_1)</td>
<td>3</td>
</tr>
<tr>
<td>region = North America</td>
<td>(b_3 b_2 b_1)</td>
<td>3</td>
</tr>
<tr>
<td>continent = Europe</td>
<td>(b_3 b_1)</td>
<td>2</td>
</tr>
<tr>
<td>continent = Asia</td>
<td>(b_3)</td>
<td>1</td>
</tr>
<tr>
<td>continent = America</td>
<td>(b_3 b_2 b_1)</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.1: Minimized retrieval functions and the numbers of bitmaps read for \(P_{\text{nation}_{\text{jd}}}\)

### 3.2.2 Total-order preserving encoding

So far, the examples of encoded bitmap indexing are all given on discrete (alphanumeric) data types. This does not imply that the application of encoded bitmap indexes is restricted to discrete data types. To build an encoded bitmap index on a numeric data type, an encoding which preserves the total-order property among the attribute values is proposed.

A trivial total-order preserving encoding for numeric data is the internal representation of the numeric type. For example, suppose we have a one-byte short integer attribute \(\text{quantity}\). To build an encoded bitmap index on \(\text{quantity}\), let us define the mapping function as the internal representation of fixed-point integers, e.g., the value “8” is encoded as 00001000_{(2)}, and “17” as 00010001_{(2)}. The index consists of eight bitmap vectors with length equal to the cardinality of the indexed table. The mapping table and the set of retrieval functions can be omitted since the internal representation can be directly used. The resulting bitmaps conform to the bit-sliced index proposed by O’Neil and Quass in [65]. Bit-sliced indexes and their variants [16] are very well suited for numerical data, especially for range queries of the form — “\(A < v\)” and “\(A \in (v_0, \cdots, v_n)\)” — are equally important, a total-order preserving encoding may provide a way of optimizing
both types of selections. For example, suppose that the attribute size of the table parts is a one-byte short integer and both types of selections exist in the application. Sometimes users may want to examine the total sales of parts with a specified range of sizes, and sometimes users may want to know the total sales of parts with specific sizes. In the second case, the sizes of interest generally cannot be expressed by a continuous range of numbers. Instead, they are expressed by a set of discrete numbers. For example, if the following selection is predefined — "size IN (2, 4, 6, 9)", the encoding shown in Figure 3.8 both preserves the total-order and is well-defined with respect to the above selection.

<table>
<thead>
<tr>
<th>size</th>
<th>encoded value</th>
<th>size</th>
<th>encoded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deleted</td>
<td>00000000</td>
<td>NULL</td>
<td>00000001</td>
</tr>
<tr>
<td>1</td>
<td>00000010</td>
<td>2</td>
<td>00000011</td>
</tr>
<tr>
<td>3</td>
<td>00000100</td>
<td>4</td>
<td>00000111</td>
</tr>
<tr>
<td>5</td>
<td>00001010</td>
<td>6</td>
<td>00001011</td>
</tr>
<tr>
<td>7</td>
<td>00001100</td>
<td>8</td>
<td>00001101</td>
</tr>
<tr>
<td>9</td>
<td>00001111</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 3.8: Total-order preserving encoding

For continuous range selections, the algorithms proposed in [65] and [16] are still applicable, after replacing the selected values with their encoded values. The query revision can be easily done by a table-lookup in the mapping table. For predefinable discrete range selections, encodings which minimize the number of bitmaps accessed can be defined. For the above example, the minimized retrieval function for "size IN (2, 4, 6, 9)" is \( b_7 b_6 b_5 b_4 b_1 b_0 \).

### 3.2.3 Range-based encoding

Another possible variation of encoded bitmap indexes for numeric data types is to define an encoding based upon ranges instead of single values. However, note that such an index is not isomorphic to its indexed table, since some relationships between the attribute values and the tuples are missing in the index. Nonetheless, such an incomplete index can be quite useful for attributes which are seldom queried by single value, but instead are queried by range, e.g., numerical attributes such as qty, or discount of lineitems. This idea is demonstrated by a simple example.

Suppose that we want to define a bitmap index on lineitems.discount. The domain of discount is between 0 and 100, indicating the percentage of discount of one lineitem in one order. In addition, the following range selections are frequent — "6 \( \leq \) discount \( < \) 10", "8 \( \leq \) discount \( < \) 12", "10 \( \leq \) discount \( < \) 13" and "16 \( \leq \) discount \( < \) 20". According to the predefined selections, the domain of interest is first divided into six disjoint partitions, as Figure 3.9 shows.

![Figure 3.9: Example of using range-based encoding](image)

Next, the set of intervals — \{[6, 8), [8, 10), [10, 12), [12, 13), [13, 16), [16, 20]\} — is encoded as Figure 3.10(a) shows. For range selections, say "8 \( \leq \) discount \( < \) 12", the retrieval function is \( b_5 b_4 b_0 + b_3 b_2 b_1 b_0 \) which can be reduced to \( b_1 b_0 \). The reduced retrieval Boolean functions for all predefined range selections are listed in Figure 3.10(b).
If no range selection is pre-definable, or the ranges are evenly scattered in the attribute domain, which results in many 1-element disjoint partitions, then the range-based bitmap index will resemble an encoded bitmap index on a set of single values instead of on a set of ranges.

<table>
<thead>
<tr>
<th>intervals</th>
<th>encoded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[6,8)</td>
<td>000_{(g)}</td>
</tr>
<tr>
<td>[8,10)</td>
<td>001_{(g)}</td>
</tr>
<tr>
<td>[10,12)</td>
<td>101_{(g)}</td>
</tr>
<tr>
<td>[12,13)</td>
<td>100_{(g)}</td>
</tr>
<tr>
<td>[13,16)</td>
<td>010_{(g)}</td>
</tr>
<tr>
<td>[16,20)</td>
<td>110_{(g)}</td>
</tr>
</tbody>
</table>

(a) Range encoding

(b) Minimized retrieval Boolean functions

Figure 3.10: Using encoded bitmap indexes for range-based indexing

In any case, the concept of well-defined encoding is applicable to range-based bitmap indexes: it applies not only to a given set of ranges, but also to a given partitioning on the ranges. For the above example, subject to the given set of ranges and the partitioning, the encoding defined in Figure 3.10(a) is well-defined.

3.3 Performance Analysis: Analytical Approach

In this section, the space requirements of building/maintaining encoded bitmap indexes and time requirements of using them for query processing are discussed. The performance of encoded bitmap indexing is compared to that of B-trees and to that of its predecessor — simple bitmap indexing.

3.3.1 Performance of encoded bitmap indexing

The space requirement of an encoded bitmap index is the product of the number of bitmap vectors by the cardinality of the indexed table. According to Definition 3, we know that the number of bitmaps in an encoded bitmap index is a logarithmic function of the cardinality of the indexed attribute.

Suppose we have a table, \( \mathcal{I} \), and \( n \) is the cardinality of \( \mathcal{I} \), denoted by \( n = |\mathcal{I}| \). To build an encoded bitmap index on the attribute \( A \) of \( \mathcal{I} \) requires

\[
\frac{\left\lceil \frac{n}{\pi} \right\rceil \cdot \log_2 m}{\pi} + \frac{m(d + \lceil \log_2 \frac{m}{\pi} \rceil)}{\pi} \approx \frac{n k}{\pi} + \frac{m(d + k + 1)}{\pi} \quad \text{pages,} \tag{3.2}
\]

where \( m \) is the cardinality of \( A \), denoted by \( |A| \), \( \lceil \log_2 m \rceil \) is the number of bitmaps, and \( \pi \) is the physical page size in bytes. The first term denotes the pages required to build the bitmaps, and the second term denotes the space required for the mapping table. The notations used in the analysis of encoded bitmap indexes are listed in Table 3.2.

If updates to the indexed table occur, bitmaps must also be maintained to reflect the changes in the base data. Expressing the complexity of maintenance by number of pages to be accessed,
the cardinality of the table \( \mathcal{T} \), denoted by \( |\mathcal{T}| \)

- \( m \) the cardinality of the attribute \( A \), denoted by \( |A| \)

- \( k \) the number of bitmaps, \( k = \lfloor \log_2 m \rfloor \)

- \( d \) the size of \( A \)'s data type, i.e., \( \text{sizeof}(A) \) in language C

- \( \pi \) physical page size in bytes

- \( |\Delta_T| \) the number of tuples in one data loading

- \( \delta \) the cardinality of a range selection

- \( c_e \) the cost of processing encoded bitmap indexes

- \( \gamma \) the number of variables reduced during the minimization of the retrieval Boolean function

Table 3.2: Notations used for analysis of encoded bitmap indexing

The cost of maintaining encoded bitmap indexes is

\[
\frac{|\Delta_T| \cdot \lfloor \log_2 m \rfloor}{8 \cdot \pi} \text{ pages,}
\]

where \( |\Delta_T| \) is the number of tuples in one data loading. This assumes that the cost of lookups in the mapping table is negligible and, for updates with domain expansion, the cost of maintenance is the above expression plus the cost of adding new bitmaps.

The cost of applying an index in query processing often determines the feasibility of the index. An index with high maintenance cost but with low processing cost may still be useful. However, an index with low maintenance cost, but high processing cost, will soon reach the limitation of its usefulness. In Data Warehouses, where queries are mostly read-only and updates are performed in batch-mode, low index processing cost is a more favorable property than low index maintenance cost.

Since in bitmap indexing bitmaps must be read in their entirety, the cost of index processing is expressed by the numbers of bitmaps being read during query processing. In addition, query processing includes evaluation of selection, projection and other query operations. However, in this section, only the costs of retrieving indexes are discussed and analyzed, since retrieving the indexes (including searching and reading) is a basic operation of index processing. It provides an important indication and characterization of the performance of indexes. However, for usage of query optimizers to choose a better plan, the costs of index processing must be analyzed by cases of query processing algorithms.

Let us now examine the cost of bitmap processing for retrievals. Obviously, in the worst case, all the bitmaps in a bitmap index are accessed. That is, the processing cost is the number of bit vectors in the index. On the other hand, in the best case, only one bitmap is read. Generally speaking, let the cost of processing an encoded bitmap index be denoted by \( c_e \), then

\[
c_e = \lfloor \log_2 m \rfloor - \gamma, \tag{3.3}
\]

where \( \gamma \) is the number of variables reduced by minimization of the retrieval Boolean function, and \( 1 \leq c_e \leq \lfloor \log_2 m \rfloor \) (or \( 0 \leq \gamma < \lfloor \log_2 m \rfloor \)).\(^6\)

\(^6\)The retrieval Boolean function for queries which select the whole attribute domain will result in a tautology, i.e., a table scan is performed. We do not consider such a case to be the best case of bitmap processing, even though no bitmaps are read. In practice, by performing a table-lookup on the mapping table, a query processor will know at the start of index processing that the index should be abandoned and that a table scan should be performed instead. The ability to find out that a selection predicate is a tautology is a favorable feature. I will show what this ability will help the query optimizer to further minimize/simplify the execution plan later in Section 4.2.
To determine \( \gamma \), we have to define the **cardinalities of range selections**. The cardinality of a range selection is the size of the interval of the selection.\(^7\) For example, given the following selection — “\( A \text{ IN } \{a, b, c, d\} \)”, then the cardinality of the selection is four, denoted by \( \delta \), and \( 0 \leq \delta \leq |A| \), where \( A \) is the selecting attribute. The term \( \gamma \) in Equation (3.3) is a function of \( \delta \), and is highly dependent on the encoding. The upper bound of \( \gamma \), i.e., for the cases where the optimal encoding exists, can be determined by Property 2. In other words, Property 2 describes the behavior of \( c_e \) if a well-defined encoding for all selections exists.

**Property 2** Suppose we have an encoded bitmap index on \( A \) and a range selection on \( A \) with cardinality equal to \( \delta \). Using the index to retrieve the set of values, in the best case,

\[
\log_2 m - \theta(\delta)
\]

bitmaps are accessed in order to evaluate the range selection. \( \theta(\delta) \) is defined by

\[
\theta(\delta) = \begin{cases} 
  p, & \text{if } (\delta \bmod 2^p) = 0, \\
  q, & \text{if } ((\delta \bmod 2^p) \bmod 2^q+1) = 2^q, 0 \leq q \leq p-1,
\end{cases}
\]

(3.4)

where \( p \) is the largest integer, such that \( \frac{\delta}{2^p} \geq 1 \).

For example, given an attribute \( A \) with domain \( \{a, b, c, \ldots, z\} \), \( |A| = 26 \), and a selection “\( A \text{ IN } \{a, b, c, d, e, f\} \)”, \( \delta = 6 \), in the best case, four bitmaps are read to retrieve data with “\( A \text{ IN } \{a, b, c, d, e, f\} \)”.

\[
\log_2 26 - \theta(6) = 5 - 1 = 4,
\]

since \( \theta(6) = 1 \). The meaning of Equations (3.3) and (3.4) is straightforward. The number of bitmaps which are accessed is upper-bounded by the number of bitmaps in the index, i.e., \( \log_2 |A| \), and as the ranges of the selections change, the numbers of bitmaps accessed during query processing swing between \( \log_2 |A| \) and \( \log_2 |A| - p \), where \( p \) is the largest integer, such that \( \frac{\delta}{2^p} \geq 1 \).

### 3.3.2 Comparing encoded bitmap indexes with B-trees

Before comparing the performance of encoded bitmap indexes with that of B-trees, let us briefly review some basics about B-trees. Only B*-trees are discussed, since there is no reason to assume that primitive B-trees are still used in DBMSs nowadays. The variables used in the analysis for B*-trees are listed in Table 3.3.

Some operations on B-trees, including *insertion*, *deletion* and *traversal*, are fundamental for building, maintaining and retrieving B-trees. It is well known that the cost of these operations is dependent on the height of the tree, which in turn depends on the space utilization of the B-trees. Therefore, the discussion on B-trees is begun with the **space utilization**.

---

\(^7\)Precisely speaking, we are only interested in the **effective** cardinality of range searches. That is, those values in the search range but not in the attribute domain are excluded from the selection predicate. For example, given an attribute \( A \) with domain \( \{a, b, c, d\} \), and a selection “\( A \text{ IN } \{a, b\} \)”, then the effective cardinality of the above selection is two, since \( c \notin A \).
<table>
<thead>
<tr>
<th>NOTATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>the cardinality of the indexed table, i.e., the number of items in leaf nodes</td>
</tr>
<tr>
<td>$M$</td>
<td>the capacity (degree) of the B*-tree leaf nodes</td>
</tr>
<tr>
<td>$M^*$</td>
<td>the capacity (degree) of the non-leaf nodes</td>
</tr>
<tr>
<td>$t$</td>
<td>split factor of the B*-tree leaf nodes, $t \geq 1$</td>
</tr>
<tr>
<td>$t^*$</td>
<td>split factor of the B*-tree non-leaf nodes, $t^* \geq 1$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>minimal loading factor of the B*-tree leaf nodes</td>
</tr>
<tr>
<td>$\mu$</td>
<td>the storage utilization factor at leaf nodes</td>
</tr>
<tr>
<td>$\mu^*$</td>
<td>the storage utilization factor at non-leaf nodes</td>
</tr>
<tr>
<td>$h$</td>
<td>the height of the B*-tree, $1 \leq h$, and $h \in \mathbb{N}$</td>
</tr>
</tbody>
</table>

Table 3.3: Notations used for analysis of B*-trees

According to the results from [57] and [23], the space utilization of a B-tree at leaf nodes is either

$$
\mu = t \cdot \ln \frac{t + 1}{t} \text{ or } (3.5)
$$

$$
\mu = \frac{\phi}{1 - \phi} \ln \frac{1}{\phi} + \mathcal{O}\left(\frac{1}{M}\right), \text{ where } \phi = \frac{\lfloor \frac{M+1}{t+1} \rfloor}{M} (3.6)
$$

respectively. $\phi$ is the minimal loading factor of the leaf nodes. It reflects the fact that when a split occurs, $tM + 1$ data items are redistributed evenly over $t + 1$ nodes. Therefore, there will be at least $\lfloor \frac{tM + 1}{t+1} \rfloor$ data items in any leaf node. The fraction of $\lfloor \frac{tM + 1}{t+1} \rfloor$ over $M$ reveals how full a leaf node is; this is defined as the minimal loading factor.

The two equations converge toward each other as $M$ approaches infinity. (c.f. Appendix A.1) Both equations are steady state functions, i.e., the space utilization will converge to a value. The experimental results in [23] and [57] show that as long as $n$ (the cardinality of the indexed table) is large enough, both equations provide a very good approximation of the space utilization of the B-trees at leaf nodes. For details about the storage utilization problem of B-trees, please refer to [57] and [23]. Here the result is simply applied for further analysis of B-trees and, for the sake of simplicity, Equation (3.5) is applied for later discussion only.

When a B-tree is built on a table with cardinality equal to $n$, the number of leaf nodes of the B-tree equals

$$
\frac{n}{M \cdot \mu}
$$

Then, the height of the B-tree is determined by

$$
h = \lceil \log_{(M^* \cdot \mu^*)} \left( \frac{n}{M \cdot \mu} \right) \rceil + 1,
$$

where $M^*$ and $\mu^*$ are the degree and the space utilization of non-leaf nodes, respectively. The term, $(M^* \cdot \mu^*)$, denotes the expected order of non-leaf nodes ($M^* > M^* \cdot \mu^*$). For the sake of simplicity, the storage utilization is assumed to be the same at all levels, i.e., $\mu = \mu^*$, and the splitting factor for both leaf and non-leaf nodes is also the same, i.e., $t = t^*$. (In Appendix A.2, it is shown that nonhomogeneous splitting factors at both leaf and non-leaf nodes do not have a significant effect on the space requirements and space utilization of the B-tree.) Now, the average path length of retrieving a B-tree is

$$
h = \lceil \log_{(M^* \cdot \mu)} \left( \frac{n}{M \cdot \mu} \right) \rceil + 1, \quad (3.7)
$$
which is equal to the height of the B-tree, since all leaf nodes are at the same level.

Now the space requirements of building a B-tree are derived as follows.

\[
\frac{n}{M \cdot \mu} + \sum_{i=0}^{h-2} (M^* \cdot \mu)^i = \frac{n}{M \cdot \mu} + \frac{(M^* \cdot \mu)^{h-1} - 1}{M^* \cdot \mu - 1} \quad (3.8)
\]

\[
\approx \frac{n}{M \cdot \mu} + \frac{(M^* \cdot \mu)^{h'-1} - 1}{M^* \cdot \mu - 1} \text{ nodes} \quad (3.9)
\]

are required to build a B-tree, where \( h' = \log_{(M^* \cdot \mu)} \left( \frac{n}{M \cdot \mu} \right) + 1 \). The first term of Equation (3.8) denotes the number of leaf nodes, and the second one denotes the approximate number of non-leaf nodes. Note that the term \( \sum_{i=0}^{h-2} (M^* \cdot \mu)^i \) computes the number of non-leaf nodes of a complete B-tree. Therefore, it overestimates the number of non-leaf nodes in most cases. In order not to disadvantage B-trees in comparison with encoded bitmap indexes, \( h \) is substituted with \( h' \) in Equation (3.9) to simulate the best cases of the space requirements of B\(^{++}\)-trees. In Appendix A.3, it is shown that Equation (3.9) is a very good approximation for the best-case space requirements. As a matter of fact, this substitution underestimates the number of non-leaf nodes slightly, disadvantaging encoded bitmap indexes in the comparison as a result. This is to point out a strict boundary condition where encoded bitmap indexing is more space efficient than even the best case of B-trees.

**Space requirements**

Assuming that one physical page is allocated for one node, then Equation (3.9) denotes the number of pages allocated for B-trees. By letting

\[
\left( \frac{nk + k\pi}{8\pi} + \frac{m(d + \frac{k}{8} + 1)}{\pi} \right) \leq \left( \frac{n}{M \cdot \mu} + \frac{(M^* \cdot \mu)^{h'-1} - 1}{M^* \cdot \mu - 1} \right)
\]

\[
\Rightarrow \frac{nk + 8k}{8\pi} + \frac{m(8d + k + 8)}{8\pi} \leq \frac{1}{M \cdot \mu} \left( n + \frac{n - M \cdot \mu}{M^* \cdot \mu - 1} \right) \quad (3.10)
\]

we obtain a boundary condition that encoded bitmap indexes are more space efficient than B-trees. That is, if Inequality (3.10) is true, then encoded bitmap indexes are more space efficient than B-trees. From Figure 3.11(b) we can see that increasing the split factor (SF) from 7 to 100, which increases the storage utilization factor from 0.9618 to 0.995033, does not significantly reduce the space requirement of B-tree. The two curves representing the space requirements of the two B-trees almost overlap. The two points where \( n = 25,0276 \times 2^{20} \) and \( n = 1.070145 \times 2^{20} \) are the boundary points for encoded bitmap indexes (EBIs), with attributes’ cardinalities equal to 10,000,000 and 500,000, respectively. That is, for large tables with cardinalities equal to 25.03 \( \times 2^{20} \) or more, the space required to build an encoded bitmap index on attributes with cardinalities less than or equal to 10,000,000 is less than that to build a B-tree. In addition, the larger the tables are, the larger the differences between the space required by encoded bitmaps and B-trees. As for attributes with smaller cardinalities, the boundary conditions are also smaller. For example, for attributes with \( m = 500,000 \), if the tables’ cardinalities are 1.07 \( \times 2^{20} \) or more, then encoded bitmap indexes are more space efficient than B-trees.

In Figure 3.11, the space requirements of both B-trees and encoded bitmap indexes are sketched by varying split factors and the cardinalities of indexed attributes, where \( \pi = 4096 \) bytes and
Figure 3.11: Boundary analysis with respect to table cardinality, split factor and attribute’s cardinality

$M = M^* = 500$. The $x$-axis denotes the cardinality of the indexed table in $2^{20}$ and the $y$-axis the space requirements in mega bytes. In order to be able to observe the behaviors of both indexes at small $n$, Figure 3.11(a) is redrawn in logarithmic scale in Figure 3.11(b).

Obviously, the larger the indexed table is, the more beneficial the encoded bitmap indexes. This is simply because of the compact representation of indexed values by bitmaps. This result is exactly what is needed in the Data Warehousing environment, where the cardinalities of fact tables are usually very large. (In Chapter 5, time and space requirements of building both B-trees and bitmap indexes are measured in real cases.)

It is also worth mentioning that the curves in Figure 3.11 are computed by assigning $\mu = 0.9618$ (where $M = 500$ and $t = 7$), instead of $\mu = t \cdot \ln \frac{t+1}{t}$. (Table 2 of [57]) It is simply because the steady state equation is applicable only when $n$ and $M$ are large enough. In the data warehousing environment, it is reasonable to assume that the cardinalities of the indexed tables are large (for $n \geq 10 \times 2^{20}$, which yields an error smaller than 0.00005).
In Figure 3.12, further behavior of the space requirements with respect to different table cardinalities and attribute cardinalities is shown. Most of the time, the surface representing the space requirement of EBI is under that of B*-trees. Only when the cardinality of the indexed attribute is very large and that of indexed table is very small, i.e., wide-domain attributes of very small tables, then the surface of EBI is over that of B*-trees. This is not a typical case in the Data Warehousing environment.

Processing costs

Before beginning with the analysis of index processing cost of both B-trees and encoded bitmaps, let us first introduce two types of range queries, which are used in later discussions to compute the cost of query processing using B-trees and encoded bitmaps. The first type, defined as Type I (or, continuous range search), is denoted by “A ≤ α”, where A is the selecting attribute. The second type, defined as Type II (or, discrete range search), is denoted by “A \( \in \{a_1, \ldots, a_k\} \)”. It is also called an IN-list. A single value selection is treated as a special case of range queries. The terms used for the following discussion are defined in Table 3.4.

<table>
<thead>
<tr>
<th>NOTATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \delta )</td>
<td>the selectivity of the range selection, defined by the ratio of the number of selected tuples to the cardinality of the selected table</td>
</tr>
<tr>
<td>( \delta )</td>
<td>the cardinality of the range selection, for Type II queries, it is equal to the number of values in the IN-list clause</td>
</tr>
<tr>
<td>( \epsilon_\alpha(0) )</td>
<td>the time required to initialize an I/O operation, i.e., the seek time plus the rotation latency, ( \epsilon_\alpha(0) = t_s + t_r )</td>
</tr>
<tr>
<td>( \epsilon_x(n) )</td>
<td>the time required to transfer ( n ) data blocks, ( \epsilon_x(n) = n \cdot t_x )</td>
</tr>
<tr>
<td>( \epsilon_\rho )</td>
<td>the time required for a random I/O operation, including the seek time, the rotation latency and the transfer time, ( \epsilon_\rho = \epsilon_\alpha(0) + \epsilon_x(1) )</td>
</tr>
</tbody>
</table>

Table 3.4: Notations used for analysis of B**-trees (continued)

The I/O costs of retrieving B-trees for Type I and II queries are

\[
\epsilon_\rho \cdot \left[ \log_2(\frac{n}{M \cdot \mu}) \left( \frac{n}{M \cdot \mu} \right) + 1 \right] + \epsilon_\alpha(0) + \epsilon_x\left( \frac{sn}{M \cdot \mu} \right),
\]

(3.11)
respectively. The first terms of the two equations denote the cost of traversals from the root to the leaf nodes; the remaining terms denote the cost of scanning the leaf nodes. For Type II queries, since the values are discrete, each match will likely incur a new traversal from the root. Therefore, the traversal cost will be $O(\delta)$ times that of Type I, and there will likely be $O(\delta)$ multi-block I/Os on the leaf nodes. The number of multi-block I/Os is expressed by $O(\delta)$, meaning that the actual number of multi-block I/Os is a function of $\delta$ and $1 \leq O(\delta) \leq \delta$, since not all values in the IN-lists will incur a new search from the root. In practice, most systems store the root node or all the non-leaf nodes in memory to reduce the number of disk I/Os. Therefore, for the sake of simplicity, it is assumed that all the non-leaf nodes are stored in main memory. Then, Equation (3.11) and (3.12) can be simplified and rewritten as

$$\epsilon_a(0) + \epsilon_x\left(\frac{s \cdot n}{M \cdot \mu}\right)$$

and

$$\epsilon_a(0) \cdot O(\delta) + \epsilon_x\left(\frac{s \cdot n}{M \cdot \mu}\right),$$

respectively. The I/O costs of scanning encoded bitmaps for both Type I and II queries is

$$\epsilon_a(0) \cdot c_e + \epsilon_x\left(c_e \cdot \frac{n}{8\pi}\right),$$

where $c_e$ (defined in Section 3.3.1) is the number of bitmaps being read. The number of initial multi-block I/Os is the number of bitmaps to be read.

In Figure 3.13, the cost functions of one B-tree and two encoded bitmap indexes are plotted, where the indexed table has 20M tuples, and the cardinalities of the indexed attributes are 500,000 and 10,000. The curves are plotted with the following parameters: $\mu = 96\%$; $\pi = 8K$ bytes; $\epsilon_a(0) = 0.024$ seconds; $\epsilon_x(1) = 0.0015$ seconds; $c_e = \lceil\log_2 m\rceil$, and $O(\delta) = 1$. By letting $O(\delta) = 1$ and $c_e = \lceil\log_2 m\rceil$, we place encoded bitmap indexing at a disadvantage. That is, all the bitmaps are read during selection evaluation (the worst case for encoded bitmap indexes), while all I/Os on the B-tree are sequential.

Nevertheless, this figure still shows that the processing costs of encoded bitmaps are lower than that of B-trees if the selectivities of selections are beyond about $12\% \sim 15\%$. It reveals that for single selective selections, B-trees show their strength, as in most of the cases in OLTP systems. However, in OLAP systems, queries usually involve large data sets. OLAP queries may produce very small result sets, i.e., they are very selective, but they still touch large data sets due to the nature of statistical analysis. A statistical analysis can only be confident if the sample space is large enough. The result sets are small because they are either summarized through Group By clauses and aggregation functions, or because they are results of refinement by complex selection conditions. In the latter case, the selectivity of each individual selection may be very high, but overall selectivity is low. In such a scenario, B-trees reveal their weaknesses.

Consider the following example. Suppose that the overall selectivity of SQL 5 is only $10\%$, but the selectivities of both individual selections are about $30\%$. Using B-trees to evaluate both selections, the intuitive cost of the multiple index scan will be the sum of both single index scans
plus the cost of performing an intersection on the result sets. On the other hand, if bitmaps are
used, the cost of the multiple index scan will be the total time of reading all the bitmaps of
both indexes (on part\_id and supp\_id) plus a bitwise logical AND on the two result bitmaps.
According to Figure 3.13, the execution plan using B-trees for SQL 5 performs much slower
than that using bitmaps. This is because, at selectivity of 30%, B-trees perform worse than
bitmaps, and in addition, set intersections are much more expensive than bitwise operations.

**SQL 5**

```
SELECT part\_id, supp\_id, qty
FROM lineitems
WHERE part\_id IN ("P00002", "P00003", "P00004", "P00005") AND
      supp\_id IN ("S00002", "S00003", "S00004", "S00005")
```

A consequence of high processing costs of B-trees at high selectivities is: if we want to apply the
index to output a large fraction of the base table in sorted order for other subsequent operations,
such as Group By or Join, then encoded bitmap indexes are better choices than B-trees, since
the index processing costs of bitmap indexes depends on the number of bitmaps only and are
much smaller than those of B-trees at high selectivities.

Another cascading consequence is: with a large number of multiple selections, bitmaps resulting
from each selection evaluation can be efficiently merged by bitwise operations and the final
result is still a bitmap index. On the other hand, expensive set operations must be applied
to merge results of multiple index scans using B-trees, and the final result sets lose the index
structures of B-trees. In other words, any further processing on the results of B-tree evaluation
will also require expensive set operations.

### 3.3.3 Comparing encoded bitmap indexing with simple bitmap indexing

Showing the advantages of encoded bitmaps over B-trees justifies the introduction of a novel in-
dexing technique to DBMSs. Showing the advantages of encoded bitmaps over simple bitmaps
justifies the necessity of changes (improvements) to its predecessor.
Space requirements

As discussed in Section 2.3.1, the space requirements of building a simple bitmap index are

$$\frac{n \cdot m}{8\pi}$$

pages, while the space requirements of building an encoded bitmap index are

$$\frac{nk}{8\pi} + \frac{m(d + \frac{k}{\pi} + 1)}{8\pi}$$

pages.

The notations follow the convention used in this section, and are listed again for ease of reference in Table 3.5.

<table>
<thead>
<tr>
<th>NOTATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>the cardinality of the table $\mathcal{T}$, denoted by $</td>
</tr>
<tr>
<td>$m$</td>
<td>the cardinality of the attribute $A$, denoted by $</td>
</tr>
<tr>
<td>$k$</td>
<td>the number of bitmaps, $k = \lceil \log_2 m \rceil$</td>
</tr>
<tr>
<td>$d$</td>
<td>the size of $A$’s data type, i.e., $\text{sizeof}(A)$ in language C</td>
</tr>
<tr>
<td>$\pi$</td>
<td>physical page size in bytes</td>
</tr>
<tr>
<td>$\delta$</td>
<td>the cardinality of a range selection</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>the number of variables reduced during the minimization of the retrieval Boolean function</td>
</tr>
<tr>
<td>$c_e$</td>
<td>the cost of processing encoded bitmap indexes, expressed by the number of bitmaps accessed</td>
</tr>
<tr>
<td>$c_s$</td>
<td>the cost of processing simple bitmap indexes, expressed by the number of bitmaps accessed</td>
</tr>
</tbody>
</table>

Table 3.5: Notations used for analysis of bitmap indexing

Obviously, by letting

$$\frac{nk}{8\pi} + \frac{m(d + \frac{k}{\pi} + 1)}{8\pi} < \frac{n \cdot m}{8\pi}$$

$$\implies m > \frac{(n + 8)k}{n - k - 8(d + 1)}, \text{ where } (n - k - 8(d + 1)) > 0$$

the space requirements of simple bitmaps are larger than those of encoded bitmaps. Figure 3.14 shows the space requirements of simple and encoded bitmap indexes. The graphs are plotted with $\pi = 8K$ and $d = 16$ bytes.

The space requirement of a simple bitmap index is a linear function of the cardinality of the indexed attribute, while that of an encoded bitmap index is a logarithmic function of the indexed attribute’s cardinality. As the cardinality grows, the differences between the space requirements of both indexes increase rapidly. Moreover, the sparsity of simple bitmaps is $\frac{m-1}{m}$ on average, while the sparsity of encoded bitmaps is $\frac{1}{2}$, independent of $m$.

Processing costs

The processing cost of encoded bitmap indexes, expressed by the number of bitmaps being accessed during query processing ($c_e$), is

$$c_e = \lceil \log_2 m \rceil - \gamma,$$
where $\gamma$ is the number of variables reduced by the minimization of retrieval function. The processing cost of simple bitmap indexes, expressed by $c_s$, is

$$c_s = \min(\delta, \frac{m}{2}, m - \delta) + 1,$$

where $\delta$ is the cardinality of the range selection. The addition by 1 is due to the necessity of ANDing the result bitmap to the existence bitmap. The $\min()$ function reflects the fact that no more than half of the bitmaps will be read. Of course, this is only true if the bitmap index covers the whole attribute domain. For example, suppose the domain of attribute Gender is \{feminine, masculine, neutral, bisexual, asexual\}, but that only three simple bitmaps are defined on Gender — $b_M$, $b_F$ and $b_N$ representing masculine, feminine and neutral, respectively. The negation of $b_F$ is not equivalent to $b_M + b_N$.

Obviously, $c_e$ and $c_s$ depend on $m$ and $\delta$. Figure 3.15 plots both $c_e$ and $c_s$ by assigning $m = 50$ and varying $\delta$. The curve for the best case is calculated according to Property 2, and the curve for the worst case is drawn by assigning $\gamma = 0$.

The boundary condition where the processing cost of encoded bitmap indexes is lower than
that of simple bitmaps is derived as follows.

\[ |\log_2 m| < \min(\delta, \left\lfloor \frac{m}{2} \right\rfloor, m - \delta) + 1 \quad \Rightarrow \quad \begin{cases} \text{if } \delta \leq \left\lfloor \frac{m}{2} \right\rfloor, & |\log_2 m| < \delta + 1 \\ \text{if } \delta > \left\lfloor \frac{m}{2} \right\rfloor, & |\log_2 m| < m - \delta + 1 \end{cases} \]

\[ \Rightarrow |\log_2 m| - 1 < \delta < m + 1 - |\log_2 m| \quad (3.16) \]

From the above discussion, we see that simple bitmaps degenerate rapidly as the cardinalities of indexed attributes and/or range searches increase. On the other hand, encoded bitmaps solve the problems arising from high cardinalities and retain the advantageous properties of bitmap indexing.

### 3.4 Bit Slices for Continuous Data Types

Another recent related piece of work, but applying a different approach, is the work by Chan and Ioannidis [16, 17]. As discussed in Section 2.3.3, bit slices, specifically binary bit slices, are bitwise vertical projections of the indexed attribute. In [16], the authors proposed a general design scheme for bit slices by augmenting the bases to accepting non-binary or even non-uniform bases and by applying non-equality bit encoding schemes. In this section, the design issues discussed in Chan’s work are introduced for the sake of completeness. Encoded bitmap indexes are well-suited for discrete (alphanumeric) data types, but not limited (c.f. the total-order preserving encoding), and bit slices are well-suited for continuous (numeric) data types. In the next section, both our work and Chan’s work will be included to introduce a global index design scheme.

In Section 2.3.3 (Figure 2.9), an example of a bit-sliced index on lineitems.price with decimal bases is given, with two different bit-encoding schemes — the one with equality-encoding and the other with range-encoding. The choice of the bases obviously affects the space requirements and the performance of bit slices in query processing. For the examples in Figure 2.8 and 2.9, say sizeof(price) = 16, the binary bit slices consist of 16 bitmaps, and the decimal bit slices consist of \(10 \times \mathcal{N}\) bitmaps, where \(\mathcal{N}\) denotes the number of components. In Figure 2.9, \(\mathcal{N} = 2\) is the maximal number of decimal digits among the attribute values. The bit-sliced index in Figure 2.9(c) consists of \(9 \times \mathcal{N}\) bitmaps only, since the last bitmap of each component is a 1-vector and thus can be omitted. In general, for uniform bases, as the magnitude of the base increases, so does the number of total bitmaps in the bit slices.

Using the binary bit-sliced index in Figure 2.8, the selection “price ≤ 25” will be transformed to the following bitmap operation, and 15 bitmaps are read.

\[
\text{price} \leq 25 \quad \Rightarrow \quad \sum m(0, 1, 2, 3, \ldots, 25) \quad \text{(sum of minterms)} \\
\Rightarrow \quad b_{15} b_{14} b_{13} b_{12} b_{11} b_9 b_8 b_7 b_6 b_5 (b_4 + b_3 + b_2 b_1)
\]

On the other hand, using the decimal-based bit slices shown in Figure 2.9(b) to evaluate the above selection, the following bitmap operation is performed.

\[
\text{price} \leq 25 \quad \Rightarrow \quad b_1^2 + b_0^2 + b_2^1 (b_5^1 + b_4 + b_3^1 + b_2^1 + b_1^1 + b_0^0)
\]

The transformation is done as follows. If we express a decimal number \(D\) as \(d_2d_1\), then the value of \(D\) is equal to \(d_2 \times 10^1 + d_1 \times 10^0\), where \(d_2\) is the digit at position 10\(^1\) and \(d_1\) at position
The condition — $D \leq 25$ — is equivalent to $d_2 \leq 1$ OR ($d_2 = 2$ AND $d_1 \leq 5$). The above bitmap operation has the same semantics as $d_2 \leq 1$ OR ($d_2 = 2$ AND $d_1 \leq 5$).

Using the bit slices of Figure 2.9(c), the above selection is transformed to the following bitmap operation:

$$b_1^2 + b_2^2b_0^0$$

The term $b_1^2$ selects data with $d_2 \leq 1$, and $b_2^2b_0^0$ selects data with ($d_2 = 2$ AND $d_1 \leq 5$). The above example shows that the performance of bit slices improves as the magnitude of their bases increases. Note that the number of components is determined by the magnitude of bases. In addition, it is quite straightforward to see that for range searches, range bit-encoding performs better than equality bit-encoding.

Bit slices can also have non-uniform bases. It is worth mentioning that non-binary uniform based bit slices are often less efficient in both space and time among those indexes with the same number of components. In [16], theorems are defined as guidelines for defining space optimal or time optimal bit-sliced indexes with well-defined bases. A well-defined base, say $<b_N, \cdots, b_1>$, consists of finite number of components, i.e., $N \in \mathbb{N}$, such that $b_N = \lceil \frac{|A|}{\prod_{i=1}^{N-1} b_i} \rceil$, where $A$ is the attribute to be indexed. The space and time optimal bit-sliced indexes are defined as follows.

**Space-optimum** Given an integer $N$, the $N$-component bit-sliced index with the base $<b-1, \ldots, b-1, b, \ldots, b>$, where $b = \lfloor \sqrt[N]{|A|} \rfloor$ and $r$ is the smallest positive integer such that $b^r(b-1)^{N-r} \geq |A|$, is the space optimal $N$-component bit-sliced index.

**Time-optimum** Given an integer $N$, the index with the base $<2, \ldots, 2, \lceil \frac{|A|}{2^{N-1}} \rceil>$ is the time-optimal $N$-component bit-sliced index.

Three points need to be stated here. First, the optimal time is derived based on the index processing algorithms proposed in [16] and [65]. Second, the optimal time is only true for a subset of selections, i.e., Type I (continuous range search) queries. For Type II (IN-lists) queries, the optimality of the bit slices needs further examination. Third, the optimal space and time defined above are subject to a given number of components, i.e., they describe a local optimum. For example, a 3-component space-optimal index might perform better than a 5-component time-optimal index in both time and space respects, or vice versa. Globally, for all bit-sliced indexes, those with binary uniform bases are space optimal; in contrast, those with the bases equal to the cardinalities of the indexed attributes, i.e., resulting in simple bitmap indexes, have the worst space utilization. As for global time-optimum with or without space constraints, it requires further analysis of index performance with respect to both query types — Type I and Type II queries, and will be discussed in next section.

From the above discussion, we see that there exists a dilemma between time and space in the design of bit slices. Encoded bitmap indexes provide possible solutions to this dilemma. In principle, an encoded bitmap index is a binary bit-sliced index on the encoded attribute domain. Through its binary base, it minimizes the space requirement, and at the same time, it
provides optimization potential through well-defined encoding to minimize the time complexity at query processing.

Although encoded bitmap indexes can be treated as binary bit slices, they have two advantages over binary bit-sliced indexes. First, the number of bitmaps of an encoded bitmap index is no larger than that of binary bit slices, since the number of encoded bitmaps is decided by the base two logarithm of the indexed attribute’s cardinality, while the number for binary bit slices is decided by the size of the indexed attribute’s data type. Furthermore, for deleted tuples, bit slices need an extra existence bitmap, $b_E$, while defining “deleted” as zero in an encoded bitmap index eliminates the need of an extra existence bitmap. This reduces not only the space requirement but also the processing time.

Second, encoded bitmap indexes have more optimization potential than binary bit slices. Through well-defined encoding, the number of bitmaps accessed during query processing can be reduced, while almost all the bitmaps, including the existence bitmap, in binary bit slices must be read.

However, these are not the most convincing reasons for using encoded bitmap indexes. Instead, the fact that encoded bitmap indexes are more well-suited for discrete (alphanumeric) data types than bit slices, while bit slices are more well-suited for continuous (numeric) data types than encoded bitmaps, is the key point. For an optimized query performance, both encoded bitmaps and bit slices have not only their own playgrounds, but they can also work with each other through bitwise logical operations (a nice feature of bitmap indexing), since queries are likely to involve both types of data.

Some words need to be said here about encoded bitmap indexing before we begin with the topic of bitmap index design. The definition of an encoded bitmap index in Section 3.1, the processing of encoded bitmaps using retrieval Boolean functions, and the discussion derived from well-defined encoding in Section 3.1.2 are all based on the choice of binary bases. It does not imply that an encoded bitmap index must have binary base. However, for a non-binary encoded bitmap index, the definition of the well-defined encoding and all the work derived from it, including the index processing algorithms, must be revised. Since it is not the intention of this work to cover all the cases in the design space (defined in Section 2.3.4) of bitmap indexing, only binary encoded bitmap indexes will be discussed. An example of a non-binary encoded bitmap index can be found in Appendix B.

In a word, along the three dimensions of the design space for bitmap indexing — domain encoding, value decomposition and bit encoding — defined in Section 2.3.4, the following heuristics is suggested as simple, general guidelines of index design: 1) Apply domain encoding to alphanumeric attributes, which are involved mostly in discrete range searches, and 2) use range bit-encoding for numeric attributes, which are involved mostly in continuous range searches. As for the dimension of value decomposition, the discussion in the next section provides a theoretical basis to choose an optimal decomposition scheme for numeric attributes, with or without space constraints.
3.5 Bitmap Index Design: Time Optimal Index

As mentioned above, encoded bitmap indexes are used for alphanumeric attributes, regardless of types of range queries. For numeric attributes that are likely to be involved in both types of range queries, the time optimum of bit slices by varying the bases is discussed. In [16], the algorithm, RangeEval-Opt (listed in Appendix C.1 for reference), for evaluation of Type I (continuous) range queries using bit slices was proposed. In [92], two more algorithms are introduced for evaluation of Type II (discrete) range queries using bit slices, and the time complexities of these two algorithms are analyzed. Based upon the time functions of all the three algorithms, a new design criterion of global time optimal indexes for both types of range queries is derived [92]. Finally, the case where space constraints exist is discussed.

3.5.1 Continuous range selections

Using the algorithm RangeEval-Opt to evaluate Type I (continuous) range selections on \( \mathbb{B}_{<b_N,...,b_n>} \) with range bit-encoding, the time function, expressed by the number of bitmaps accessed, is

\[
time^c \left( \mathbb{B}_{<b_N,...,b_n>} \right) = 2N - \sum_{i=1}^{N} \frac{1}{b_i} + 1 \left( \frac{1}{b_I} - 1 \right)
\]

\[
\approx 2N - \eta
\]

(3.17) (the worst case), (3.18)

where \( \eta = 1 \), if the range operator is one of \( \{<,\leq,>,\geq,=,\neq\} \), and \( \eta = 0 \), if the range operator is one of \( \{=,\neq\} \) [16]. From the above equations, we can see that the time efficiency of a bit-sliced index degrades as the number of components, \( K \), increases. In principle, the fewer components a bit-sliced index has, the more time-efficient the index is.

The above time efficiency analysis is only true if the range operators are confined to the set \( \{<,\leq,>,\geq,=,\neq\} \), i.e., continuous range selections. Anomalies arise if we simply apply the above equation to analyze the time efficiency of bit slices for discrete range selections. Although discrete range predicates of the form — \( A \in V \), or \( A \notin V \) — could be expressed by a disjunction of equality selection predicates, e.g., \( (A = v_0) \ OR \ ... \ OR \ (A = v_m) \), it would be very inefficient to evaluate discrete range predicates in such a brute-force way using RangeEval-Opt. Therefore, it is unreasonable to use the time function based on RangeEval-Opt to analyze the performance of the index in answering discrete range selections.

Consider the following example. Suppose we have a bit-sliced index, \( \mathbb{B}_{<10,10,10>} \), on attribute \( A \) using range \( \leq \) bit-encoding. For the selection predicate “\( A \) IN \{864,764\}”, 12 bitmaps are accessed using RangeEval-Opt, since RangeEval-Opt treats and evaluates each value in the operand set separately, namely \( A = 864 \ OR \ A = 764 \). Six bitmap scans are required for evaluating each of the values. Obviously, discrete range predicates can be evaluated more efficiently by simply avoiding reading the same bitmap more than once. In the above example, the bitmaps for evaluating the two digits at positions \( 1^3 \) and \( 1^0 \), i.e., \( b_0^2, b_0^3, b_1^1, \) and \( b_1^4 \), are read twice using RangeEval-Opt.

In order to define a reasonable time function for evaluating discrete range predicates using bit slices, two algorithms for this purpose are introduced and their performance is analyzed [92].
3.5.2 Discrete range selections

Algorithms 2 and 3 evaluate discrete range predicates using bit slices with range \( (\leq) \) and equality \((=)\) bit-encoding, respectively. The basic idea of these algorithms is to avoid rescanning the same bitmap for consequent equality tests. Every required bitmap is scanned exactly once, and all the comparisons involving the memory-resident bit-segments are performed during the same run.

The algorithms work as follows. Before evaluating the predicate \( A \in \mathbb{V} \), the values in the set \( \mathbb{V} \) are parsed once to examine what bitmaps are required for the evaluation (line 4 to line 7 in Algorithm 2, and line 4 to line 6 in Algorithm 3). Then, all the required bitmaps are read into the buffer, and the algorithms loop for each value in \( \mathbb{V} \) and perform the equality comparisons digit by digit. In practice, the total size of required bitmaps may not fit into memory. Therefore, an implementation of the algorithms may loop for the reading of bitmaps page by page (line 8 to line 16 in Algorithm 2, and line 7 to line 13 in Algorithm 3).

**Algorithm 2** Discrete Selection, range \((\leq)\) bit-encoding

Input: A bit-sliced index with the base, \(< b_\mathcal{N}, \ldots, b_1 >\), where \( \mathcal{N} \) is the number of components and \( b_j^i \) denotes the \( j \)-th bit vector of \( i \)-th component.

Selection predicate \( A \in \mathbb{V} \), where \( \mathbb{V} = \{ v_1, \ldots, v_k \} \), where each value \( v_j \) \((1 \leq j \leq k)\) is represented as \( v_{\mathcal{N},r} \cdots v_{j,1} \) \((0 \leq v_{j,i} < b_i, 1 \leq i \leq \mathcal{N})\).

Output: A bitmap vector representing the set of tuples which satisfy the range-selection predicate, \( A \in \mathbb{V} \).

1) **Begin**
2) \( \text{let } \mathbb{B} = 1 \text{ and } \mathbb{B'} = 0; \)
3) **initialize** \( \mathcal{N} \)  arrays of bits \( M_i[b_i-1]; (1 \leq i \leq \mathcal{N}) / \ast M_i[0] \cdots M_i[b_i-2]/; \)
4) **for** \( i = 1 \text{ to } \mathcal{N} \)
5) \( \text{for } j = 1 \text{ to } k \)
6) \( \text{if } (v_{j,i} < b_i-1) \text{ then } M_i[v_{j,i}] = 1; \)
7) \( \text{if } (v_{j,i} > 0) \text{ then } M_i[v_{j,i}-1] = 1; \)
8) **for** \( i = 1 \text{ to } \mathcal{N} \)
9) \( \text{for } j = 0 \text{ to } b_i - 2 \)
10) \( \text{if } (M_i[j] = 1) \text{ then read } b_j^i; \)
11) **for** \( j = 1 \text{ to } k \)
12) **for** \( i = 1 \text{ to } \mathcal{N} \)
13) \( \text{if } (v_{j,i} = b_i-1) \text{ then } \mathbb{B} = \mathbb{B} \cdot (b_j^{i-1}); \)
14) \( \text{else if } (v_{j,i} = 0) \text{ then } \mathbb{B} = \mathbb{B} \cdot (b_j^{i-1}); \)
15) \( \mathbb{B'} = \mathbb{B'} + \mathbb{B}; \)
16) \( \text{return } \mathbb{B'}; / \ast \text{ handle the non-existent tuples and null-value here} \ast / \)
17) **End**

Now we can analyze the time complexity of the above two algorithms by counting the number of bitmaps read. Generally speaking, to evaluate a digit, \( \alpha \), in component \( i \), the bitmap representing \( \alpha \) in the \( i \)-th component, i.e., \( b_i^\alpha \), is read. (For range bit-encoded bit slices, \( b_i^{b_i-1} \) is also read, if \( 0 < \alpha < b_i - 1 \).) For example, to evaluate the equality-text involving the digit \("6\) of component-2, the bitmaps, \( b_6^2 \) and \( b_6^2 \), are read in Algorithm 2, and \( b_6^2 \) is read in Algorithm 3.

This assumes that the key values are decomposed by the bases \(< b_\mathcal{N}, \ldots, b_1 >\), and the values
Algorithm 3 Discrete Selection, equality (=) bit-encoding

Input: A bit-sliced index with the base, \( b_{N}, \ldots, b_{1} \), where \( N \) is the number of components and \( b_{j}^{i} \) denotes the \( j \)-th bit vector of the \( i \)-th component. Selection predicate \( A \in \mathbb{V} \), where \( \mathbb{V} = \{v_{1}, \ldots, v_{k}\} \), where each value \( v_{j} \) (\( 1 \leq j \leq k \)) is represented as \( v_{j,N} \cdots v_{j,1} \) (\( 0 \leq v_{j,i} < b_{i}, 1 \leq i \leq N \)).

Output: A bitmap vector representing the set of tuples which satisfy the range-selection predicate, \( A \in \mathbb{V} \).

1) \textbf{Begin}
2) \hspace{1em} let \( B = 1 \) and \( B^{r} = \emptyset \);
3) \hspace{1em} initialize \( N \) arrays of bits \( M_{i}[b_{i}]: (1 \leq i \leq N), / */M_{i}[0] \cdots M_{i}[b_{i}-1]/ * /
4) \hspace{1em} for \( i = 1 \) to \( N \)
5) \hspace{2em} for \( j = 1 \) to \( k \)
6) \hspace{3em} \( M_{i}[v_{j,i}] = 1 \);
7) \hspace{1em} for \( i = 1 \) to \( N \)
8) \hspace{2em} for \( j = 0 \) to \( b_{i} - 1 \)
9) \hspace{3em} if \( (M_{i}[j] = 1) \) then read \( b_{j}^{i} \);
10) \hspace{1em} for \( j = 1 \) to \( k \)
11) \hspace{2em} for \( i = 1 \) to \( N \)
12) \hspace{3em} \( B = B \cdot b_{j}^{i} \);
13) \hspace{1em} \( B^{r} = B^{r} + B_{j}^{i} \);
14) \hspace{1em} return \( B^{r} ; / */\text{handle the non-existent tuples and null-value here} */ /
15) \textbf{End}

in each component are evenly distributed between 0 and \( b_{i} (1 \leq i \leq N) \). The time functions of both algorithms are defined as follows.

Using Algorithm 2 to evaluate Type II (discrete) range selections on \( b_{N}, \ldots, b_{1} \) with range bit-encoding, the time function is

\[
time^{d}(b_{N}, \ldots, b_{1}) = \sum_{i=1}^{N} \sum_{j=0}^{k-1} \frac{1}{C_{b_{i}}^{b_{i}+k-1}} \left( \min(2(k-j), b_{i}) C_{j}^{k-1} C_{k-j}^{b_{i}} \right), \tag{3.19}
\]

while using Algorithm 3, the time function is

\[
time^{d}(b_{N}, \ldots, b_{1}) = \sum_{i=1}^{N} \sum_{j=0}^{k-1} \frac{1}{C_{b_{i}}^{b_{i}+k-1}} \left( (k-j) C_{j}^{k-1} C_{k-j}^{b_{i}} \right), \tag{3.20}
\]

where \( C_{n}^{m} = \frac{m!}{n!(m-n)!} \) denotes the number of combinations of choosing \( n \) from \( m \). (Details about the derivation of the time functions can be found in Appendix A.6.)

In the worst case, or if \( k \) is large enough (i.e., the cardinality of the range selections), both Algorithm 2 and 3 read all the bitmaps of the index. In such cases, the time functions for discrete range evaluation using bit slices are defined by their space requirements.

\[
time^{d}(b_{N}, \ldots, b_{1}) = \text{space}(b_{N}, \ldots, b_{1}) = \begin{cases} \frac{N}{\sum_{i=1}^{N} (b_{i} - \eta)}, & \text{if } b_{i} = 2, 1 \leq i \leq N \\ \sum_{i=1}^{N} (b_{i} - \eta), & \text{if } b_{i} > 2, 1 \leq i \leq N \end{cases} \tag{3.21}
\]

where \( \eta = 0 \) for equality bit-encoding, and \( \eta = 1 \) for range bit-encoding. As a result, the \( N \)-component time optimal indexes for discrete selections are those \( N \)-component space optimal
indexes. This result conflicts with the theorems derived for continuous range selection in [16], where the time-optimum and the space-optimum result in different index designs.

### 3.5.3 Global time optimum

The above discussion shows that conflicting index design criteria exist for Type I and Type II queries. Since most attributes might be involved in both types of selections, choosing either of the design criteria presents a dilemma. One straightforward solution is to design an index for each type of selection, i.e., an index with the fewest possible components for continuous range selections, and another index with binary bases for discrete range selections.

Another approach is to find a global time optimal index for both types of selections. In Figure 3.16, time optimal indexes for either continuous selections or discrete selections on an attribute $A$ with $|A| = 1024$ are illustrated. A point on a curve represents an index (labeled with the number of components), the $x$-distance denotes the time required to evaluate a discrete selection predicate using the index, i.e., $time^d()$, and the $y$-distance denotes the time required to evaluate a continuous selection predicate using the index, i.e., $time^c()$. The computation of the time functions is based on Equations (3.17) and (3.21).

![Time-optimal Indexes](image)

Figure 3.16: Time optimal indexes, $|A|=1024$

As Figure 3.16 shows, choosing an index which reduces the time required to evaluate continuous selections will increase the time required to evaluate discrete ones, and vice versa. In order to find the global time optimal index, the following time function is defined

$$
\rho \cdot time^c(\mathcal{B}) + (1 - \rho) \cdot time^d(\mathcal{B}),
$$

(3.22)

where $\mathcal{B} \in \mathcal{I}$, $\mathcal{I}$ is the universal set of bit-sliced indexes, and $\rho$ is the probability of occurrence of continuous range selections given on the indexed attribute of $\mathcal{B}$. The minimum of the function, which is defined as the break-even point, characterizes the global time optimal index for both types of selections.

Note that in our example, for the sake of clarity, $\mathcal{I}$ is confined to the set of $I^C \cup I^D$, where $I^C$ ($I^D$) denotes the time optimal indexes for continuous (discrete) selections. As a matter of fact, this restriction provides an efficient way to find an approximate optimal solution, since $I^C$ and $I^D$
describe boundaries of the bit-sliced index space. One describes the optimum for continuous selection evaluation, and the other describes the optimum for discrete selection evaluation.

The two functions \( \rho \cdot \text{time}^c(I^C) + (1-\rho) \cdot \text{time}^d(I^D) \) and \( \rho \cdot \text{time}^c(I^D) + (1-\rho) \cdot \text{time}^d(I^D) \) — are plotted in Figure 3.17(a), and the minimum point of the curves, \((10,10)\), is the break-even point. This means a 10-component bit-sliced index on attribute \( A \) ([\( A \)]= 1024) is optimized for both types of selections. (At the point \( n = 10 \), the index is a binary uniform bit-sliced index with ten components.)

![Time-optimal Indexes](a) \( \rho = 0.5 \)

![Time-optimal Indexes](b) \( \rho = 0.75 \)

*Figure 3.17: Global time optimal indexes*

Figure 3.17(b) reveals the curves with \( \rho = 0.75 \), i.e., 75% of the selections are continuous. The point \((5,9.75)\) is the break-even point, which means the 5-component time optimal index for discrete selections is also the global time optimal index.

### 3.5.4 Global time optimal indexes under space constraint

Under a space constraint \( M \), the global time optimal index is defined by

\[
\min \left( \rho \cdot \text{time}^c(\mathbb{B}) + (1-\rho) \cdot \text{time}^d(\mathbb{B}) \right), \ \mathbb{B} \in \mathcal{T}',
\]

where \( \mathcal{T}' \) is the set of bit slices whose space requirements are no larger than \( M \), i.e., \( \mathcal{T}' = \{ \mathbb{B} | \text{space}(\mathbb{B}) \leq M \} \).

To avoid the exhaustive search in the whole index space, the search space is also confined to the boundaries, i.e., to find the smallest \( \mathcal{N} \) and \( \mathcal{N}' \), such that \( \text{space}(I^C_{\mathcal{N}}) \leq M \) and \( \text{space}(I^D_{\mathcal{N}'}) \leq M \), and let \( \mathcal{T}' = \{ I^C_{\mathcal{N}} | i \geq \mathcal{N} \} \cup \{ I^D_{\mathcal{N}'} | j \geq \mathcal{N}' \} \). \( I^C_{\mathcal{N}} \) and \( I^D_{\mathcal{N}'} \) denote the \( \mathcal{N} \)-component time optimal index for the continuous selection type and the \( \mathcal{N}' \)-component time optimal index for the discrete selection type, respectively. Then, the approximate global time optimal indexes under the space constraint \( M \) is given by

\[
\min \left( \rho \cdot \text{time}^c(\mathbb{B}) + (1-\rho) \cdot \text{time}^d(\mathbb{B}) \right), \ \mathbb{B} \in \mathcal{T}'
\]

Following the example in Figure 3.17(b) ([\( A \)= 1024]), suppose that \( M = 50 \) bitmaps, i.e., maximal 50 bitmaps can be stored in the system, then \( \mathcal{N} = 3 \) and \( \mathcal{N}' = 6 \), since for all \( i \geq \mathcal{N} \) and \( j \geq \mathcal{N}' \).
3.6 Related Work

In the literature, much work has been done on indexing schemes and their use in database systems. In this section, we review some work which is relevant to Data Warehousing.

**Dynamic bitmaps** Dynamic bitmaps are built dynamically on high cardinality attributes [74]. If there are $n$ different values in the attribute domain, they are encoded onto $n \cdot (\log_2 n)$-bit continuous binary numbers. Dynamic bitmaps are special cases of encoded bitmap indexes, where the encoding function trivially maps the domain onto a continuous integer set. The significance of encoding was not discussed in dynamic bitmaps.

**Dynamic range-based bitmaps** A dynamic range-based bitmap indexing for high cardinality attributes with data skew was proposed in [93]. The idea is to partition the domain into some equally populated subsets, and simple bitmap vectors are constructed, one for each subset.

Compared with range-based encoded bitmap indexes introduced in Section 3.2.3, the two approaches differ from each other in the following aspects: (1) In [93], partition was done by distribution of the attribute values, while range-based encoding partitions attribute domains according to pre-defined range selections. (2) In [93], Wu and Yu investigated how to dynamically adjust the partition of the ranges to balance the population of all buckets with respect to the distribution of attribute values. However, range-based encoded bitmap indexes do not have the problem of imbalance. Because the predefined selection predicates are used to partition the attribute domains, the retrieval functions will, therefore, exactly match the desired tuples. In the cases that selection predicates are not pre-definable, or the predicates result in a very large number of small partitions, a range-based encoded bitmap index resembles an encoded bitmap index on a set of single values with a total-order preserving encoding.

**Compressed bitmaps** Encoded bitmap indexes and bit-sliced indexes exploit domain encoding and value decomposition to reduce the space requirements of simple bitmap indexes at logical design. Another alternative that reduces the space requirements at physical design is
to compress bitmaps [2, 54, 61]. The use of bitmap compression has advantages with respect
to storage utilization, but compressed bitmaps have some problems. Most importantly, if the
compressed bitmaps must be decompressed before performing operations on them, the decom-
pression overheads may diminish the savings in space and time. Algorithms for performing
Boolean operations on compressed bitmaps and their performance are reported in [2, 54, 61].
Due to the data-dependent nature of compression techniques, the performance of bitmap
indexes and their algorithms may be influenced by the compression techniques in a way that
is not subject to index design. Therefore, the discussion in this work is concentrated on un-
compressed bitmaps. Query processing algorithms are proposed for uncompressed bitmaps,
and their performance is also measured without the effect of compression. Nevertheless, to
integrate bitmap indexing into real world systems, compression is a technique that should be
taken into account [67, 66, 83].

**Transposed files and bit-transposed files** Conventional files store data as a collection of
records (tuples). Transposed files store data as a collection of attributes, i.e., they can be viewed
as vertical partitions of conventional files [88, 82, 3]. Bit-transposed files are bitwise vertical par-
titions of the conventional files [89]. Bit-transposed files are actually bitmaps in a bitmap index.
The bit-encoding schemes and value decomposition schemes discussed in [16, 17] originated
from [89]. Compared with our work, Wong did not address the issue of domain encoding
in [89], and only selection evaluation using bit-transposed files was discussed, while in this
work we discuss query processing and optimization using bitmap indexes globally, not just for
single query operation.

**Projection Index and DataIndex** DataIndexes are variants of projection indexes [25]. A projec-
tion index is a projection on a single attribute. Basic DataIndexes generalize projection indexes
by allowing any number of attributes to be in the indexes. Join DataIndexes are also projection
indexes on the foreign keys, but, instead of storing the key values, tuple ids of the referenced
tuples of the dimension tables are saved. A partitioning scheme for reducing memory require-
ments at run time and compression algorithms to reduce space requirements of DataIndexes
are proposed in [37, 26].

**Multidimensional indexes** Multidimensional indexes have also sparked recent discussion
in Data Warehousing [14, 31, 34, 73, 6, 7, 70, 9, 53]. One of them is called the Universal B-
trees (UB-trees) [6, 7, 59, 70, 60]. The UB-tree requires linear space of the number of stored
objects for storage and logarithmic time for the operations — insertion, retrieval and deletion.
The strength of UB-trees is the use of them as clustered indexes and the underlying data is
physically organized based on a multidimensional ordering, named Z-addresses. Using UB-
trees as primary, clustered indexes eliminates the needs of building multiple secondary B-trees,
and the problems of index selection disappear [8]. However, bitmap indexes introduced in this
work are used as secondary indexes. They are not directly comparable.

**SUMMARY** In this chapter, encoded bitmap indexing is introduced, which on one hand solves
the performance degeneration of simple bitmap indexing due to poor space utilization at high
cardinalities, and on the other hand provides better query performance than simple bitmap
indexing and B-trees can. The maintenance of encoded bitmaps and issues about the encoding
are also discussed, including the definition of the well-defined encoding and other theorems
derived from it. Three more examples (other than the variations introduced in Section 2.3.5) by
applying application-dependent encodings show the usefulness of encoded bitmap indexing
in Data Warehousing environments.

Encoded bitmap indexes are well-suited for, but not restricted to, attributes which are involved in discrete range selections. For continuous range selections, another variation of bitmap indexing, bit-sliced indexing [65, 16, 17], is a better choice. For attributes which are likely to be involved in both types of selections, the global time optimum of an index by considering probabilities of both selection types in users’ queries is defined. The definition of the global time optimum is used as a guideline in designing an optimal index. Finally, how to revise the global time optimum with presence of space constraints is also discussed.

As discussed in Chapter 2, if indexes are used for selection evaluation only and different indexes are applied exclusively, the application of indexes will be very restricted, even for selections. Although in this chapter the performance of the selection operation on bitmap indexes is analyzed (since searching and retrieving are basic operations for processing an index), it does not imply that bitmap indexes are used for selection evaluation only. In the next chapter, how to exploit bitmaps to process and optimize queries, and how bitmaps change the mechanism of query processing and the philosophy of query optimization will be introduced.
Chapter 4

Query Processing and Optimization using Bitmaps

In OLAP systems, short response times are as important as short query processing times. Especially, users may refine their queries based upon the responses of prior rough queries. Subject to different performance goals, techniques applied for query processing and optimization may differ. In this chapter, query processing techniques using bitmaps that optimize not only the processing of individual query operators but also the global query processing will be discussed. The techniques introduced here are aimed at minimizing the response time and at the same time to keep the query processing time low.

4.1 Performance Goal and Methodologies

The performance measurement is application dependent, and the techniques applied to achieve the performance requirements are usually subject to data volume and to logical and physical system architectures. The logical system architecture includes issues such as logical database design, index design and algorithm design, and the physical system architecture includes issues such as physical data layout, disk layout, and hardware architecture.

In this work, not all the topics listed above are covered. Instead, we concentrate on some of the logical issues that are required to build a high performance Data Warehouse system. First of all, the measurements of performance should be defined. Unlike OLTP systems, where performance is often measured by transaction throughput, the performance of Data Warehouse systems is measured by both the response time and the processing time. The response time is defined by the time interval from the point in time of the query submission until the time when the first result is returned to the user. The processing time is defined by the time interval from the query submission to the end of the query execution. A short response time is important, since the end user may take further actions, e.g., refining her/his query, making a decision, or even aborting the execution of the query, based upon the partial results returned by the system. The processing time is critical, since an important objective of a Data Warehouse system is to support decision making that is time critical.

Once the performance measurements are defined, several approaches may be taken to achieve the performance goals. In this work, we concentrate on applying indexing techniques. In the
last chapter, a new indexing technique, *encoded bitmap indexing*, is introduced. In this chapter, let us see how to exploit it, together with other variants of bitmap indexing, to optimize query processing. Unlike traditional indexes, bitmap indexes are used in a wide range of query operations, including *selections*, *Group Bys*, *aggregations* and *Order Bys*. Due to the characteristics of bitmaps, several optimization techniques can be applied at run time to further improve the query performance. For example, statistical and distribution information about the underlying data can be easily and efficiently derived from bitmaps for dynamic query optimization, since that information is encapsulated in a precise and economic data structure — *bits*. For another example, bitmaps encapsulate the physical addresses of the desired data in bit positions. This information will never be lost, no matter how many times the bitmaps are manipulated. This property makes bitmaps designed for different purposes work with one another very well, and the results of bitmap manipulations are still bitmaps. Making a good usage of this property can postpone data fetching to the very last phase of query processing, such that only data of interest are retrieved.

4.1.1 Overview of query processing using bitmap indexes

The proliferation of bitmap indexes enriches the landscape of query processing and optimization in Data Warehouse systems. Later in this chapter, how bitmap indexes can be used to process and optimize individual query operations is discussed. These algorithms serve as building blocks for a *bitmap-enabled* query processor/optimizer. Before doing that, let us first take a global view of the flow of query execution and point out when bitmap indexes can be used.

![System components of a query processor](image)

(a) A usual flow of query processing  
(b) A bitmap-enabled query processor

Figure 4.1: System components of a query processor

Figure 4.1(a) depicts the usual flow of query execution. It is divided into three major steps — *query parsing*, *optimization* and *execution* [35]. The first two steps are also known as *query compilation*. Traditionally, users’ queries expressed in SQL are parsed into expression trees¹ of

¹Conventionally, the query parser takes queries written in SQL and converts them into parse trees that are further
the relational algebra. Algebraic laws are applied to transform the logical query plans, and the resulting expressions are turned into a physical execution plan by choosing the best-suited implementations of the physical query operators\(^2\) with the lowest costs.

Figure 4.1(b) shows the system diagram of a bitmap-enabled query processor. It shows that the system architecture and the flow of query execution do not change. What really changes is the internal architecture of each system component. The changed components are shadowed in Figure 4.1(b). They are discussed in bottom-up order.

First, the introduction of bitmap indexes to query processing has direct impact on the implementation of individual physical query operators, such as selection and aggregation. In addition, the logic of the interactions between multiple query operations changes accordingly. For a quick example (details about the design of bitmap algorithms for different query operators are discussed in the following sections), a conventional query processor may evaluate a SELECT-FROM-WHERE query as follows. Among many selection conditions, the index (if available) on the most selective attribute is applied to evaluate one of the selections. The results of the index processing will be used to retrieve data, and the rest of the selections are directly evaluated on the selected data, which are passed on to Group Bys, Joins or other operations. Since no access path exists for the intermediate results, the methods chosen for subsequent operations are restricted to those which do not require access paths. On the other hand, a bitmap-enabled query processor may use bitmap indexes to evaluate all of the indexed selection conditions. The result of index processing — which is still a bitmap — is not used immediately to retrieve data. Instead, subsequent\(^3\) query operations are also evaluated with help of bitmap indexes (again, if available). Finally, the resulting bitmaps of different query operations are combined (by bitwise logical operators) to fetch data, followed by some, if any, post-processing. The control flows of both cases are depicted in Figure 4.2.

Second, the availability of bitmaps enriches the potential for query optimization. Query opti-
\(^2\)One logical query operator, i.e., one operator in the relational algebra, is usually implemented by one or more physical operators. In addition, there are some helper functions, like table-scan, which do not map to any relational operator.

\(^3\)As a matter of fact, in the bitmap world, selection needs not be evaluated before other query operations.
mization consists of two sub-steps. The first one transforms users’ queries expressed in high-level languages into the relational algebra, and finds optimal algebraic expressions equivalent to the original queries. This is called *logical optimization*. The second one chooses an execution plan by mapping the relational operators to physical implementations and is called *physical optimization*.

The introduction of bitmap algorithms provides more choices at the phase of physical optimization. In addition, the choice of bitmap indexes retroactively simplifies the logical optimization phase by making some transformation rules redundant. For example, one of the most commonly applied rules for conventional query optimization is “*pushing the most selective selections down the expression tree as far as possible*”. This is no longer necessary, since multiple index scans can be efficiently realized by bitmaps. In addition, query evaluation can be proceeded without immediate realization of selection operations on base data. Subsequent operators can be evaluated on bitmaps, followed by a logical *AND* to the selection bitmap to retrieve nothing but all the qualified data.

Another example is the rule — “*breaking down and moving the projection down the tree as far as possible by creating new projection lists as needed*”. For the subtree in an expression tree where query operations are evaluated on bitmaps, this rule is no longer necessary. This is because no data is fetched until the end of bitmap manipulation, and thus there is no need to apply projection to narrow down the size of intermediate results, since there is no intermediate result.

A favorable effect of making rules unnecessary is to reduce the size of the search space of transformation, which in turn reduces the cost of optimization. The transformation rules of the relational algebra that is extended by the introduction of bitmap indexing are listed in Appendix D. The introduction of a new indexing technique should not have effects on the algebra. However, by applying a new indexing technique, some originally unacceptable costly transformations may become very time-efficient. On the other hand, some transformations, which resulted in very different performance before, may make no difference in performance now.

In addition, the introduction of bitmap indexes provides better chances for rewriting queries, especially, in Data Warehouses where joins are given on key/foreign-key relationships, named *Star-Joins*. Since Joins are expensive, they are the very best candidates to be rewritten, or even to be eliminated. Query rewrites involve many issues that will be discussed in a separate subsection.

### 4.1.2 Query rewrites

Although “*query rewrites*” is not a new topic, bitmap indexing makes it possible to rewrite users’ queries such that either 1) expensive query operators are replaced with low-overhead operators, or 2) in spite of using the same operators, the revised queries can be optimized later at the generation phase of physical execution plan. The following example shows both cases. Suppose we have the following query.
SQL 6

```sql
SELECT c.mktsegment, SUM(l.qty)
FROM lineitems l, orders o, customers c
WHERE l.order_id = o.order_id AND
      o.cust_id = c.cust_id AND
      c.nation_id BETWEEN '01' AND '03' AND
      o.priority = 'HIGH'
GROUP BY c.mktsegment
```

Suppose that a group-set (simple) bitmap index is defined on the fact table `lineitems`, as Figure 4.3(b) shows (the same example as in Section 2.3.5). In addition, another simple bitmap index is defined on `lineitems.order_id`. The bitmaps on `lineitems.order_id` are not illustrated due to space limitations. A snapshot of the database at the time of query processing is also illustrated in Figure 4.3.

(a) Snapshot of the dimension table `customers`

| cust_id | mktsegment | nation_id | ...
|---------|------------|-----------|---
| C00001  | BUILDING   | 01        |   |
| C00002  | AUTOMOBILE | 02        |   |
| C00003  | AUTOMOBILE | 03        |   |
| C00005  | HOUSEHOLD  | 02        |   |
| C00007  | AUTOMOBILE | 03        |   |
| C00009  | FURNITURE  | 02        |   |
| C00013  | BUILDING   | 03        |   |
| C00014  | FURNITURE  | 02        |   |
| ...     | ...        | ...       | ...

(b) Snapshots of the dimension table `orders` and the fact table `lineitems`. \( b_A \) is the group-set bitmap for the group `AUTOMOBILE`, \( b_B \) for `BUILDING`, \( b_F \) for `FURNITURE`, \( b_H \) for `HOUSEHOLD`

Figure 4.3: Group-set bitmap index

For the sake of simplicity and clarity, in Figure 4.3 only data of interest are printed. Besides, a
simple bitmap index is used in this example, since it is to demonstrate the idea of how to rewrite a query, not the differences between different bitmap indexes. Nonetheless, the correctness and the applicability of the idea is not restricted to simple bitmap indexing.

Conventionally, SQL 6 will be transformed into the following expression in the relational algebra.

\[ \Sigma_{\text{mktsegment}}, \text{sum(qty)} \left( \text{lineitems} \bowtie_{\text{order_id}} \text{o} \bowtie_{\text{cust_id}} \text{cust} \right), \quad \text{where} \]

\[ \text{cust} = \sigma_{\text{nation_id} \text{between} '01' \text{and} '03'} \text{customers}, \]

\[ \text{o} = \sigma_{\text{priority} = 'HIGH'} \text{orders} \]

and \( \Sigma \) denotes the aggregation and the Group By operator. The subscripts of \( \Sigma \) denote the list of grouping attribute(s) and the aggregate function(s) to be calculated, e.g., \( \Sigma_{A, \text{sum(B)}} R \) groups the tuples of \( R \) by values of \( A \), and for all tuples within the same group (i.e., with the same value of \( A \)), the values of \( B \) are summed up. Note that \( \Sigma \) is not a standard relational operator; it is introduced into the relational algebra to ease the discussion. (A complete list of operators and notation used in this work can be found in Appendix F.1.)

Before rewriting the query, let us first introduce the philosophy, the principles and the prerequisite of rewriting a query in the Data Warehousing environments. First, the rewrite philosophy, as is stated in [68], is:

"Whenever possible, a query should be converted to a single SELECT operator".

In Data Warehouses, core data are usually characterized into fact tables, whose descriptive properties are modeled and classified into dimension tables. Fact tables are associated with dimension tables through key/foreign-key references. As stated before, the main purpose of a Data Warehouse is to support OLAP applications. A typical OLAP query may access a large fraction of a fact table (since it stores the core business data) by conditions given on dimensions (since the business knowledge and dimensions are modeled in dimension tables), and/or group the final results by some dimensions. As a result, a typical OLAP query may contain multiple Joins from the central fact table to its surrounding dimension tables, named Star-Joins. In database systems, it is well known that Joins are expensive. Therefore, they are the very best candidates during query revision to be eliminated or substituted. To achieve this goal, the following principles serve as guidelines for rewriting.

1. Selections on dimension attributes should be rewritten as selections on dimension keys if no group-set bitmap index is defined on the selecting attributes.\(^4\)

2. Grouping by dimension attributes should be rewritten as grouping by dimension keys if no group-set bitmap index is defined on the grouping attributes.

3. If selection conditions are also given on the grouping attributes, selection conditions are used first to refine the number of groups before rewriting the grouping condition.

4. Other selections given on the same dimension table, which have been rewritten as selections on the dimension key, are applied to refine the number of groups while rewriting the grouping condition.

\(^4\)Group-set bitmap indexes can be used to evaluate selections as well. Please c.f. Section 2.3.5, where we have discussed variants of bitmap indexes. Although they are designated for, and also named after, other purposes, they can all be used to evaluate selections.
5. Grouping by multiple dimensions is rewritten independently.

The idea behind these principles is to transform selections (or, groupings) on dimension attributes into selections on dimension keys, which are also foreign keys in the fact table. In other words, by transforming selections (or, groupings) on dimensions to selections on fact tables, the joins to the dimension tables can be omitted.\(^5\) If dimensions are normalized, i.e., resulting in a Snow-flake Schema, the transformation is applied recursively until the keys that are also the foreign keys of the fact tables are reached.

In order to express the above transformation, we have to extend the semantics of the Group By operator. The standard Group By operator divides the tuples of the operand table into groups by single values, i.e., the tuples within the same group share the same value of the grouping attribute. For each group, the aggregate function, sum\((B)\) in our example, is carried out. This is expressed in the relational algebra by

\[
\sum_{A, \text{sum}(B)} R = \bigcup_{\forall v \in A} \pi_{A, \text{sum}(B)} (\sigma_{A=v} R)
\]

Note that the subscript of the \(\Sigma\) operator denotes the list of projected attributes. In addition, the attribute name \(A\) denotes the domain of the attribute when it appears on the right hand side of a binary operator; otherwise it denotes the variable.

In this work, the semantics of the Group By operator is extended to include “grouping by multi-values”. It is expressed by

\[
\sum_{P(A), \text{sum}(B)} R = \bigcup_{\forall p_j \in P(A)} \pi_{P(A, \text{sum}(B)} (\sigma_{\text{A} \in p_j} R),
\]

where \(P(A)\) denotes a partitioning on the domain of \(A\). \(P(A) = \{p_1, \ldots, p_n\}\), where \(p_j \subseteq A\) \((1 \leq j \leq n)\), \(\bigcup_{j=1}^{n} p_j = A\) and \(p_i \cap p_j = \emptyset\) \((1 \leq i, j \leq n)\). In fact, it is not necessary for \(P(A)\) to cover the whole domain of \(A\). In terms of the Entity-Relationship model, the total specification, i.e., \(\forall v \in A, \exists p_j (1 \leq j \leq n)\) or \(\exists v \in p_j\), is not required. Furthermore, the disjointness constraint, i.e., \(\forall 1 \leq i, j \leq n, p_i \cap p_j = \emptyset\), is not a necessary condition for the correctness of the query rewrite, either. In the Data Warehousing environment most, if not all, cases of query-rewrites satisfy both constraints.\(^6\) The referential integrity ensures the total specification, and the functional dependency between keys and non-keys guarantees the disjointness constraint. The referential integrity and functional dependency are strict conditions for the correctness of the query rewriting techniques proposed in this work. That is, if they are satisfied, the result sets of revised queries will be equivalent to those of original queries. However, if they are violated, it does not imply the dissatisfaction of the correctness. In Appendix A.5, the proof to the correctness of query rewrites is given.

Although the evaluation of projections is easy, we still need to discuss some things about projections for the sake of completeness. First, projecting an aggregation means a projection on the attribute to be aggregated, followed by the aggregate function on the projected data. Second,

\(^5\)Unlike existing query-rewrite methods in the literature, the rewriting rules introduced here rewrite queries at both the data and syntactic level, instead of only at the syntactic level. That is, the query-rewrite rules introduced here substitute not only operators at the logical level but also operands at the physical level.

\(^6\)The multi-value Group By operator is implemented in the experimental version of \(*+\)HSQL — a bitmap-enabled mini-SQL Engine used for performance evaluation in Chapter 5. Currently, the implementation relies on the disjointness constraints.
in cases of multi-value grouping, projecting a multi-value set means expanding the value of the partitioning attribute over the result set. (An example follows.)

With help of the multi-value Group By operator, we can rewrite the above query as follows.

\[
\text{cust} = \sigma_{\text{nation.jd} \text{ bbn}'01' \text{ and } '03'} \text{ customers}
\]

\[\sigma_{\text{priority}='HIGHER'} \text{ orders}\]  

\[
P(\text{cust}.id) = \{p_A, p_B, p_F, p_H\}, \quad \text{where } p_A = \pi_{\text{cust}.id}(\sigma_{\text{mktsegment}='AUTOMOBILE'} \text{ cust})
\]

\[
p_B = \pi_{\text{cust}.id}(\sigma_{\text{mktsegment}='BUILDING'} \text{ cust})
\]

\[
p_F = \pi_{\text{cust}.id}(\sigma_{\text{mktsegment}='FURNITURE'} \text{ cust})
\]

\[
p_H = \pi_{\text{cust}.id}(\sigma_{\text{mktsegment}='HOUSEHOLD'} \text{ cust})
\]

\[
P'(\text{order}.id) = \{p'_A, p'_B, p'_F, p'_H\}, \quad \text{where } p'_A = \pi_{\text{order}.id}(\sigma_{\text{cust}.id \in p_A \text{ o'}})
\]

\[
p'_B = \pi_{\text{order}.id}(\sigma_{\text{cust}.id \in p_B \text{ o'}})
\]

\[
p'_F = \pi_{\text{order}.id}(\sigma_{\text{cust}.id \in p_F \text{ o'}})
\]

\[
p'_H = \pi_{\text{order}.id}(\sigma_{\text{cust}.id \in p_H \text{ o'}})
\]

\[
\Sigma_{P'(\text{order}.id), \text{ sum(qty) lineitems} = \bigcup_{\forall p'_k \in P'(\text{order}.id)} \pi_{P'(\text{order}.id), \text{ sum(qty)}}(\sigma_{\text{order}.id \in p'_k \text{ lineitems})}}
\]

The above revised query works conceptually as follows. (It is said “conceptually” because there is no assumption of what kind of access paths are used.) First, the set of tuples in both \text{customers} and \text{orders} is confined to those which satisfy the selection conditions, and denote them by \text{cust} and \text{o’}. Second, in order to be able to omit the join to the customer-dimension, the grouping by \text{customers.mktsegment} is transformed to the grouping by \text{lineitems.order.id}. To do that, the second rewrite rule mentioned above is applied recursively. It results in the partitioning \( P'(\text{order}.id) \). That is, for each \( p'_k \in P'(\text{order}.id) \), \( p'_k = \{t.\text{order}.id | \forall t \in \text{o’} \text{ and } s \in \text{cust, } \exists t.\text{cust}.id = s.\text{cust}.id \text{ and } s.\text{mktsegment} = k\} \), where \( k \) equals \text{AUTOMOBILE}, \text{BUILDING}, \text{FURNITURE} and \text{HOUSEHOLD}, respectively. Now, we use \( P'(\text{order}.id) \) to select tuples from \text{lineitems} that belong to the same market-segment, followed by a projection on \( P'(\text{order}.id) \) and \text{sum(qty)}. As described above, a projection on a multi-value set means expanding the value of the partitioning attribute over the result set. For example, for \( p'_k \) where \( k = \text{AUTOMOBILE} \), denoted by \( p'_A \) for short, the projection on \( P'(\text{order}.id) \) in the following expression

\[
\pi_{P'(\text{order}.id), \text{ sum(qty)}}(\sigma_{\text{order}.id \in p'_A \text{ lineitems})}
\]

inserts \text{AUTOMOBILE} in the first column of the result set.

The joins in SQL 6 are rewritten to a sequence of selections on the fact table, followed by a union of the aggregation results. Whether or not the revised query is more time-efficient than the original one depends strongly on the selectivities of selections, the size of operand tables and what physical operators are chosen to evaluate the logical operations. It is assumed that the sizes of operand tables and intermediate results of selections are too large to fit into the main memory and discuss next how bitmap indexes can help to efficiently perform the above revised query. In our example, since we have a group-set bitmap index defined on \text{lineitems.order.id}, expressions (4.1) and (4.2) are transformed to selections on \text{order.id} (by the first rewriting rule discussed above).
Evaluating the selection

\[ \sigma_{\text{nation}_j \in \text{btm '01' and '03'}} \] customers,

on the snapshots of the database shown in Figure 4.3 is equivalent to the selection

\[ \sigma_{\text{cust}_j \in C}\text{customers.} \]

where \( C \) equals the set \{’C00001’, ’C00002’, ’C00003’, ’C00005’, ’C00007’, ’C00009’, ’C00013’, ’C00014’\}. Similarly, we can rewrite the selections

\[ \sigma_{\text{cust}_j \in C}(\sigma_{\text{priority} = '\text{HIGH}'} \text{orders}) \]

with

\[ \sigma_{\text{order}_j \in O}(\text{orders}) \]

where \( O \) equals \{’000032’, ’000102’, ’000161’, ’000196’, ’000199’, ’000226’, ’000263’, ’000295’, ’000320’, ’000354’, ’000357’, ’000358’, ’000386’, ’000389’, ’000391’, ’000513’, ’000516’, ’000518’, ’000545’, ’000546’, ’000579’, ’000580’\}. The set \( O \) denotes the set of order\(_j\)'s whose members have 'HIGH' priority and are submitted by customers from nations numbered '01', '02' or '03'. Now it is ready to transform the above selection to bitmap operation. Since \( O \) denotes the set of orders of concern, we can construct a selection bitmap by \text{ORing} all the bitmaps representing the values in \( O \), denoted by \( f_O \). \(^7\)

Expression (4.5) can now be mapped to the following bitmap operation,

\[
\bigcup_{m \in \text{mktsegment}} \pi_m, \text{sum(qty)}(\sigma_{f_O \text{m}} \text{lineitems}),
\]

where \( f_m \) denotes the bitmap for the attribute value \( m, m \in \{\text{AUTOMOBILE, BUILDING, FURNITURE, HOUSEHOLD}\} \). Note that Expression (4.6) is a query plan at the physical level, \( i.e., \) it reveals the information about physical access paths being used. For example, the results of the bitmap operations at the subscript of the selection operator determine the set of tuples to be retrieved. The bit positions of those 1 bits in the resulting bitmap indicate the offsets of the desired tuples in the table space. SQL 6 is now transformed to a sequence of bitmap operations, followed by an aggregate function. The results of SQL 6 are as follows.

<table>
<thead>
<tr>
<th>mktsegment</th>
<th>sum(qty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMOBILE</td>
<td>2248</td>
</tr>
<tr>
<td>BUILDING</td>
<td>2331</td>
</tr>
<tr>
<td>FURNITURE</td>
<td>1643</td>
</tr>
<tr>
<td>HOUSEHOLD</td>
<td>33</td>
</tr>
</tbody>
</table>

From the above discussion about query-rewrite in Data Warehousing, the following guidelines are recognized for bitmap index design:

\(^7\)In cases of encoded bitmap indexes, the retrieval Boolean functions for all the values in \( O \) are \text{ORed} to obtain the final retrieval function, too.
- Define bitmap indexes on the foreign keys (to dimension tables) of the fact tables.
- Define group-set bitmap indexes on the potential grouping attributes.
- Define bit-sliced indexes on the additive attributes which will potentially be aggregated.³

Although not quite relevant here, it is worth mentioning that there is still a lot of optimization work to be done while implementing the above rewriting rules and access functions. For example, the I/Os of reading the operand table can be minimized by avoiding reading the same page more than once, or through an optimal accessing sequence of the pages [77].

4.1.3 Pipelining

Pipelining operations, instead of materializing the intermediate results, reduces the number of I/Os and increases intra-query parallelism. Pipelining successive operations and feeding sorted results promise short response time. To be able to feed sorted results implies either that the input data should also be read in the right order, or that a post-retrieval sorting of results is performed. For the first alternative, the input table is either sorted or hashed. However, in Data Warehouses sorting or hashing the huge fact table is very time consuming and thus results in long response time. Intuitively, it would even result in longer response time than the second alternative, since the results of OLAP queries, being summarized through aggregations and Group Bys, are usually much smaller than the operand tables, and thus an in-memory sort may be adequate to sort the results by the second alternative.

However, neither post-retrieval sorting nor prior sorting/hashing meets our performance requirement, since the system cannot respond until either the aggregation plus a post-retrieval sort are completed, or the whole input table is sorted/hashed.

Index-based access methods can produce results in the desired order due to the sorted nature of the indexed keys. So can bitmap-based access methods. However, in practice one problem arises. Secondary access paths result in random accesses which are much more costly than sequential scans. As a result, index-based methods do not necessarily perform better than sort or hash based methods which produce sorted data by sequential scans. To tackle this problem, one may optimize the I/Os of index-based methods to meet the page-I/O nature, and remedy the higher costs of random I/Os.

Two basic techniques to optimize I/Os are: 1) avoiding reading the same page more than once, and 2) reading neighboring pages in a single I/O by reading some unnecessary pages in between [77]. The first technique minimizes the number of I/Os, while the second one minimizes the time of I/Os. The two techniques are not conflicting, but complementary.

As discussed before, bitmaps have the duality property, i.e., they encapsulate both addressing information at tuple- and page-levels. Although the page-level addresses are implicit, the transformation from the tuple-level bitmaps to their page-level counterparts can be piggy-backed on other bitmap operations (a detailed discussion will be found in following sections). Therefore, they are done without much extra cost. Once the page-level bitmaps are available, the above two techniques can be applied to find an optimal access plan using low overhead bitwise operations on the bitmaps.

³Methods of evaluating aggregates directly on bit-sliced indexes are discussed in [65].
By using the above feature, bitmap-based access methods can produce results in desired order without the costly sorting or hashing on the huge fact tables, which makes bitmap-based methods ready for pipelining. This is especially favorable in Data Warehouses.

### 4.2 Selection Evaluation

In Chapter 3, the basics of variant bitmap indexes and their usages for selection evaluation are introduced. In this section, let us concentrate on advanced optimization techniques for both encoded bitmap indexes and bit-sliced indexes for selections. The discussions are divided into two parts — static optimization and dynamic optimization. For static optimization, i.e., optimization at design time, an optimal algorithm for continuous (Type 1) range selections using bit-sliced indexes is introduced. For dynamic optimization, i.e., optimization at run time, two techniques are discussed. One of these uses encoded bitmap indexes and the other uses bit-sliced indexes.

#### 4.2.1 Static optimization

In [16], an algorithm — RangeEval-Opt (listed in Appendix C.1 for reference), was proposed to improve its former version proposed in [65]. A further improvement of RangeEval-Opt is proposed in [92].

In order to clearly show the improvement to the original algorithm, let us follow the example in [16] and reveal the idea in a pictorial way. Suppose that a 3-component range \( \leq \) bit-encoded bit-sliced index with decimal base on attribute * and the predicate “*0, !=” are given. Using Algorithm RangeEval-Opt to evaluate the predicate results in the execution tree as shown in Figure 4.4(a), which is equivalent to \( b_7^3 + b_8^3 (b_8^2 + b_8^0 b_4^1) \).

![Figure 4.4: Transformation of the execution tree for \( A \leq 864 \). OR denotes OR and AND denotes AND](image)

Figure 4.4: Transformation of the execution tree for \( A \leq 864 \). OR denotes OR and AND denotes AND

Suppose that for a certain running state of a database, the second digit of all the values of * is no larger than 5, i.e., in the component-2 of the index, the bit vectors \( b_7^2, b_8^2, b_7^2 \) and \( b_8^2 \) are all set to “1”. By replacing the corresponding bit vectors with “1” vectors, we have the first tree in Figure 4.4(b). By applying \( x \cdot 1 = x \) (identity law) and \( x + 1 = 1 \) (dominance law) of Boolean algebra, the execution tree is reduced to one node, i.e., instead of 5 bitmap scans plus 4 logical operations (the tree in Figure 4.4(a)), only 1 bitmap (the bit vector \( b_8^3 \) of component-3) is read.
The saving is significant.

The reduction of a partial sub-tree can also help to reduce bitmap processing costs. Following the above example, if the most significant digit of \( A \) is no larger than 8, then the original execution tree can be replaced by the tree in Figure 4.5(a). Applying the rule \( x \cdot 1 = x \), we can connect the right sibling subtree of the 1-node directly to the root, as shown in Figure 4.5(b). The computation complexity is reduced from 5 bitmap scans plus 4 logical operators to 4 bitmap scans and 3 logical operators. For equality predicates (\( =, \neq \)), the idea also works.

![Figure 4.5: Transformation of execution tree for \( A \leq 864 \)](image)

Obviously, in order to be able to apply such a reduction, the information about the percentage of population of each bit vector is needed. Without much extra cost, this information can be computed at the time of index creation and can be synchronized every time the data are uploaded in batch mode to the Data Warehouse. The revised version of RangeEval-Opt, which implements the above tree-reduction technique, is defined in Algorithm 4.

### 4.2.2 Dynamic optimization

In this section, query optimization strategies at run time is introduced. Cost models are also defined for the cost-efficiency analysis.

As discussed before, selection evaluation consists of two phases — the index scanning phase and the data fetching phase. In the index scanning phase, selection predicates are evaluated on indexes and the results of this phase are used as access plans in the data fetching phase. The idea of the principle of inclusion and exclusion\(^9\) is as follows: by using a “coarse” filtering (instead of exact matching), the time of the index scanning phase is reduced. However, the desired data as well as some undesired data is included in the second phase. Therefore extra tests must be performed to exclude the unqualified data. If the extra cost of eliminating superfluous tuples in the second phase is smaller than the time-saving in the first phase, query performance improves.

\(^9\)The name, the principle of inclusion and exclusion, has been borrowed from the set theory. It describes a different scenario here from that in the set theory.
The basic idea of the inclusion/exclusion approach is to identify the conditions under which an approximate access plan in the index scanning phase will still provide a globally optimized query response- and processing-time.

The following example illustrates how the idea works. Suppose we have a selection predicate “\( A \in \{u, v, w\} \)”, and an encoded bitmap index on \( A \) is defined as shown in Figure 4.6. The retrieval Boolean function for \( A \in \{u, v, w\} \), \( f_u + f_v + f_w \), is reduced to \( b_3 b_2 (b_1 + \bar{b}_0) \). (The function is illustrated using a Karnaugh Graph in Figure 4.7(a).) That is, 4 bitmap scans plus 5 logical operators are required in the index scanning phase.

If we include the value \( t \) in the selection predicate and make it “\( A \in \{t, u, v, w\} \)”, as shown in Figure 4.7(b), then the corresponding retrieval function, \( f_t + f_u + f_v + f_w \), is reduced to \( b_3 b_2 \). The time complexity is reduced from 4 bitmap scans plus 5 logical operators to 2 bitmap scans plus 1 logical operator.

However, the deliberate inclusion of \( t \) in the index scanning phase causes extra cost in the

---

**Algorithm 4** Range evaluation with tree-reduction

Input: A bit-sliced index with the base, \(< b_N, \ldots, b_1 >\), where \( N \) is the number of components and \( b_j \) denotes the \( j \)-th bit vector of \( i \)-th component. For a bit vector, \( b_i, \Theta(b_i) \) denotes the percentage of “1”’s in \( b_i \).

Selection predicate \( A \ op \ v \), where \( op \in \{<, >, \leq, \geq, =, \neq\} \).

Output: A bit vector representing the set of tuples which satisfy the selection predicate, \( A \ op \ v \).

1) Begin
2) \( B = 1; \)
3) if (\( op \in \{<, \geq\} \)) then \( v = v - 1; \)
4) decompose \( v \) into \( v_N v_{N-1} \cdots v_1; \)
5) if (\( op \in \{<, >, \leq, \geq\} \)) then
   6) if (\( v_i < b_i-1 \) and \( \Theta(b_{v_i}) \neq 1 \)) then \( B = b_{v_i}; \)
   7) for \( i = 2 \) to \( N \)
      8) if (\( v_i < b_i-1 \) and \( \Theta(b_{v_i}) \neq 1 \)) then \( B = b_{v_i} \cdot b_i; \)
      9) if (\( v_i \neq 0 \) and \( \Theta(b_{v_{i-1}}) \neq 1 \)) then \( B = b_{v_{i-1}} \cdot b_i; \)
   10) else
      11) for \( i = 1 \) to \( N \)
          switch (\( v_i \))
          12) case \( v_i = 0: \)
              13) if (\( \Theta(b_i) \neq 1 \)) then \( B = b \cdot b_i; \)
          case \( v_i = b_i - 1: \)
          15) if (\( \Theta(b_{b_i-1}) \neq 1 \)) then \( B = b \cdot b_{b_i-1}; \)
      16) else return (\( B = \emptyset; \))
          case 0 \( < v_i < b_i - 1: \)
      18) if (\( \Theta(b_{b_i}) \neq 1 \) and \( \Theta(b_{v_{i-1}}) \neq 1 \)) then \( B = b \cdot (b_{v_i} \oplus b_{v_{i-1}}); \)
      19) else if (\( \Theta(b_{b_{i-1}}) = 1 \) then return (\( B = \emptyset; \))
          21) else \( B = b \cdot b_{b_i-1}; \)
      22) end switch
      23) if (\( op \in \{>, \geq\} \)) then return \( \emptyset \) else return \( B; \) /* filter out non-existent tuples before return */
24) End

**Encoded bitmap indexes**

The basic idea of the inclusion/exclusion approach is to identify the conditions under which an approximate access plan in the index scanning phase will still provide a globally optimized query response- and processing-time.

The following example illustrates how the idea works. Suppose we have a selection predicate “\( A \in \{u, v, w\} \)”, and an encoded bitmap index on \( A \) is defined as shown in Figure 4.6. The retrieval Boolean function for \( A \in \{u, v, w\} \), \( f_u + f_v + f_w \), is reduced to \( b_3 b_2 (b_1 + \bar{b}_0) \). (The function is illustrated using a Karnaugh Graph in Figure 4.7(a).) That is, 4 bitmap scans plus 5 logical operators are required in the index scanning phase.

If we include the value \( t \) in the selection predicate and make it “\( A \in \{t, u, v, w\} \)”, as shown in Figure 4.7(b), then the corresponding retrieval function, \( f_t + f_u + f_v + f_w \), is reduced to \( b_3 b_2 \). The time complexity is reduced from 4 bitmap scans plus 5 logical operators to 2 bitmap scans plus 1 logical operator.

However, the deliberate inclusion of \( t \) in the index scanning phase causes extra cost in the
Figure 4.6: An example of encoded bitmap indexing on $A$

Figure 4.7: Karnaugh graph of retrieval Boolean functions

second phase. For the above example, if we use the resulting bitmap of $b_2b_2$ to fetch the data, some undesired tuples are also read, which might take extra I/O time, and in order to filter those undesired tuples out of the final result, extra CPU time is involved. In spite of the extra cost, the inclusion/exclusion approach might still result in better query performance due to the characteristics of page-I/Os and different distributions of underlying data.

In order to determine which of the two query execution plans (exact match or coarse filtering) is better, the following cost model is proposed.

**Cost Model for Inclusion-AND-Exclusion Method**

The cost of query processing is defined by the total number of pages read in both the index scanning and the data fetching phases. Let us first define the terms and the extra data structure used in the cost model.

A **tuple-level bitmap** uses one bit for one tuple. A bit is set to 1 if the attribute of the corresponding tuple is equal to the index key; otherwise it is set to 0. Analogous to a tuple-level bitmap, a **page-level bitmap** uses one bit for one page. A bit is set if the attribute of any tuple in the corresponding page equals the index key. That is, the number of bits in a page-level bitmap indicates the number of pages which will be accessed in order to fetch data with the given index key. A page-level bitmap is derived from a given tuple-level bitmap, and is defined as follows.

**Definition 10 (Page-Level Bitmaps)** Suppose we have a bitmap (also named tuple-level bitmap), $b$, which is $n$ bits in length, denoted by $|b| = n$, and let $\pi$ be the logical page size in bytes and $p$ be the blocking factor of a page (i.e., the number of database objects in a page). The page-level bitmap of $b$ with respect to $p$ — denoted by $b^p$ — is a bitmap with $m$ bits, where $m = \lceil n/p \rceil$. The bits in $b^p$ is defined by

$$\forall j \ (1 \leq j \leq m), \ \exists i \ (1 \leq i \leq p), \ \exists b[(j-1)p + i] = 1, \Rightarrow b^p[j] = 1,$$
where $b^P[j]$ denotes the $j$-th bit of $b^P$.

Figure 4.8 depicts the construction of a page-level bitmap from its tuple-level bitmap. The tuple-level bitmap is divided into $m$ $p$-bit segments, except the last one. Each $p$-bit segment of the tuple-level bitmap corresponds to one bit in the page-level bitmap. The bits in the page-level bitmap are set if any bit in their corresponding $p$-bit segments is set. Other terms are defined in Table 4.1.

![Figure 4.8: Transforming tuple-ids into data-block-ids](image)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>a bit vector</td>
</tr>
<tr>
<td>$</td>
<td>b</td>
</tr>
<tr>
<td>$\mathcal{T}$</td>
<td>a table</td>
</tr>
<tr>
<td>$</td>
<td>\mathcal{T}</td>
</tr>
<tr>
<td>$w(\mathcal{T})$</td>
<td>width of table $\mathcal{T}$ in bytes</td>
</tr>
<tr>
<td>$\pi$</td>
<td>logical page size in bytes</td>
</tr>
<tr>
<td>$p = \left\lceil \frac{\pi}{w(\mathcal{T})} \right\rceil$</td>
<td>blocking factor (the number of tuples per page for $\mathcal{T}$)</td>
</tr>
<tr>
<td>$b^P$</td>
<td>a page-level bitmap of $b$ with respect to $p$</td>
</tr>
<tr>
<td>$\Omega(b)$</td>
<td>the number of &quot;1&quot; bits in the bit vector $b$</td>
</tr>
<tr>
<td>$\Theta(b)$</td>
<td>the percentage of &quot;1&quot; bits in $b$, $\Theta(b) = \frac{</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>the number of variables reduced during the minimization of the retrieval Boolean function</td>
</tr>
<tr>
<td>$\delta$</td>
<td>the cardinality of a range selection</td>
</tr>
</tbody>
</table>

Table 4.1: Notations used in the cost analysis of the inclusion/exclusion approach

For the above example, the cost-efficiency analysis is performed as follows. Using the exact-match approach, in the index scanning phase the bitmap $B = b_3b_2(b_1 + b_0)$ is evaluated, and the result is used to access the desired tuples in the data fetching phase. The total I/O cost of this approach will be

$$\frac{4 \cdot |b|}{8\pi} + \Omega(B^P) \text{ pages,} \tag{4.7}$$

where the first term denotes the costs of the index scanning phase, and the second term denotes the costs of the data fetching phase. On the other hand, using the approach of inclusion/exclusion, the bitmap $B = b_3b_2$ is evaluated in the index scanning phase, and $B$ is used
to fetch data in the second phase. The total I/O cost is

$$\frac{2 \cdot |b|}{8\pi} + \Omega(B^p) \text{ pages.} \quad (4.8)$$

If Equation (4.8) is less than Equation (4.7), it is beneficial to perform the inclusion/exclusion approach.

$$\frac{2 \cdot |b|}{8\pi} + \Omega(B^p) < \frac{4 \cdot |b|}{8\pi} + \Omega(B^p) \implies \Omega(B^p) - \Omega(b^p) < \frac{|b|}{4\pi}$$

A general form for the cost-efficiency analysis can be derived as follows.

$$\Omega(b^p) - \Omega(b^p) < \frac{\gamma \cdot |b|}{8\pi}, \quad (4.9)$$

where $b$ is the resulting bitmap of the exact-match approach, $b_\phi$ is the resulting bitmap of the inclusion/exclusion approach, $b^p$ and $b^p_\phi$ are the page-level bitmaps of $b$ and $b_\phi$, respectively, and $\gamma$ is the number of variables (bitmaps) which are reduced from the retrieval Boolean function after the inclusion of additional values into the selection predicate. Simply speaking, the right hand side of Inequality (4.9) denotes the total number of I/O-savings in pages gained through the principle of inclusion and exclusion in the index scanning phase, and the left hand side denotes the additional I/O cost of reading the extra data pages arising from the approach of inclusion/exclusion in the data fetching phase. If Inequality (4.9) is true, then the inclusion/exclusion approach provides a better query performance.

In practice, the calculation of the term $\Omega(b^p) - \Omega(b^p)$ (in Inequality (4.9)) could be expensive and I/O intensive, since both page-level bitmaps $b^p$ and $b^p_\phi$ have to be read. In order to reduce the overhead of optimization, the above term can be replaced with an approximation using a statistical model. That is, it can be estimated by the expected number of page accesses. The expected number of page accesses of a query is defined as a function of the selectivity of the query and can be computed by the following probability model.

As discussed in Section 2.1.1, we can use the result of the coloring problem (Equation (2.1)) to estimate the number of page accesses. Recall that the expected number of colors by selecting $k$ balls of $n$ different colors, if for each color there are $p$ identical balls, is

$$n \cdot \left(1 - \prod_{r=1}^{k} \frac{pn \cdot \frac{n-1}{n} - r + 1}{pn - r + 1}\right)$$

That is, given a query $Q$, which selects $k$ tuples from $n$ pages, where each page can accommodate $p$ tuples, the expected number of page hits by $Q$, denoted by $E(Q)$, is

$$E(Q) = n \cdot \left(1 - \prod_{r=1}^{k} \frac{pn \cdot \frac{n-1}{n} - r + 1}{pn - r + 1}\right) \text{ pages.}$$

The inclusion/exclusion approach increases the selectivity of the query by including additional values into the range selection. For example, the number of selected tuples is changed to $k'$ and the revised query is denoted by $Q'$. Then, the expected number of page hits by $Q'$, denoted by $E(Q')$, is

$$E(Q') = n \cdot \left(1 - \prod_{r=1}^{k'} \frac{pn \cdot \frac{n-1}{n} - r + 1}{pn - r + 1}\right) \text{ pages.}$$
Now, Inequality (4.9) can be estimated by

\[ E(Q') - E(Q) < \frac{\gamma \cdot |b|}{8\pi} \]  \hspace{1cm} (4.10)

**What to Include**  Another issue concerning the inclusion/exclusion approach is what to include. This itself is an optimization problem. Here a quick way to examine how much reduction an inclusion can eventually achieve is provided. The examination is especially useful to quickly identify the negative cases, where no effort to optimize the query is worth the time the optimization takes, or where the query cannot ever be optimized. Therefore, before investing much time in trying to optimize a query, it would be preferable to know if it is definitely not worth doing it.

Recall the cardinality of a range selection, denoted by \( \delta \), defined in Section 3.3.1. Given an attribute \( A \) and a range selection on \( A \), \( \{A \ op \ V\} \), the cardinality of the range selection is the cardinality of the set \( S \), such that \( S = \{v|v \in A, \ and \ v \ op \ V\} \). If \( V \subseteq A \), then \( |S|= |V| \). Following the above example, for the selection, \( \{A \in \{u, v, w\}\} \), since \( \{u, v, w\} \subseteq A \), the cardinality of the selection range is \( |\{u, v, w\}| = 3 \). For later discussion, it is assumed that \( V \subseteq A \).

To find out what to include and to avoid unnecessary attempts, one must answer the following questions: at least how many additional values must be included in the operand set, \( V \), in order to make a further reduction? And, how much reduction can be achieved?

To answer the first question, a simple test on the operand set, \( V \), is performed. If the cardinality of the range selection, denoted by \( |V| \), is odd, then a further reduction is possible by including one additional value into \( V \). If \( |V| \) is even and \( |V| \neq 2^n \ (n \in \mathbb{N}) \), then at least two values must be included in \( V \) to make a further reduction possible. In addition, at most \( 2^j - (|V| - 2^j) \) values are included in \( V \) to make a further reduction possible, where \( i \) is the largest integer such that \( \frac{|V|}{2^i} > 1 \), and \( j \) is the smallest integer such that \( \frac{2^j}{|V|} > 1 \). The reason why the minimum number of additional inclusions is one or two is behind the idea of making a \( 2^n \ (n \in \mathbb{N}) \) neighboring cells in the Karnaugh graph, as Figure 4.7(b) shows. Note that this is only the necessary condition. Satisfying this condition does not imply the existence of a reduction.

To answer the second question, we should explore the relationship between the cardinalities of range selections and the probably minimal numbers of bitmap scans in the index scanning phase. Assuming the existence of a well-defined encoding, Table 4.2 lists the minimal number of bitmap scans with respect to different cardinalities of range selections for attribute \( A \) and \( |A| = 8 \). For example, if we extend the cardinality of the range selection from 3 to 4, in the best case, the number of bitmap scans could drop from 3 to 1.

<table>
<thead>
<tr>
<th>Cardinality of selection (( \delta ))</th>
<th>Optimized number of bitmap scans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: Optimized number of bitmap scans with respect to cardinality of range selection

The computation of the table is based on Property 2 in Section 3.3.1. It describes the scenario:
for the best case, to process a selection on \( A \), the number of bitmap scans is \( \left\lfloor \log_2 |A| \right\rfloor - \theta(\delta) \), where

\[
\theta(\delta) = \left\{ \begin{array}{ll}
p, & \text{if } (\delta \text{ modulo } 2^p) = 0, \\
q, & \text{if } ((\delta \text{ modulo } 2^p) \text{ modulo } 2^{q+1}) = 2^q, \ 0 \leq q \leq p - 1,
\end{array} \right.
\]

and \( p \) is the largest integer, such that \( \frac{\delta}{2^p} \geq 1 \).

Note that the cardinality of range selection does not unconditionally imply the number of bitmap scans as listed in Table 4.2. Nevertheless, the numbers provide a quick estimation of the saving in the index scanning phase and the cost arising in the data fetching phase. As an extreme example of this, if the cardinality of selection is expanded to 8 in Table 4.2, although the cost of the index scanning phase is reduced to 0, a table scan is required to access all the data, since all attribute values are included in the predicate.

In addition, Table 4.2 is used not only to check how much reduction an inclusion might lead to, but also to determine how effective an inclusion is. An effective inclusion is one that can lead to further reduction of the retrieval function. For example, to expand the cardinality of a selection from 4 to 6 can never be an effective inclusion.

For some situations, a coarse inclusion might perform even better than the best inclusion if the costs of finding the inclusion is considered. For example, instead of finding the best inclusion, the cardinality of selection range can be easily expanded by finding the common variables of the retrieval min-terms. For the example in the beginning of this section, the common variables of \( b_3b_2\overline{b}_1 \) and \( b_3b_2\overline{b}_0 \) are \( b_3b_2 \), i.e., expanding \( b_3b_2(\overline{b}_1 + \overline{b}_0) \) to \( b_3b_2 \). In this example, it is also the best expansion. Although there might exist a better inclusion, the cost of finding it might not be compensated by its benefit. Algorithm 5 finds the set of common variables of the retrieval Boolean function at a complexity of \( O(|V|) \).

**Algorithm 5** Finding common variables

Input: Selection predicate \( A \in V, V = \{v_1, \ldots, v_k\} \)

- An EBI on \( A \) with the mapping function \( M : A \rightarrow \{b_{n-1} \cdots b_0\} | b_i \in \{0, 1\}, 0 \leq i < n \}
- A set of Boolean variables used in the retrieval min-terms, \( \{b_{n-1}, \ldots, b_0\} \)

Output: A set of common variables, \( C \), of the retrieval function for \( A \in V \)

1) \hspace{1em} Begin
2) \hspace{2em} set \( C = \emptyset \)
3) \hspace{2em} initialize an array of bits, \( B[n] \)
4) \hspace{2em} set \( B = 1 \)
5) \hspace{2em} for \( i = 2 \) to \( k \)
6) \hspace{3em} \( B = B \cdot (M(v_{i-1}) \odot M(v_i)) \)
7) \hspace{3em} /* \odot \text{ denotes exclusive-NOR, } \cdot \text{ denotes AND, and } M(v_i) \text{ denotes the encoded value of } v_i */
8) \hspace{2em} for \( i = 0 \) to \( n - 1 \)
9) \hspace{3em} if \( B[i] = 1 \) then \( C = C \cup \{b_i\} \)
10) \hspace{2em} return \( C \)
11) \hspace{1em} End

So far, we have discussed the application of the inclusion/exclusion approach to encoded bitmap indexes for the selection operator “\( \in \)”. For finite domains, other operators, such as \( \subseteq, \neq, < \text{ and } > \), can all be rewritten using \( \in \). Therefore, the above discussion does not lose its generality.
Bit-sliced indexes

Now, how to apply the inclusion/exclusion principle to numerical data using bit-sliced indexes is shown. Following the example in section 4.2.1, to evaluate the predicate “$A \leq 864$”, the execution tree is shown in Figure 4.4(a).

The idea is to expand the range of the selection such that tree-reduction on the execution tree occurs, e.g., by enlarging the range of the selection from “$A \leq 864$” to “$A \leq 894$”, the execution tree is reduced as Figure 4.9(a) shows.

![Figure 4.9: Execution trees for $A \leq 894$ and $A \leq 869$](image)

As another example, if we change the predicate to “$A \leq 869$”, the execution tree is reduced further, as Figure 4.9(b) shows. We can see that the scale of enlargement in selection ranges does not imply the scale of reduction in bitmap scans. The latter example reduces the number of bitmap scans to 3, while the former example reduces the number of bitmap scans to 4, in spite of larger expansion of the selection range.

Inequality (4.10) is also used as the cost model to determine whether an expansion in selection range improves the query performance or not. Instead of using page-level bitmaps to compute the extra cost arising in the data fetching phase, the distribution of the underlying data is used to estimate the change in query selectivity. For the above example, assuming the attribute values of $A$ are evenly distributed within $100 \leq A \leq 900$, changing the predicate from “$A \leq 864$” to “$A \leq 869$” increases the query selectivity by $\frac{5}{801}$, while changing the predicate to “$A \leq 894$” increases the query selectivity by $\frac{30}{801}$.

For numeric data types with even distribution, it is preferable to expand the selection range of the least significant digit first, since the higher the query selectivity increases, the higher the extra cost in the data fetching phase.

### 4.3 Optimizing Group Bys and Aggregation Using Bitmaps

The implementation of relational operators is often affected by the amount of memory available. One-pass algorithms, which carry out the operations in one single pass, require enough memory to accommodate the whole intermediate results, or the operand tables being sorted in the order required by the operation. In the Data Warehousing environment, it is not reasonable to make the assumptions that intermediate results fit into the main memory, or that the
Operand tables are sorted in desired order.

In the relational algebra, many operators, such as Group Bys and Joins, involve one basic physical operation — matching. That is, data items, which match some given conditions, are brought together, followed by specific operations. In the case of Group Bys and aggregations, data are aggregated and duplicates are eliminated. In the case of Joins, Cartesian Products are performed on data items of the same group from two operands. In the literature, two basic approaches are used to perform “matching” — sorting and hashing [38].

For either sorting or hashing approaches, once the data sets are larger than the available memory, the algorithms are implemented as multi-pass iterators. Multi-pass sorting/hashing algorithms have some points in common. They divide the operands recursively until the space requirements of the algorithms are met, then are followed by a merge phase. The algorithms cannot feed any data to the consumer-operators before the merging phase begins. In addition, the intermediate results before the merge begins must be materialized, as Figure 4.10 shows.

![Data flows of multi-pass sort-merge algorithms](image)

Figure 4.10: Data flows of multi-pass sort-merge algorithms

This also means that the system will not respond before the whole operand table is partitioned and the merging phase begins. In Data Warehouses, the partitioning will take quite a long time, since the fact tables are usually huge. The effects of these approaches will cascade if subsequent operators still apply sorting/hashing methods to perform matching operations. In the following sections, another approach that performs the matchings using bitmap indexes without physically sorting or hashing the operand tables is introduced. Furthermore, the algorithms are implemented as one-pass operations in the experimental version of bitSQL. For cases of overflows (i.e., results cannot be held in memory), we will discuss the technique used to handle overflows.

### 4.3.1 Optimizing Group Bys

**Performing Group Bys before Joins**

In [94] and [18], the advantages of processing Group Bys before Joins are discussed. Stated briefly, early processing of Group Bys reduces the size of intermediate results that are fed to the
join operator, and thus reduces the costs of performing the join. Here a further optimization of Group Bys by exploring functional dependencies among attributes is introduced. Through this optimization, some unnecessary Joins, typically key/foreign-key Joins, can be even skipped in query processing. The following example illustrates the idea.

**SQL 7**

```sql
SELECT c.mktsegment, SUM(l.qty) FROM lineitems l, orders o, customers c WHERE l.order_id = o.order_id AND o.cust_id = c.cust_id GROUP BY c.mktsegment
```

The query trees of SQL 7 are depicted in Figure 4.11. Figure 4.11(a) shows the execution tree generated by traditional query optimizers which postpone Group Bys until all Joins are done. Figure 4.11(b) shows an execution tree resulting from pushing down the Group Bys past Joins. The Group Bys at different heights of the tree reduce the size of the intermediate results, and thus eventually reduce the I/O costs of subsequent Joins.

If we follow the execution plan of the query tree in Figure 4.11(b) and introduce the usage of bitmap indexes at the lowest operator level, i.e., Group By `order_id`, we still have to apply hashing or sorting to evaluate the Group Bys of `cust_id` and `mktsegment`, since no index is defined on the intermediate results. In such a case we lose the benefit of bitmaps. Therefore, it is desirable to push all the Group Bys down to the lowest operator level, as shown in Figure 4.11(c).

```
Join
orders lineitems
Join
lineitems
Group By mktsegment
sum (qty)
Group By cust_id
Join
customers
Join
orders
Group By (order_id's)
lineitems
```

Figure 4.11: Execution trees of SQL 7

Pushing all the Group Bys to the level of the base table is possible whenever functional dependencies exist that uniquely determine the grouping attributes at the higher level. Recall that in query-rewriting introduced in Section 4.1.2, functional dependencies guarantee the disjointness constraints. This situation will always arise when snow-flaking was used to break out dimensional hierarchies. In the example, `order_id` functionally determines `cust_id`, which in turn functionally determines `mktsegment`. Since functional dependency is transitive, `order_id` determines `mktsegment`. Therefore, the Group By can be executed on the corresponding subsets of `order_id` since there exists one partitioning on the attribute domain of `lineitems.order_id` that divides its attribute domain into several disjoint subsets, and all the values in one subset map to the same value of `mktsegment`. If no other attributes in the dimension table(s) are required by other operators, the Joins in the dotted area can even be omitted.
The logical query plans representing the execution trees in Figure 4.11(a), (b) and (c) are

\[
\begin{align*}
\Sigma_{\text{mktsegment}, \sum(\text{qty})} (\text{lineitems} \bowtie \text{orders} \bowtie \text{customers}), \\
\Sigma_{\text{mktsegment}, \sum(\text{qty})} (\text{customers} \bowtie \sum_{\text{cust}.id, \sum(\text{qty})} (\text{orders} \bowtie \Sigma_{\text{order}.id, \sum(\text{qty})} (\text{lineitems}))) \\
\Sigma_{P(\text{order}.id), \sum(\text{qty})} (\text{lineitems}),
\end{align*}
\]

respectively. \(P(\text{order}.id)\) is a partitioning on \(\text{lineitems}.\text{order}.id\) using \(\text{customers}.\text{mktsegment}\), and is defined as follows.

Let \(P(\text{order}.id) = \{p_1, \ldots, p_m\}\), then

\[
\forall t, s \in \text{lineitems} \ (t \neq s), \ \text{and} \ {t.\text{order}.id, s.\text{order}.id} \subseteq p_j \ (1 \leq j \leq m)
\]

\[
\exists r, q \in \text{orders}, \ \text{and} \ u, v \in \text{customers},
\]

\[
\exists t.\text{order}.id = r.\text{order}.id, r.\text{cust}.id = u.\text{cust}.id, \ \text{and} \ s.\text{order}.id = q.\text{order}.id, q.\text{cust}.id = v.\text{cust}.id
\]

\[
\implies u.\text{mktsegment} = v.\text{mktsegment}
\]

That is, all \(\text{order}.id\)s submitted by the customers who belong to the same \(\text{mktsegment}\) are brought into the same group.

There are many approaches which can be applied to perform the partitioning described above. For example, the most straightforward one is to group the joined table of both \(\text{orders}\) and \(\text{customers}\) on \((\text{mktsegment}, \text{order}.id)\). Since the theme of this work is bitmap indexing, the approaches exploiting group-set bitmap indexes are shown next.

An efficient approach is to define group-set bitmap indexes on potential grouping attributes at design time. In our example above, if a group-set bitmap index is defined on \(\text{lineitems}.\text{order}.id\) using \(\text{mktsegment}\), then the partitioning is done without extra cost at run time. Each group, \(p_i\) in the above example, is represented by one bitmap, and the physical query plan of SQL 7 is mapped to a sequence of selections followed by a union, as follows.

\[
\Sigma_{P(\text{order}.id), \sum(\text{qty})} \text{lineitems} = \bigcup_{m \in \text{mktsegment}} \pi_m, \sum(\text{qty}) (\sigma_{f_m} \text{lineitems})
\]

If, however, no group-set index is defined on \(\text{lineitems}.\text{order}.id\) using \(\text{mktsegment}\), then the group-set bitmaps can still be built on the fly, with help of the bitmap index on its dimension key, i.e., \(\text{order}.id\). (It is assumed that bitmap indexes are defined on dimensional foreign keys of the fact table(s).) The construction of group-set bitmaps is expressed in tuple relational calculus, just to show its feasibility.

For all \(m \in \text{customers}.\text{mktsegment}\), \(f_m\) denotes the retrieval Boolean function for the value \(m\) of \(\text{mktsegment}\); \(g_i\) is the retrieval Boolean function for the value \(i\) of \(\text{order}.id\), and \(\cup\) denotes the logical disjunction. Then, \(f_m\) is defined as follows.

\[
f_m = \cup g_i, \quad \text{where} \ i \in O_m, \ \text{and} \ O_m = \{t.\text{order}.id | \forall t \in \text{lineitems}\}
\]

such that

\[
\exists r \in \text{orders}, u \in \text{customers},
\]

\[
\exists t.\text{order}.id = r.\text{order}.id, r.\text{cust}.id = u.\text{cust}.id \ \text{and} \ u.\text{mktsegment} = m
\]
Evaluating Group Bys using bitmaps

Now, having the group-set bitmaps, we are ready to design the algorithms for Group By evaluation. The challenges here are 1) to guarantee short response time, 2) to produce intermediate results in desired order to enable pipelining, and 3) to cope with the page-I/O nature to minimize the I/O costs. To guarantee short response time, index-based accesses are applied to perform the matchings of the Group By operator. Neither sorting nor hashing methods are used, since they cannot respond before the merging phase begins, and the partitioning phase may take quite long in Data Warehouses where tables to be sorted/hashed are immense. In addition, since it is the nature of index-based algorithms to produce results in some sort order, pipelining subsequent operations can be achieved without extra effort. However, although index-based algorithms can precisely locate the desired tuple, they often result in random accesses and duplicate reads of the same page, which in turn result in large I/O costs and poor performance. To tackle this problem the solutions are introduced in two stages. First, assume that the memory size is large enough to accommodate the aggregation results, the bitmaps and other space needed by the algorithm. This assumption helps to reveals the simplicity of our algorithm. Then, later in the second stage the assumption is relaxed and a general algorithm is proposed.

UNLIMITED BUFFER SIZE  Like most indexes, bitmaps provide precise locations of individual indexed data items. Conceptually, the Group By evaluation using bitmaps is quite easy and straightforward. Following the example of SQL 7, suppose that we have four group-set bitmaps defined on lineitems using mktsegment, as Figure 4.12 shows.

For each value in the domain of mktsegment, say “AUTOMOBILE”, the grouping bitmap b_A is used to retrieve all the tuples which belong to the group “AUTOMOBILE”. In this example, tuples with order_ids in \{000161, 000196, 000199, 000295\} are grouped together, and attribute values of qty of these tuples are summed up. Then, the resulting tuple (AUTOMOBILE, 799) is fed to the end user. The same procedure goes for “BUILDING”, “FURNITURE” and “HOUSEHOLD”. The final results will be
However, in computer systems, I/O operations are page-I/Os. That is, one physical page is the minimal unit of an I/O operation. Therefore, to implement an algorithm in a real system, the effect of page-I/Os should be taken into consideration. For the above example, suppose that three tuples fit into one physical page. Then, following the conceptual procedures described above, to evaluate the aggregation of the group “AUTOMOBILE”, pages 1, 2 and 3 are read. To aggregate the group “BUILDING”, pages 1, 2 and 3 are read; pages 1 and 4 are accessed for the group “FURNITURE”, and no page is accessed for the group “HOUSEHOLD”. As a result, page 1 is read three times, page 2 and 3 are both read twice, and page 4 is read once.

Taking into account the effects of page-I/Os, the above procedures are revised slightly to minimize the number of I/Os, such that each page is accessed exactly once. Similarly, for each value in the domain of mktsegment, say “AUTOMOBILE”, the grouping bitmap $b_A$ is used to retrieve all the pages which contain tuples belonging to the group “AUTOMOBILE”. As Figure 4.12 shows, pages 1, 2 and 3 are accessed. However, unlike the former method, to maximize the pay-offs of each I/O operation, once a page is moved into the main memory, all tuples on that page are processed, not just the tuples belonging to the group of concern. One flag is used for each physical page to indicate whether or not the page has been read. For read pages, they need not be accessed again.

For example, to process the group “AUTOMOBILE”, pages 1, 2 and 3 are moved into the memory one by one. After a page is moved into memory, attribute qty of all the tuples is cumulated into a hash-table. The hash-table contains one entry for each group, and tuples of the same group are hashed into the same entry. After reading pages 1, 2 and 3, the hash-table contains the data, as Figure 4.13(a) shows. At this time, the entry (AUTOMOBILE, 799) is finished and can be pipelined to subsequent operations, or to the end user.

![States of the hash-table after reading pages 1, 2 and 3](image1)

After evaluating the group “AUTOMOBILE”, pages 1, 2 and 3 are marked as “read”. While evaluating the group “BUILDING”, we find out that all the pages containing tuples of the group have been processed. Then, without further processing, the entry (BUILDING, 1614) is sent to the output buffer. By evaluating the group “FURNITURE”, page 4 is read, and the tuple is hashed and cumulated into the hash-table. The state of the hash-table after processing page 4 is shown in Figure 4.13(b). The entry (FURNITURE, 953) can now be sent to the output buffer. Since no tuple exists for the group “HOUSEHOLD”, the processing ends.

![States of the hash-table after reading page 4](image2)
The above revised method produces results in some order without physically sorting or hashing the entire operand table. Instead, it exploits indexes to obtain a sequence of page accesses to guarantee the ordered outputs. It involves I/Os of reading four pages and the group-set bitmaps, and it requires one input buffer and the space for the hash-table and the read/unread flags. The above procedures are summarized in Algorithm 6.

Algorithm 6 uses a sequence of page-level group-set bitmaps to evaluate the Group By operation. Note that the grouping bitmaps have already filtered out the unqualified tuples by bitwise ANDing their tuple-level counterparts to the resulting bitmap of selection evaluation before being converted to page-level bitmaps. In addition, the ordering of the output is predefined to the algorithm and is given by the sequence of grouping subset, \( \langle g_1, \ldots, g_k \rangle \). (Later other cases in which the order is not predefined to the evaluation are discussed.)

**Algorithm 6 bitmapGrouping** — Group By evaluation using bitmaps

Remark: Incremental evaluation of aggregates and Group Bys with unlimited buffer size

Input: Grouping attribute(s) \( G \), whose domain is further divided into a sequence of grouping subsets, \( \langle g_1, \ldots, g_k \rangle \), \( g_i \subseteq \text{domain}(G) \) and \( \{b_1, \ldots, b_k\} \) are the grouping bitmaps for \( \{g_1, \ldots, g_k\} \), respectively.

An aggregate function, \( f \),

The attribute to be aggregated, \( A \), and

An operand table, \( T \)

Output: grouping results

1) Begin
2) define a bit vector Read with the same length as \( b_i \), \( 1 \leq i \leq k \);
3) let \( \text{Read} = 0 \);
4) define a perfect hashing function, \( h_t \), such that \( h_t(v) = h_t(v') \), \( \forall v, v' \in g_i \) \( 1 \leq i \leq k \);
5) define a hash table, \( HT \), \( HT[i] \) denotes the \( i \)-th entry of \( HT \);
6) for each \( b_i \) in the sequence \( \{b_1, \ldots, b_k\} \)
7) for each \( j \)-th bit in \( b_i \), denoted by \( b_i[j] \)
8) if \( (b_i[j] \text{ and } \sim \text{Read}[j]) \) /* \( \sim \) denotes the negation operator */
9) read in the \( j \)-th page of \( T \), denoted by \( T[j] \);
10) for each tuple \( t \) in \( T[j] \)
11) cumulate \( t.A \) into \( HT[h_t(t.G)] \);
12) \( \text{Read}[j] = 1 \);
13) output \( f(HT[h_t(g)]) \), for any \( g \in g_i \);
14) End

Figure 4.14 illustrates the execution of Algorithm 6. For the first group \( g_1 \), its corresponding page-level bitmap \( b_1 \) is used to access the pages that contain tuples belonging to \( g_1 \) (called “\( g_1 \)-pages” for short). While processing \( g_1 \)-pages, tuples in the \( g_1 \)-pages that belong to other groups are also hashed and cumulated, as the dotted lines indicate. After scanning all the \( g_1 \)-pages, the aggregate result of the first group can be pipelined to the next operator, or to the end user.

For the second group \( g_2 \), \( b_2 \) is used to access the \( g_2 \)-pages. Those \( g_2 \)-pages which are also \( g_1 \)-pages can be omitted, since they have been processed while processing the group \( g_1 \). The information about which page has been read is stored in an auxiliary bit vector, \( \text{Read} \). For pages which have been processed, their corresponding position in the \( \text{Read} \) bit vector is set. In Figure 4.14, the processed pages are illustrated in brighter grey, the pages being accessed are in darker grey, and the pages in white are so far irrelevant. After scanning all the \( g_2 \)-pages which
have not been read before, the result of \( g_2 \) can be pipelined to the next operator, and the same procedure is repeated until all the groups are evaluated. Algorithm 6 produces results stepwise and feeds them to the next operator or users without reading any page more than once.

**LIMITED BUFFER SIZE** If the Group By result cannot fit into main memory, Algorithm 6 is revised and the buffer size is considered as an input parameter to the algorithm. The basic idea is to keep a moving window of the set of aggregates which are being grouped. The window size is determined by the buffer size. In Algorithm 7, the window size is given and denoted by \( w \). At the beginning, the first \( w \) groups (i.e., \( g_1, \ldots, g_w \)) from the total \( k \) groups are evaluated, as Figure 4.15(a) shows. The evaluation of the \( w \) groups is similar to that in Algorithm 6. The selection bitmaps, \( b_1, \ldots, b_w \), are used sequentially to determine which page should be read. Once a page is moved into main memory, instead of all tuples, only those tuples which belong to any of the \( w \) groups are processed. After all \( g_1 \)-pages are read, the result for \( g_1 \) is pipelined to the next operation and its entry in the hashing table can now be used for \( g_{w+1} \). Note that after one \( g_1 \)-page is processed, its corresponding bits in all \( b_1, \ldots, b_w \) are cleared. So later while processing any group of \( g_2, \ldots, g_w \), no page is accessed twice. The window moves on and includes \( g_{j+w} \), after \( g_j \) (1 \( \leq j \leq k-w \)) is finished, as Figures 4.15(b) and (c) show.

In Algorithm 6 and 7, groups are processed according to their orderings in the sequence of \( \langle g_1, \ldots, g_k \rangle \), which is predefined to our algorithms. Many factors, such as performance requirements or users’ requirements, may affect the policies and the mechanisms of defining...
Algorithm 7 bitmapGrouping — Group By evaluation using bitmaps with limited buffers

Remark: Incremental evaluation of aggregates and Group Bys with limited buffer size

Input: Grouping attribute(s) \( G \), whose domain is further divided into a sequence of grouping subsets, \( \{g_1, \ldots, g_k\}, g_i \subseteq \text{domain}(G) \) and \( \{b_1, \ldots, b_k\} \) are the grouping bitmaps for \( \{g_1, \ldots, g_k\} \), respectively

An aggregate function, \( f \),
The attribute to be aggregated, \( A \),
An operand table, \( T \), and
The buffer size \( w \)

Output: grouping results

1) Begin
2) define a hash table, \( HT \), of size \( w \), where \( HT[i] \) denotes the \( i \)-th entry of \( HT \), \( 1 \leq i \leq w \);
3) define a perfect hashing function, \( ht \), such that \( ht(v) = ht(v'), \forall v, v' \in g_i(w) + 1 \);
4) let \( m = 1, n = w \);
5) for each \( b_i \) in the sequence \( \{g_1, \ldots, g_k\} \)
   for each \( j \)-th bit in \( b_i \), denoted by \( b_i[j] \)
   if \( b_i[j] \)
     read in the \( j \)-th page of \( T \), denoted by \( T[j] \);
9) for each tuple \( t \) in \( T[j] \)
10) cumulate \( t.A \) into \( HT[ht(t.G)] \), if \( t.G \in g_i, m \leq l \leq n \);
11) clear all \( b_i[j], \forall m \leq l \leq n \);
12) output \( f(HT[ht(g)]) \), for any \( g \in g_i \);
13) let \( m = m + 1, n = \min(n + 1, k) \);
14) End

such sequences. However, no matter what is chosen, it does not affect the evaluation algorithms, as long as the sequences are predefinable. In the following, the cases where the order of processing is not pre-definable to the evaluation algorithm are discussed since they affect the basics of the algorithm.

4.3.2 Optimizing Order Bys

In users’ queries, Order By clauses may be specified to sort the outputs. The query results may be either ordered by attribute(s) or by aggregation. In particular, we are interested in the latter case where the sort order is based on the aggregation. For example, the user wants to aggregate \( \text{sum}(l.qty) \) by \text{mktsegment} \( c \), but needs the results to be sorted by the aggregation — \( \text{sum}(l.qty) \), as SQL 8 defines.

SQL 8

```sql
SELECT c.mktsegment, SUM(l.qty)
FROM lineitems l, orders o, customers c
WHERE l.order_id = o.order_id AND o.cust_id = c.cust_id
GROUP BY c.mktsegment
ORDER BY SUM(l.qty)
```

Since the ordering and the aggregates are based on the same Group By criterion, we cannot determine the ordering before we evaluate the Group By and the aggregates. As a result, the system cannot respond until all the operations are performed. However, if we evaluate the operations simultaneously, there is a chance to shorten the response time. The idea is to perform
the aggregation and sorting at the same time, until the ordering among the aggregates is found, then the remaining aggregation is carried out group by group (as Algorithms 6 and 7 do).

Algorithm 8 requires both tuple-level grouping bitmaps and a binary bit-sliced index on the attribute to be aggregated. It applies both indexes to evaluate the aggregates digit by digit (beginning with the most significant digit), and at the same time the incomplete aggregates are sorted. As long as a total ordering among the incomplete aggregates exists, the remaining evaluation is carried out in the ordering found above to enable feeding results as early as possible.

Through the use of bitmap indexing and the auxiliary information that is automatically available in such an index, i.e., the number of tuples in each group, it is possible to determine the ordering in the set of incomplete aggregates and thus to continue evaluating the aggregates in the right order to enable pipelining, as early as possible. Let us see how the algorithm works on the database snapshot, shown in Figure 4.16(a). For the sake of clarity, decimal digits are used, instead of binary digits, in this explanation. (In Algorithm 8, binary bit slices of the aggregate attribute are used.) Nonetheless, the mathematical correctness of the algorithm is not affected.

### Algorithm 8 bitmapOrderedGrouping — Bitmap grouping involving Order By aggregation

Remark: Incremental evaluation of aggregates and Group Bys ordered by aggregates

**Input:** Grouping attribute(s) $G$, whose domain is further divided into a sequence of grouping subsets, $(g_1, \ldots, g_k)$, $g_k \subseteq \text{domain}(G)$, and $\{b_1, \ldots, b_k\}$ are the tuple-level grouping bitmaps for $(g_1, \ldots, g_k)$, respectively

An aggregate function, $f$

The attribute to be aggregated, $A$

Numbers of tuples in each group, $c_1, \ldots, c_k$

An operand table, $T$, and vertical bitwise partition on $A$, denoted by $A_{m-1}, \ldots, A_0$

**Output:** grouping results

1. **Begin**
2. **define** arrays $\text{sum}[]$ and $\text{seq}[]$ of $k$ integers;
3. let $\text{sum}[i] = b_i^T \cdot A_{m-1}, i = 1, \ldots, k$; /* $b_i^T$ denotes the transpose of the bit vector $b_i$ */
4. assign $\text{seq}[i], i = 1, \ldots, k$, such that $\text{sum}[\text{seq}[1]] \leq \text{sum}[\text{seq}[2]] \leq \cdots \leq \text{sum}[\text{seq}[k]]$;
5. let $h = 1$;
6. repeat
   7. let $\text{done} = \text{FALSE}$;
   8. for $i = 2$ to $k$
   9. if $(2^m - h \cdot (\text{sum}[\text{seq}[i]] - \text{sum}[\text{seq}[i-1]]) < c_{\text{seq}[i-1]} \cdot (2^m-h-1))$
   10. $\text{done} = \text{FALSE}$; /* test whether a total ordering in $\text{sum}[]$ exists */
   11. break;
   12. if (not $\text{done}$)
   13. let $h = h + 1$;
   14. let $\text{sum}[i] = 2 \cdot \text{sum}[i] + b_i^T \cdot A_{m-h}, i = 1, \ldots, k$;
   15. assign $\text{seq}[i], i = 1, \ldots, k$, such that $\text{sum}[\text{seq}[1]] \leq \cdots \leq \text{sum}[\text{seq}[k]]$;
   16. until (done or $h \geq m$);
   17. if ($h < m$)
   18. evalRestAgg($\text{sum}[\text{seq}[1]], \ldots, \text{sum}[\text{seq}[k]], \theta, T, A, (h+1)$, $\langle b_{\text{seq}[1]}, \ldots, b_{\text{seq}[k]} \rangle, \langle g_{\text{seq}[1]}, \ldots, g_{\text{seq}[k]} \rangle$);
19. return $\text{sum}[]$;
20. **End**
The execution of SQL 8 using Algorithm 8 is discussed as follows. First, \( b_i^T \cdot A_2 \) is calculated, where \( b_i^T \) is the transposed vector of \( b_i \) (\( i \in \{A, B, F, H\} \)), and \( A_2 \) is the vector of the decimal digit of \( \text{qty} \) at position \( 10^2 \). Bitmaps \( b_A, b_B, b_F \) and \( b_H \) are (tuple-level) grouping bitmaps for groups “AUTOMOBILE”, “BUILDING”, “FURNITURE” and “HOUSEHOLD”, respectively. The partial aggregate results are saved in the table, as Figure 4.16(b) shows. After cumulating the most significant digits of each group, sums of all four groups are 6, 14, 8 and 0, respectively. Now, we want to test whether or not there exists a total ordering among the final results based on the partial aggregates (line 9 in Algorithm 8). Property 3 helps us with this task.

**Property 3** Given two sets of positive integers, \( A = \{A_1, \ldots, A_m\} \) and \( B = \{B_1, \ldots, B_n\} \), suppose that each number is expressed by \( k \) decimal digits, where \( k = \lceil \log_{10} \max(A_1, \ldots, A_m, B_1, \ldots, B_n) \rceil + 1 \). \( A_{i,j} \) denotes the digit of \( A_{i} \) at the position \( 10^j \) (\( 0 \leq j < k \)). Then, for any \( r \) (\( 1 \leq r \leq k \)), if

\[
10^{k-r} \cdot \sum_{i=1}^{m} (A_{i,k-1} \cdot 10^{r-1} + \cdots + A_{i,k-r}) > 10^{k-r} \cdot \sum_{j=1}^{n} (B_{j,k-1} \cdot 10^{r-1} + \cdots + B_{j,k-r}) + n \cdot (10^{k-r} - 1)
\]

\[\implies \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j \]

Property 3 describes a strong condition such that if it is satisfied, the relation between the sums of two sets of integers can be determined. What is useful about this property is that we do not really need to sum up all the integers in both sets to be able to determine the relation of the sums. Instead, sums of partial digits of the integers may be adequate to determine the relation. Although Property 3 describes decimal numbers, it is applicable to numbers with any integral base. At line 9 in Algorithm 8, we apply Property 3 to binary numbers. The proof of Property 3 and also of the correctness of Algorithm 8 is listed in Appendix A.7.

Let us continue the execution of Algorithm 8. For the group “BUILDING”, to the best of our knowledge with the data being processed so far, the lower bound of the summary of \( \text{qty} \) is \( 14 \times 10^2 \); and, for the group “FURNITURE”, the upper bound of the summary of \( \text{qty} \) is \( 8 \times 10^2 + 99 \times 2 \), since there are two tuples in this group. Without further processing, we can assure that the \( \text{sum} \) (\( \text{qty} \)) of “BUILDING” is larger than that of “FURNITURE”, since \( 1400 > 998 \), i.e., even if all the other digits of \( \text{qty} \) of the group “FURNITURE” are 9, the \( \text{sum} \) (\( \text{qty} \)) of “FURNITURE” will still be smaller than that of “BUILDING”. (The condition described in Property 3 is satisfied with \( r = 1 \).) However, we cannot determine the ordering between the summaries of “AUTOMOBILE” and “FURNITURE”, since \( 8 \times 10^2 \geq 6 \times 10^2 + 99 \times 4 \). That is, if all the other digits of \( \text{qty} \) of the group “AUTOMOBILE” are 9, the \( \text{sum} \) (\( \text{qty} \)) of “AUTOMOBILE” could be \( 6 \times 10^2 + 99 \times 4 \) (since, there are four tuples in this group), which will be larger than the summary of “FURNITURE”. Therefore, we have to continue the evaluation by summing up one more digit, i.e., the digit at the position \( 10^1 \). (Line 13 and 14 in Algorithm 8)

Figure 4.16(c) depicts the intermediate results after summing up \( A_1 \), i.e., \( 10 \times A_2 \times b_i^T + A_1 \times b_i^T \) (\( i \in \{A, B, F, H\} \)). From the partial aggregates in Figure 4.16(c), we are now able to determine an ordering among the summaries of all the groups. We know that \( \text{sum} (\text{BUILDING's qty}) > \text{sum} (\text{FURNITURE's qty}) > \text{sum} (\text{AUTOMOBILE's qty}) > \text{sum} (\text{HOUSEHOLD's qty}) \). Since,
If we proceed with the evaluation by calling Algorithm 6 (or 7), the system will not respond until all tuples of the group "AUTOMOBILE" are read. On the other hand, alternative 2 does not produce any result until the end of processing. Therefore, the "response" time of "AUTOMOBILE" is chosen.

Let us consider the following three possible scenarios concerning the response time of alternative 2 is chosen. In Figure 4.17, the processing time of alternative 2 is shorter than the response time of alternative 1, so alternative 2 is chosen.

Although the goal is to minimize the response time, the query processing time should also be kept small. Let us consider the following three possible scenarios concerning the response time and the query processing time of the two alternatives in Figure 4.17. Since alternative 1 produces results stepwise, the time when the first partial result is available is depicted as the time of "response". On the other hand, alternative 2 does not produce any result until the end of processing. Therefore, the "response" point in time for alternative 2 is also the end of processing.

For cases in Figure 4.17(a) and (b), the choice of processing methods is obvious and unambiguous. In Figure 4.17(a), both the response time and the processing time of alternative 1 are shorter than those of alternative 2. Therefore, alternative 1 is chosen. In Figure 4.17(b), the processing time of alternative 2 is shorter than the response time of alternative 1, so alternative 2 is chosen.

After the ordering among the groups has been determined, there are two alternatives to complete the evaluation. The first is to proceed the evaluation by calling Algorithm 6 (or 7) with the sequence of ordered groups as an input parameter. The second is to proceed with the remaining evaluation by cumulating the remaining digits into the table shown in Figure 4.16(c).

Since the main performance goal is to minimize the response time, the method with shorter response time is chosen. In the above example (assuming that we want to evaluate the aggregates in descending order), if we proceed with the evaluation by cumulating the remaining digits into the table shown in Figure 4.16(c).
In Figure 4.17(c), the response time of alternative 1 is shorter than that of alternative 2, but its processing time is longer than that of alternative 2. No matter which processing method is chosen, there are always some tradeoffs. Before defining the criteria for choosing either of the processing methods, let us consider two extreme cases of Figure 4.17(c). In Figure 4.18(a), although the response time of alternative 1 is shorter than that of alternative 2, its processing time is much longer than that of alternative 2. Under some circumstances, the user may be willing to wait another second or two, but in return, she/he gets the complete answer set, instead of only some partial answers.

Figure 4.18(b) shows another extreme. The processing time of alternative 2 is shorter than that of alternative 1, but the difference is minor. In this case, it is obvious that the system will respond much earlier by choosing alternative 1 than by choosing alternative 2 and that this is done without much performance degradation due to longer processing time.

Based upon the above observation, we can define the criteria for choosing processing methods. The notations used are defined as follows. The response time of alternative 1 is denoted by $r$, and its total processing time by $t$. The processing time of alternative 2 is defined by $s$. The scenario in Figure 4.17(c) is divided into three sub-cases. First, let us consider the case where $\frac{r}{t} < \theta$, then the one with $\frac{r}{t} > 1 - \theta$, and finally $\frac{r}{t} \leq 1 - \theta$, where $0 < \theta < 0.5$. The three cases are illustrated in Figure 4.18(a), (b) and (c). Note that $0 < \frac{r}{t} < 1$, since $0 < r < s < t$.

The condition, $\frac{r}{t} < \theta$, describes the situation where, simply stated, the processing time of alternative 2 is significantly less than that of alternative 1, or $s << t$, and $\theta$ is the significance threshold. In principle, if $\frac{r}{t} < \theta$, then alternative 2 is preferable since, in spite of shorter response time, the performance degradation due to longer processing time is significant with respect to $\theta$. The choice of $\theta$ is an issue concerning human factors, i.e., the study of users’ behavior. For example, how long can a user wait without impatiently cancelling the task? Or, how long does a user spend in reading one result before going to the next one? How to choose $\theta$ is beyond the scope of this work and it will not be discussed further. More details about the range of $\theta$ can be found in Appendix C.2.
The second scenario describes the case where the difference between the processing time of both alternatives is insignificant with respect to $\theta$, i.e., $s \approx t$. For such cases, alternative 1 is preferable, since the processing time of both alternatives is similar, but alternative 1 gains performance from early response.

The above two scenarios are less ambiguous, once $\theta$ is defined. As for cases where $\theta \leq \frac{s}{t} \leq 1-\theta$, further criteria to choose a better processing method are defined. If $\frac{s-r}{r} < \theta$, then alternative 2 is preferable, since $(s-r)$ is insignificant to $r$ with respect to $\theta$. If $\frac{s-r}{r} \geq \theta$, then alternative 1 is preferable. The reason for this proposal is as follows. Since $\frac{s-r}{r} < \theta$, it means that by choosing alternative 2 the extra delay for response, i.e., $ZX_X$, is insignificant to the response time of alternative 1 with respect to $X$. In contrast, if the extra delay is significant, i.e., $\frac{s-r}{r} \geq \theta$, alternative 1 is chosen due to the shorter response time in spite of its longer processing time.

The above discussion leads to the following heuristics.

Choose alternative 1, if $s \approx t$, or $\theta \leq \frac{s}{t} \leq 1-\theta$ and $r << s-r$ \hspace{1cm} (4.11)

Choose alternative 2, if $s << t$, or $\theta \leq \frac{s}{t} \leq 1-\theta$ and $r \approx s$ \hspace{1cm} (4.12)

In Algorithm 9, the function `evalRestAgg()` performs the above test to determine which alternative is used to proceed the aggregation.

**Algorithm 9** evalRestAgg

Remark: Follow-up Evaluation of aggregates

Input: A sequence of grouping subsets, $\langle g_1, \ldots, g_k \rangle$, a sequence of tuple-level bitmaps, $\langle b_1, \ldots, b_k \rangle$,
An array, `sum[]`, which stores the partial summaries,
The significance threshold, $\theta$,
The attribute to be aggregated, $A$, and its bit slices $A_{m-1}, \ldots, A_0$
The radix position, $h$, with which the follow-up summing-up should begin
The operand table, $T$; $|T|$ denotes $T$’s cardinality

Output: grouping results

1) Begin
2) let $b_i^p$ be $b_i$’s page-level bitmap, $i = 1, \ldots, k$, and $|b_i^p|$ denotes the number of 1-bits in $b_i^p$;
3) let $s = estimateCost$(reading $m-h+1$ bit vectors of length $|T|$);
4) let $r = estimateCost$(reading $b_i^p$ pages), $t = estimateCost$(reading $T$);
5) if ($\frac{s}{r} > 1-\theta$) or ($\theta \leq \frac{s}{r} \leq 1-\theta$ and $\frac{s-r}{r} \geq \theta$)\n6) call `sum[] = bitmapGrouping`($\langle g_1, \ldots, g_k \rangle, \langle b_1^p, \ldots, b_k^p \rangle, A, T, BUFSIZE$);
7) else\n8) let $\text{sum}[i] = 2 \times \text{sum}[i] + b_i^T \cdot A_{m-j}, i = 1, \ldots, k$ and $j = h, \ldots, m$;
9) return `sum[]`;
10) End

4.3.3 Aggregation on dimension tables

In the above discussions, one of the favorable consequences of defining group-set bitmap indexes is making Star-Joins redundant in query processing. The group-set bitmaps are the result of precomputation of the Join and some matching operations. In addition, building group-
set bitmap indexes does not even require Joins to be performed if bitmap indexes on dimension key(s) exist. (cf. the definition of group-set bitmap indexes in Section 2.3.5 and rewriting groupings on dimension keys in Section 4.1.2) Note that group-set bitmaps are not restricted to evaluating Group Bys, as their name indicates. They can be used to evaluate other matching operations, like selections.

In the examples of SQL 7 and 8, the core business data, i.e., lineitems, are to be grouped by the dimension attribute mktsegment. The Star-Joins between lineitems, orders and customers serve to combine the fact table with the dimension tables, such that the grouping by dimension(s) can be carried out on the fact table. If the grouping bitmaps for mktsegment exist (either predefined or constructed on the fly), the grouping can be directly performed on the fact table without joining the fact table to the dimension tables, since the function of the Joins has already been fulfilled by the grouping bitmaps.

However, if operations other than matchings, are given on dimensions, the necessity of Star-Joins has to be reexamined. One such case is aggregation on dimension tables. Aggregate functions are usually given on additive attributes in the fact table(s) where the core business data reside. However, syntactically and also semantically, we cannot exclude the cases of aggregating dimensional attributes. For example, SQL 9 summarizes lineitems.price and parts.price by parts.brand. The price attribute in lineitems models the sales-price, while the one in parts models the purchase-price.

SQL 9

\[
\begin{align*}
\text{SELECT} & \quad \text{p.brand, SUM(l.price), SUM(p.price)} \\
\text{FROM} & \quad \text{lineitems l, parts p} \\
\text{WHERE} & \quad \text{l.part_id = p.part_id} \\
\text{GROUP BY} & \quad \text{p.brand}
\end{align*}
\]

In this section, some preprocessing is introduced to the bitmap-based grouping algorithms introduced above, such that aggregation on dimensions, along with aggregation on fact table(s), can be evaluated without performing Star-Joins. Let us illustrate the idea in an example first. Suppose that we want to evaluate SQL 9 on the database instance shown in Figure 4.19(a) and (b). The bitmap index, \( \text{parts.part_id} \), on lineitems.part_id is defined.

Remember the evaluation of Group Bys discussed in last section. Since a bitmap index\(^\text{10}\) on the foreign-key, part_id, to the dimension, parts, is defined, the grouping by parts.brand can be transformed to the grouping by subsets of part_id, leaving the Star-Join out of query execution. For example, according to the snapshot of parts shown in Figure 4.19(a), the domain of brand is equal to \( \{w, x, y, z\} \). For each value in brand, its member part_ids are defined as follows.

\[
\begin{align*}
w & = \{P00032, P00054, P00061\} \\
x & = \{P00002, P00026\} \\
y & = \{P00020, P00063, P00096\} \\
z & = \{P00095, P00099\}
\end{align*}
\]

\(^{10}\)For the sake of clarity, simple bitmap indexes is used to illustrate the ideas. Nevertheless, the methods described here are not restricted to simple bitmap indexes.
Now, we can construct the grouping bitmaps for brand using \( \mathcal{P}_{part_id} \).

\[
\begin{align*}
\mathbf{b}_W &= \mathbf{b}_{52} + \mathbf{b}_{54} + \mathbf{b}_{61} \\
\mathbf{b}_X &= \mathbf{b}_{62} + \mathbf{b}_{26} \\
\mathbf{b}_Y &= \mathbf{b}_{50} + \mathbf{b}_{63} + \mathbf{b}_{96} \\
\mathbf{b}_Z &= \mathbf{b}_{90} + \mathbf{b}_{99}
\end{align*}
\]

For each value of brand, the bitmaps for all member part_ids are \textit{ORed} to form the grouping bitmap. While constructing grouping bitmaps, the 1 bits of each bitmap for part_ids is counted. For instance, while \textit{ORing} bitmaps, \( \mathbf{b}_{52} \), \( \mathbf{b}_{54} \) and \( \mathbf{b}_{61} \), the 1 bits in each bitmap are also counted.\(^{11}\) The number of 1 bits in the bit vector \( \mathbf{b} \) is denoted by \( \Omega(\mathbf{b}) \). The sum of \( \text{parts, price} \) for the group...
"w" is

\[ \Omega(b_{32}) \times 400 + \Omega(b_{54}) \times 476 + \Omega(b_{61}) \times 70 = 1416 \]

Similarly, the sums of parts.price for "x", "y" and "z" are

\[ \begin{align*}
\Omega(b_{22}) \times 385 + \Omega(b_{26}) \times 755 &= 2280, \\
\Omega(b_{30}) \times 122 + \Omega(b_{03}) \times 260 + \Omega(b_{96}) \times 130 &= 902, \text{ and} \\
\Omega(b_{98}) \times 260 + \Omega(b_{99}) \times 315 &= 1150,
\end{align*} \]

respectively. We can see that the aggregates on dimension(s) are resolved by simple preprocessing that is piggy-backed on the construction of the grouping bitmaps. In general, for aggregate functions, \textit{sum()}, \textit{count()} and \textit{average()}, the resolutions using the above approach are defined as follows.

\[ \begin{align*}
\text{sum}(G_i.A) &= \sum_{g_{i,j} \in G_i} |g_{i,j}| \times g_{i,j}.A, \ G_i \in G, \\
\text{count}(G_i.A) &= \sum_{g_{i,j} \in G_i} |g_{i,j}|, \\
\text{average}(G_i.A) &= \text{sum}(G_i.A)/\text{count}(G_i.A),
\end{align*} \]

where \( G \) is the grouping attribute, \( G_i \) is a groupset in \( G \), \( g_{i,j} \) denotes a member of the groupset \( G_i \), and \( A \) is the dimension attribute to be aggregated. The value \( |g_{i,j}| \) is the number of occurrences of \( g_{i,j} \) in the fact table.

The advantages of the above approach are threefold. First, the joins to dimension tables in this circumstance are no longer necessary and are replaced with a counting on bitmaps and some simple arithmetical calculation. Second, the whole procedure does not induce much extra cost, since it is piggy-backed on the construction of the grouping bitmaps which are required for later grouping. Third, this approach will even shorten the response time by feeding the available aggregates on dimension attributes.

After resolving aggregates on dimension tables, the rest of the queries can be handled by Algorithm 6 or 7 accordingly.

A more general case

The above example involved only a single Join and a single Group By. Nonetheless, the above method can be extended to cover general cases, where queries involve multiple Joins (or, even Joins along the dimension hierarchies), and multiple Group Bys. For the example in SQL 10, suppose that we want to query the summaries of the sales-price and the average subscription per order and break them up by nations and market-segments of customers.\(^{12}\)

\(^{12}\)It is worth mentioning here that if attribute \textit{price} in table \textit{orders} models the total amount placed by an order, the aggregate \textit{"AVG(o.price"} results in a \textit{weighted average subscription per order. An order appears in the joined table of \textit{lineitems} and \textit{orders} for as many times as the number of parts placed in the order. Therefore, the subscription of the order is also summed up as many times as the number of parts in that order. That is, the price is weighted by the number of parts placed in an order. However, the semantical correctness of the query is not the concern, since it depends on the semantics behind the modeling. In query processing, the main concern is the correctness of the results based upon the syntax only, not upon the semantics.
To evaluate the above SQL, the grouping by dimension(s) is rewritten to the grouping on the fact table’s foreign-key(s). A brute-force approach is introduced to illustrate its feasibility, but not to indicate the implementation details. Using the query-rewrite techniques described above and bitmap indexes on the foreign-key of the fact table, “Group By c.nation_id” is transformed to a multi-value “Group By l.order_id”. One multi-value set of l.order_id corresponds to a single value of c.nation_id, and the grouping bitmap is represented by a bitmap expression of ORing all the bitmaps of the order_ids in the set. Similarly, “Group By c.mktsegment” is transformed to a multi-value “Group By l.order_id”, and one multi-value set of l.order_id map tuples of the same market-segment into the same group. The grouping bitmaps are also expressed by logical disjunctions of all bitmaps of the order_ids in the same multi-value set. Then, the grouping bitmaps for “Group By c.nation_id, c.mktsegment” can be obtained by a Cartesian Product of logical conjunctions of every pair of bitmap expressions from both group-sets c.nation_id and c.mktsegment.

For the database instance shown in Figure 4.20, suppose that we have a simple bitmap index defined on lineitems.l.order_id, and the last two digits of order_ids are used as subscripts of bitmaps. The grouping bitmap for the group (01, AUTOMOBILE) will be

\[
\left(\sum_{\text{nation_id}=01} \sum_{\text{mktsegment}=\text{AUTOMOBILE}} (b_{01} + b_{03} + b_{02} + b_{04} + b_{05} + b_{06})\right)
\]

It is further reduced to

\[
(b_{01} + b_{03} + b_{05} + b_{06})
\]

That is, the grouping bitmap for (01, AUTOMOBILE) is constructed by logical disjunction of the bitmaps for order_id equal to 000001, 000003, 000005 or 000006. On constructing the grouping bitmap, the functions, \(\Omega(b_{01}), \Omega(b_{03}), \Omega(b_{05})\) and \(\Omega(b_{06})\), are also evaluated. The aggregate AVG(o.price) for the group (01, AUTOMOBILE) can now be resolved by the following expression:

\[
\frac{\Omega(b_{01}) \times 250 + \Omega(b_{03}) \times 300 + \Omega(b_{05}) \times 190 + \Omega(b_{06}) \times 260}{\Omega(b_{01}) + \Omega(b_{03}) + \Omega(b_{05}) + \Omega(b_{06})}
\]

The same procedures are applied to resolve aggregates of other groups on o.price. After the grouping bitmaps are generated, the other aggregate, SUM(l.price), is evaluated by Algorithm 6 or 7 accordingly.

| Customers | customers | nation_id | mktsegment | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C00001</td>
<td>01</td>
<td>AUTOMOBILE</td>
<td></td>
</tr>
<tr>
<td>C00002</td>
<td>01</td>
<td>FURNITURE</td>
<td></td>
</tr>
<tr>
<td>C00003</td>
<td>01</td>
<td>AUTOMOBILE</td>
<td></td>
</tr>
</tbody>
</table>

| orders | orders_id | cust_id | price | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>C000001</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>000002</td>
<td>C000002</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>000003</td>
<td>C000001</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td>C000002</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>000005</td>
<td>C000003</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>000006</td>
<td>C000003</td>
<td>260</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.20: Snapshots of dimension tables customers and orders
Two points need to be mentioned here. First, the counting process can be quite expensive for encoded bitmap indexes, since all bitmaps in encoded bitmap indexes need to be processed. Therefore, as a design guideline, it would be preferable to maintain the information about the number of 1 bits, i.e., the number of occurrences of the value in the fact table, in the mapping table. In addition, other statistical information, such as selectivities of each bitmap, can also be collected during the creation of bitmaps, and can be incrementally maintained at the time of data updates. This information can be created/maintained with low overhead.

Second, although the algorithms introduced above are applicable when aggregating/grouping data of different granularities within a single query, such as is done by the CUBE operator, the resulting solutions will not be globally optimal. The optimization of the CUBE operator has two levels. The higher one is to determine a sequence of sub-cube calculations, and the lower one is the optimization within each sub-cube calculation [1]. The algorithms introduced in this work can be applied to the second level of optimization.

4.3.4 Related work

**Generalized Hash Teams** To the best of our knowledge, the work most related, but not directly comparable, to the bitmap-based algorithms for aggregation is the generalized hash teams in [55]. Generalized hash teams are used to evaluate matching operations, such as Join and Group By in Data Warehouses. Both the bitmap-based algorithms and hash teams enable pipelining in some order. The main differences are as follows: First, [55] is a hash-based method. In [55], a hash-team, i.e., a hashing on the grouping attribute and an indirect hashing on the join attributes, is built and used to partition the operand tables. Two or more cross-relation partitions from different operand tables can be joined and aggregated directly. In this work, with the help of bitmap indexes, aggregations and grouping are evaluated directly on fact tables in desired order without further hashing/sorting or partitioning the operand tables, even if the grouping results do not fit into main memory. Joins which serve as bridges for grouping between dimensions and fact tables are transformed to grouping using bitmaps.

Second, the indirect hashing proposed in [55] may produce false drops. A false drop is an unnecessary duplication of one tuple in two different partitions, which incurs higher I/O cost later in query processing. Besides, operand tables must be scanned and partitioned, which in turn induces the costs of writing out partitions and reading them in later, while in our approach, aggregates are directly evaluated on base tables without the cost of writing/reading intermediate partitions. It is achieved with the help of the exact information about the location of each tuple provided by bitmap indexes. In addition, in order to maximize the utilization of each page access and to avoid reading any page more than once, several groupsets are evaluated simultaneously in our algorithms. The window size of simultaneous groupsets is determined by the memory size.

Third, how Order By clauses affect the execution of hash-teams is not discussed in [55]. In this work, since one of the goals is to enable pipelining and feeding sorted results, we have to explore different cases of Order By clauses and discuss how they affect the aggregate evaluation.

**Online Aggregation** Due to users’ expectation of rapid response and of interaction with systems based upon former query results on the fly in decision making environments, online

---

13 Cross-relation partitions are those which have the same hashing value on the join attribute.
aggregation becomes an interesting issue [49, 46, 36, 69, 47]. Generally speaking, online aggregation provides approximates of the final results, allowing users to observe the progress of aggregation and eventually control/abort the execution on the fly. The performance goals are 1) to optimize the time to accuracy trade-off and 2) to minimize the time to completion, as defined in [49]. On the other hand, in this work, our performance goals are 1) to minimize the time to the first precise result and 2) to minimize the time to completion. In addition, by feeding precise results in some order enables pipelining, which the estimates of online aggregation approaches do not allow. Using the estimated completion time (as will be discussed in Section 4.3.5), users can observe the progress of the execution. However, in this work reordering the execution on the fly is not allowed in the algorithms to meet the users’ interest. The execution orders are defined prior to aggregation.

### 4.3.5 Performance of bitmap-based grouping algorithms

**Analytical Approach** In this subsection, the expected response time function and the worst-case execution time function for the bitmap-based grouping algorithms are derived. The notations used in the analysis are defined in Table 4.3, and the performance of algorithms is measured by the number of page-I/Os.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>a table</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$w(T)$</td>
<td>width of table $T$ in bytes, i.e., the length of a tuple in $T$</td>
</tr>
<tr>
<td>$\pi$</td>
<td>physical page size in bytes</td>
</tr>
<tr>
<td>$p = \lfloor \frac{\pi}{w(T)} \rfloor$</td>
<td>blocking factor (the number of tuples per page for $T$)</td>
</tr>
<tr>
<td>$g_i$</td>
<td>a multi-value groupset</td>
</tr>
<tr>
<td>$G$</td>
<td>the set of all groupsets, $G = {g_i}</td>
</tr>
<tr>
<td>$</td>
<td>G</td>
</tr>
<tr>
<td>$w$</td>
<td>window size, i.e., the number of groupsets fit into the main memory at one time</td>
</tr>
<tr>
<td>$\sigma(g_i)$</td>
<td>fraction of selected tuples in $g_i$ to the cardinality of $T$</td>
</tr>
<tr>
<td>$\Sigma(g_i)$</td>
<td>fraction of selected pages in $g_i$ to the total number of pages in $T$</td>
</tr>
<tr>
<td>$B_{g_i}$</td>
<td>tuple-level grouping bitmap for the groupset $g_i$</td>
</tr>
<tr>
<td>$B_{\gamma_i}$</td>
<td>page-level grouping bitmap for the groupset $g_i$</td>
</tr>
<tr>
<td>$\Omega(B_{g_i})$</td>
<td>the number of 1-bits in $B_{g_i}$, i.e., $\sigma(g_i) = \Omega(B_{g_i})/</td>
</tr>
<tr>
<td>$\Omega(B_{\gamma_i})$</td>
<td>the number of 1-bits in $B_{\gamma_i}$, i.e., $\Sigma(g_i) = \Omega(B_{\gamma_i}) \cdot p/</td>
</tr>
<tr>
<td>$E(g_i)$</td>
<td>the expected number of page-I/Os in processing the groupset $g_i$</td>
</tr>
<tr>
<td>$C_{\gamma_i}$</td>
<td>clustering factor of the groupset $g_i$</td>
</tr>
</tbody>
</table>

Table 4.3: Notations used in performance analysis of bitmap-based grouping algorithms

The algorithms (bitmapGrouping and bitmapOrderedGrouping) evaluate aggregation group by group. Their response time will be the execution time until the result of the very first groupset, say $g_\alpha$, is available. It can be expressed by

$$E(g_\alpha) = \frac{|T|}{p} \cdot \left(1 - \prod_{j=1}^{k} \frac{|T| - p - j + 1}{|T| - j + 1}\right), \quad \text{where } k = \sigma(g_\alpha) \cdot |T|$$  \hspace{1cm} (4.16)

The above equation is the expected number of pages being hit with respect to the selectivity $\sigma(g_\alpha)$. Note that it is calculated based on the assumption that the tuples are randomly...
distributed among the table space. However, in reality, due to the batch-mode data loading nature, the warehouse data will present some degree of clustering. For example, data of the same date from the same store are very likely to be loaded in contiguous blocks. In such cases, the real number of pages hit will be smaller than that of Equation (4.16). Later we will discuss the effects of the clustered nature of the warehouse data in our simulation, and define the term clustering factor.

As for the execution time, it is the time elapsed until the last groupset is finished. Similarly, the execution time is measured by the total number of page-I/Os. Since in our algorithms no page is accessed twice or more within a moving window, the worst-case is that for every \( w \) groups the fact table will be read once. Then, the total number of page-I/Os is

\[
\left\lceil \frac{|G|}{w} \right\rceil \cdot \frac{|E|}{p}
\]  

(4.17)

SIMULATION In the following, the performance of bitmap-based algorithms is compared with that of hash-based algorithms by simulation. The results of this subsection only serve as a quick estimate of how bitmap-based algorithms perform compared to traditional hash-based methods. Many optimization potentials are not considered in the simulation, such as rearranging the sequence of groupsets to minimize the processing costs, query-rewrites, etc. Nevertheless, the results here motivate and encourage the implementation of a bitmap-enabled query processor/optimizer. In Chapter 5, the performance will be measured in running systems.

The simulation is performed as follows. As for bitmap-based algorithms, test data are generated by different random processes. Both tuple-level and page-level bitmap indexes are built on the generated data. Using bitmap-based algorithms introduced above to simulate the processing of grouping and aggregation, the number of page-hits are counted. For the hash-based algorithm (hybrid hashing), the cost until the merging phase begins is computed, i.e., the processing cost till the response-time, according to [38].

Two data sets are tested in the experiment shown in Figure 4.21. In one set, the grouping attribute is uniformly distributed. In the other set, the grouping attribute is Zipf distributed.
Data in both sets are randomly scattered across the table space, i.e., simulating the continuous
data loading. The figures are interpreted as follows.

The vertical lines depict the costs of query execution using bitmap grouping methods, where
the \(y\)-distances of the starting points (the lowest ones, denoted by \([y]\)) denote the execution
costs until the system can respond, i.e., the response times; and the \(y\)-distances of the ending
points (the upper-most ones, denoted by \([y]\)) denote the total execution costs. All points in
between denote the timepoints when the system responds. For the hash-based method, only
the response times are depicted. Since we are convinced that bitmap-based algorithms are
much more responsive, what we want to know is how much longer bitmap algorithms need to
comeplete compared to the response time of hash methods.

The execution costs are expressed by relative measures. The \(x\)-axis is the fraction of the window
size over the total number of groupsets, i.e., \(\frac{w}{\Omega}\). This number means the fraction of groupsets
which can be processed at one time. The \(y\)-axis is the fraction of the number of page-I/Os over
the total number of pages of the operand table, i.e., \(\frac{\text{# of page hits}}{\Omega(p)}\). This number can be interpreted
as the number of times the operand table will be scanned. For cases where \(x \geq 1\), that means
the grouping results can fit into main memory, bitmap-based algorithms will not read any page
of the operand table more than once for both data sets. That is, \([y] = 1\). Therefore, the curves
are plotted within the range \(0 \leq x \leq 1\) only.

Figure 4.21 shows that for both data sets, the bitmap grouping outperforms the traditional
hash method in terms of response time. This is not surprising, since the hash method is not
optimized for response time. From Figure 4.21(b) we can also see that varying the distribution
of the attribute affects the total execution time, whereas the response time is highly dependent
on which group is processed first. In addition, the larger the value of \(x\), the closer between the
total execution time of bitmap-based methods and the response time of hash-based methods.
This reveals that under such circumstances bitmap-based methods are much more responsive
and, at the same time, quick.

**Simulating Batch-Mode Data Loading** In the above experiment, the attribute values are
generated randomly. That is, the experiment simulates a scenario of continuous data loading.
Note that the distribution of an attribute determines only the population of values in the at-
tribute domain, but not the location of the tuples containing the value within the table space.
As mentioned above, due to the batch-mode nature of data loading, warehouse data are likely
to have some degree of clustering, i.e., tuples with common attribute values are likely to be
stored in neighboring pages instead of being randomly scattered over the whole table space.
Based upon this observation, the clustering factor of a group \(g_i\), denoted by \(C_{g_i}\), is defined by
the fraction of the number of pages which are expected to contain tuples of the group over the
actual number of pages which do contain tuples of the group, as Equation 4.18 defines.

\[
C_{g_i} = \frac{E(g_i)}{\Omega(B_{g_i})} \tag{4.18}
\]

In Figure 4.22(a), two sets of data with the same attribute distribution, but different clustering
factors are generated. Data in one set is randomly scattered over the whole table space simulat-
ing continuous data loading (the smoother curve in Figure 4.22(a)), while data in the other set
is somehow clustered simulating batch-mode data loading.\(^{14}\) The counts of tuples of the same

\(^{14}\)Here it does not mean to explicitly perform clustering on the data, instead the clustering comes from the nature
group at one batch-mode data loading is simulated by the Poisson counting process with varying rate $\lambda$.\textsuperscript{15}

The average clustering factor of the first data set is 1.18484, and the average clustering factor of the second data set is 3.52436. A clustering factor of 1 means the expected number of page-hits equals the actual number of page-hits. Figure 4.22(b) shows the execution costs of bitmap grouping on both data sets. As expected, both the response and the execution time on the clustered data set are much smaller than those on the nonclustered data set. Especially at small values of $x$, i.e., where the memory size is small or the size of the grouping results is large, the performance differences on both data sets are immense.

More interesting to know is how the clustering factor affects the performance of bitmap-based algorithms. In Figure 4.23, the results show that not only the response times of bitmap-based algorithms are much smaller than that of hash-based algorithms, but the total execution times are also smaller than that of the hash-based methods. The reason is that the hash-based methods with pessimistic overflow avoidance partitions the input table before any in-memory hash table is built; this does not take advantage of the fact that data are clustered to some extent.\textsuperscript{16} As for hybrid hashing, if the data blocks of the same group are detached from one another, recursive partitioning is also inevitable; this results in higher I/O costs than the bitmap-based methods.

The results of the above preliminary experiments of bitmap grouping are satisfying. The strength of bitmaps will not be really revealed until query processing is considered as a whole, instead of treating individual query operators separately. This topic will be explored in the next chapter.

**SUMMARY** Query processing is the processing of a sequence of operations that need to be of the batch-mode data loading.

\textsuperscript{15}A stochastic process is said to be a counting process if it represents the total number of “events” that have occurred up to a specific timepoint $t$, denoted by $N(t)$. (Details about Poisson process please cf. Appendix E.1)

\textsuperscript{16}Although bucket tuning and dynamic destaging can improve performance of overflow avoidance hashing, its increasing complexity diminishes its slight improvement in response time.
optimized not only individually, but, more importantly, globally. The introduction of bitmap indexing to Data Warehousing provides opportunities to optimize different query operations individually and globally. In this chapter, query processing and optimization techniques for evaluating individual operators, such as selections, Group Bys, Order Bys and aggregations, have been discussed. With the favorable property of bitmap indexes that the results of index processing are themselves bitmaps and can easily and efficiently work with bitmaps of subsequent query operations, we are able to combine selection bitmaps with grouping bitmaps to perform selection and grouping in the same pass. In addition, bitmap-based algorithms support pipelining without physically sorting or hashing the huge fact tables. The intermediate results are fed automatically in some sorted order that enables subsequent joins, if they are not eliminated through query-rewrite, to be implemented in efficient merge-joins.

In Data Warehousing environments, where systems should be responsive and quick, pipelining subsequent operations saves I/Os from materializing intermediate results and enables producing available results as early as possible. Saving I/Os reduces the execution time, and producing results as early as possible reduces the response time.

Another important improvement to query processing in Data Warehousing is the conditional redundancy of Star-Joins through the introduction of group-set bitmaps. If the key/foreign-key joins between fact tables and dimension tables serve exclusively as the bridges of the tables, then the selections or the Group Bys can be directly or indirectly evaluated on the group-set bitmaps without carrying out the Joins.

In the next chapter, an experimental version of bitSQL will be introduced. It is an experimental version, because only part of the bitmap algorithms introduced so far are implemented, and the query optimizer in bitSQL exploits bitmaps to evaluate all queries, regardless of whether bitmap indexes are suitable for that query type or not. It is not the intention to argue that bitmap indexes perform better than any other indexes or processing methods universally. However, implementing the query optimizer in this way will help us to identify exactly those cases for which bitmaps are well-suited. The knowledge gained will in turn help us to design an optimizer at a higher level, which determines what processing methods should be applied to given query operations.
Chapter 5

Performance Measurements and Analysis

In this chapter, the performance of some running systems is compared, including the experimental version of bitSQL. The test data sets are generated using the enclosed dbgen program of the TPC-H Benchmark [85]. The database schema is as shown in Figure 2.5 in Example 1, which is also adopted from TPC-H with slight simplifications. Tests are carried out on different sizes of data sets.

The goal of the performance analysis is to identify the cases where bitmap-based algorithms perform better than conventional sort- or hash-based methods and to explain why. Equally important, it is to identify those cases where conventional methods perform better than bitmap-based algorithms. This knowledge allows us to design a query optimizer at a higher level.

In order to give a quick feeling of how the small bitSQL Engine performs against some of the commercial DBMSs, some preliminary results on a small test data set of about 250M bytes are first shown. The tests are performed on one dual-CPU Sun Ultra 60 machine with Solaris 7, and on a dual-Pentium II 450MHz PC with Microsoft NT Terminal Server 4. The system configurations of two testbeds are listed in Table 5.1. The commercial database systems used in the experiments are tagged as CDB1, CDB2 and CDB3. It is worth mentioning that bitSQL uses cooked files, while CDB1 uses raw partitions. With raw partitions, CDB1 is able to issue I/Os directly, while bitSQL issues I/Os through file I/O libraries that are less efficient than direct low-level I/Os. Detailed analysis on the test results and other settings of the testing environments, together with the design issues of bitSQL, will be discussed in later sections.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Sun Sparc Workstation</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Solaris 2.7</td>
<td>Windows NT Terminal Server 4.0</td>
</tr>
<tr>
<td>Architecture</td>
<td>sun4u sparc SUNW,Ultra-60</td>
<td>86 AT/AT Compatible</td>
</tr>
<tr>
<td>CPU</td>
<td>2 sparcv9 360 MHz CPUs</td>
<td>2 Intel 450 MHz Pentium II’s</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1 GB</td>
<td>512 MB</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>3 GB</td>
<td>1 GB</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>IBM DNES-318390W</td>
<td>IBM DPSS-318390N</td>
</tr>
<tr>
<td>DBMS</td>
<td>bitSQL</td>
<td>CDB1</td>
</tr>
<tr>
<td>Datafile</td>
<td>unix file system</td>
<td>raw device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NTFS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NTFS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NTFS</td>
</tr>
</tbody>
</table>

Table 5.1: System configurations of testbeds
5.1 Preliminary Results

The preliminary results are analyzed in two phases. In the first one (Section 5.1.1 and 5.1.2), we explore the performance of bitmaps and conventional indexes (B-trees) for two individual query operations — selections and Group Bys (aggregate functions). In the second one (Section 5.1.3), we compare their performance for general queries which involve both selections and Group Bys. All queries involve Star-Joins.

5.1.1 Evaluating selections

In both Chapters 3 and 4, it is claimed that bitmap indexing will do well in selection evaluation, especially in Data Warehousing where selection ranges are from medium to large. In the following, some results of evaluating selection-only queries on the test databases are reported. We vary the selectivities of selections in different queries. The selectivity of a query is defined by the ratio of the number of selected tuples over the cardinality of the base table.

Figure 5.1 illustrates the results of the tests run by bitSQL and CDB1 on Sun Sparc Ultra 60. The time dimension is in log-scale in order to show values at small magnitude more clearly. The lengths of the bars denote the duration of query execution, and the first numbers in the brackets denote the response time, and the second ones denote the query completion time. The first bar in each group represents the response and execution time of bitSQL. (The response times of bitSQL are presented by the lengths of leading grey boxes.) The run times of the commercial database system are marked as CDB1. Those marked as CDB1* and CDB1+ are the execution time of CDB1 with FIRST.ROWS and AVOID.FULL optimizer directives, respectively. With AVOID.FULL, CDB1 will avoid table scans and apply indexes defined on the table(s) to evaluate the query. With FIRST.ROWS, CDB1 will choose a plan that optimizes the process of finding only the first row that satisfies the query. We see that bitSQL outperforms CDB1 in most cases of the selection-only queries (SOQs), regarding both the response and execution time. Since in this part of analysis we are more interested in short response time and in comparing performance of bitmap indexes and B-trees, the optimizer directives, FIRST.ROWS and AVOID.FULL, are applied to force CDB1 to optimize the response time and to use B-trees.

From the charts in Figure 5.1, we see that the optimizer directives do not always result in better performance. For the cases of SOQ 1, 2 and 5, table scans perform even better than applying B-trees to evaluate the query. The reasons are simply the high selectivities and the high overheads of index intersection of B-trees. Let us discuss the queries case by case. SOQ 1 and 2 are defined as follows.

SOQ 1

```
SELECT c.mktsegment, l.qty
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  o.priority IN ('1-URGENT', '2-HIGH')
```

SOQ 2

```
SELECT c.nation_id, l.qty
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  o.year IN ('1993', '1994', '1995')
```

SOQ 1 and 2 involve single selections only. The intention of these experiments is simply to investigate how well bitmap indexes can do without the factor of index intersection, since bitmaps can perform index intersection efficiently by logical operations, whereas index inter-
section by B-trees is carried out by costly matching algorithms exploiting possibly hashing or sorting. Due to the high selectivities of both SOQ 1 and 2 (refer to Table 5.2), by default CDB1 invokes a table scan on `orders` to evaluate the selection conditions on `o.priority` and `o.year` for SOQ 1 and 2, respectively, and a table scan on `customers`, plus an index scan on `lineitems`, followed by hash joins on the three tables. Although indexes are defined on both `o.priority` and `o.year`, applying those indexes to evaluate the selection conditions will not bring query performance. In contrast, as the execution time marked as CDB1\(^+\) (i.e., those invoking AVOID\_FULL optimizer directive on `orders`) for SOQ 1 and 2 show, bringing B-trees into selection evaluation results in much worse performance than simply applying table scans, regarding both the response and execution time.

By invoking the FIRST\_ROWS directive, the response time of both SOQ 1 and 2 reduce drastically down to about 1 second. However, the execution time increases by about a factor of two, comparing with that of default processing (marked as CDB1). This result is understandable. The FIRST\_ROWS directive forces CDB1 to use costly nested-loop joins, since hashed joins, in spite of their better efficiency, require higher initial overheads to build the hash table, and thus result in longer response time.

For these two queries, bitSQL outperforms CDB1 in all ways. bitSQL applies group-set bitmaps on both `o.priority` and `o.year` to evaluate the selection conditions, and the resulting bitmaps are used to retrieve data from `lineitems` directly. As discussed in Section 2.3.5, group-set bitmap indexes eliminate the necessity of Star-Joins. The system responds after the generation of the selection bitmap is completed, and the first result is selected from the fact table according to the bitmap. The projection on attributes of `customers` is carried out by sequential scans on `customers`, instead of by joins. Most fraction of the response time is spent in constructing the selection bitmap.
The results of the above two queries, on one hand, confirm our discussions in former chapters that with medium to high selectivities, the performance degenerates by exploiting B-trees. On the other hand, they justify the introduction of bitmap indexing, which reduces both the response and execution time immensely in these cases. For SOQ 1 and 2, the response time of bitSQL is about 1/5 to 1/6 of the best that CDB1 can do, and the execution time of bitSQL is about 1/3 to 1/4 of the best of CDB1.

Before continuing the discussion of other queries, let us see some minor figures of the two testing DBMSs and some statistics of the queries. The volume of base data are of about 250M bytes. The bitmaps defined in bitSQL consume ca. 64M bytes, and the B-trees defined in CDB1 consume ca. 96M bytes. Both the duration of data loading and index building for bitSQL are 67 seconds and 279 seconds, respectively. Those for CDB1 are 1181 seconds and 920 seconds. Although the time of data loading, or that of index building are not the main concerns, it is worth mentioning that bitmap indexes require less space and time than B-trees. What attributes are indexed and the types of indexes are listed in Table 5.3(b).

<table>
<thead>
<tr>
<th>Query</th>
<th>Selectivity</th>
<th>Response-Execution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOQ 1</td>
<td>40%</td>
<td>0.26-66.33</td>
<td>163.28-219.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.29-566.25</td>
<td>791.7-978.9</td>
</tr>
<tr>
<td>SOQ 2</td>
<td>46%</td>
<td>0.24-75.10</td>
<td>169.5-321.64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4-685.54</td>
<td>865.7-1029.04</td>
</tr>
<tr>
<td>SOQ 3</td>
<td>0.00163%</td>
<td>0.93-1.12</td>
<td>37.89-38.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.68-1.13</td>
<td>0.88-1.33</td>
</tr>
<tr>
<td>SOQ 4</td>
<td>0.00166%</td>
<td>0.93-1.13</td>
<td>32.72-33.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31.27-31.97</td>
<td>0.52-0.9</td>
</tr>
<tr>
<td>SOQ 5</td>
<td>8%</td>
<td>0.52-13.99</td>
<td>1.31-71.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.13-71.91</td>
<td>798.75-848.06</td>
</tr>
<tr>
<td>SOQ 6</td>
<td>0.0244%</td>
<td>7.58-7.81</td>
<td>24.23-54.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.01-1.22</td>
<td>805.63-832.49</td>
</tr>
</tbody>
</table>

Table 5.2: Statistics of the testing queries

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Space Requirements</th>
<th>Time Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data</td>
<td>Indexes</td>
</tr>
<tr>
<td>bitSQL</td>
<td>250MB</td>
<td>64MB</td>
</tr>
<tr>
<td>CDB1</td>
<td>250MB</td>
<td>96MB</td>
</tr>
</tbody>
</table>

(a) Space and time requirements of the test databases

<table>
<thead>
<tr>
<th>bitSQL</th>
<th>Indexed attributes</th>
<th>CDB1</th>
<th>Indexed attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lineitems.part_id</td>
<td>B-tree</td>
<td>lineitems.order_id</td>
</tr>
<tr>
<td></td>
<td>lineitems.supp_id</td>
<td>B-tree</td>
<td>lineitems.supp_id</td>
</tr>
<tr>
<td></td>
<td>lineitems.order_id</td>
<td>B-tree</td>
<td>lineitems.part_id</td>
</tr>
<tr>
<td></td>
<td>parts.brand</td>
<td>B-tree</td>
<td>parts.type</td>
</tr>
<tr>
<td></td>
<td>parts.type</td>
<td>B-tree</td>
<td>parts.brand</td>
</tr>
<tr>
<td></td>
<td>suppliers.nation_id</td>
<td>B-tree</td>
<td>suppliers.nation_id</td>
</tr>
<tr>
<td></td>
<td>orders.priority</td>
<td>B-tree</td>
<td>orders.priority</td>
</tr>
<tr>
<td></td>
<td>orders.orderstatus</td>
<td>B-tree</td>
<td>orders.orderstatus</td>
</tr>
<tr>
<td></td>
<td>orders.orderstatus</td>
<td>B-tree</td>
<td>orders.orderstatus</td>
</tr>
<tr>
<td></td>
<td>orders.cust_id</td>
<td>B-tree</td>
<td>orders.cust_id</td>
</tr>
<tr>
<td></td>
<td>orders.year</td>
<td>B-tree</td>
<td>orders.year</td>
</tr>
<tr>
<td></td>
<td>orders.month</td>
<td>B-tree</td>
<td>orders.month</td>
</tr>
<tr>
<td></td>
<td>orders.day</td>
<td>B-tree</td>
<td>orders.day</td>
</tr>
</tbody>
</table>

(b) Indexes defined in the test databases

Table 5.3: Statistics of test databases and indexed defined in both bitSQL and CDB1
In the following, cases at the other end of the selectivity spectrum are discussed. We compare the performance of bitmaps and B-trees in very selective queries in SOQ 3 and 4, with and without the factor of the join operator. They are defined as follows.

**SOQ 3**

```sql
SELECT l.part_id, o.order_id
FROM orders o, lineitems l
WHERE{o.order_id} = '{000064}'
```

**SOQ 4**

```sql
SELECT l.order_id, l.part_id
FROM lineitems l
WHERE l.order_id = '000064'
```

The selectivities of both queries are 0.00163%. Let us look at the results first. For SOQ 3, bitSQL is slightly behind the best time of CDB1. It is surprising, not that it is slower, but that is only “slightly” slower.

Both **FIRST_ROWS** and **AVOID_FULL** directives will cause CDB1 to apply the B-tree on `order_id` to retrieve qualified tuples of `lineitems` which are to be joined to `orders`. With very low selectivities, bitmap scans have higher start-up costs than B-tree traversals, since the whole bitmap needs to be read, while the number of nodes visited in a B-tree equals the height of the tree. In our test database, since an encoded bitmap index on `order_id` is defined, evaluating the single-value selection on `order_id` will need to read all the bitmaps. As a result, using B-trees in this case leads to a better response time than using bitmaps. As for the execution time, the query execution with **FIRST_ROWS** results in shorter execution time than the one with **AVOID_FULL**, exceptionally but not surprisingly. If the result set is small, the initial overheads of the hash join will overwhelm its benefits, as it is in this case. The nested-loop join invoked by the **FIRST_ROWS** directive performs better than the dynamic hash join invoked by **AVOID_FULL** directive this time. Comparing with bitSQL, bitSQL provides even shorter execution time than CDB1. It is because, as mentioned above, with the group-set bitmaps of `order_id` on `lineitems`, SOQ 3 is evaluated without carrying out the Join operation. In cases of very selective queries where B-trees are well-suited, like SOQ 3 and 4, bitmaps still provide an acceptable performance, comparing with B-trees. It is a satisfying result.

The default processing of CDB1 in this case is not discussed any further, since CDB1 applies sequential scans on both `lineitems` and `orders`, followed by a dynamic hash join. It is not reasonable to compare the performance of bitSQL to this case, even if it leads to the advantages of bitSQL.

In SOQ 4, SOQ 3 is rewritten to omit the join operation. bitSQL treats both the queries identically. As we see in Figure 5.1, the execution times of SOQ 3 and 4 by bitSQL are almost the same. The idea behind SOQ 4 is to compare the performance of bitmaps and to that of B-trees without the factor of joins. The chart for CDB1 in SOQ 4 reflects the effect of skipping the joins. Both the response and execution time are smaller than that of bitSQL. Nevertheless, the performance of bitSQL is still comparable to that of CDB1.

The cases of CDB1 and CDB1* of SOQ 4 will not be discussed any further, either. For lack of precise statistical information about the underlying data, CDB1 does not reasonably optimize the queries. However, one point worth mentioning here is, the implementation of bitSQL for the time being does not yet apply statistics of data to perform query optimization, either.

---

1 It is due to the bias in the statistical information of the underlying data. Most commercial systems provide the `UPDATE STATISTICS` statement to refresh the statistics. After that, the query optimizer, which chooses execution plans according to the statistics, will be able to do the optimization more precisely.
The last two queries in this part of analysis involve multiple selections, and SOQ 6 involves a further selection on non-indexed attribute. They are defined as follows:

\[
\text{SOQ 5} \quad \begin{align*}
\text{SELECT} & \quad \text{o.order	extunderscore id, lqty} \\
\text{FROM} & \quad \text{orders, lineitems} \\
\text{WHERE} & \quad \text{o.order	extunderscore id = l.order	extunderscore id AND} \\
& \quad \text{o.orderstatus = 'F' AND} \\
& \quad \text{o.year IN (\text{\textquoteleft1994\textquoteright, \textquoteleft1996\textquoteright) AND} \\
& \quad \text{o.month IN (\text{\textquoteleft01\textquoteright, \textquoteleft02\textquoteright, \textquoteleft03\textquoteright, \textquoteleft10\textquoteright, \textquoteleft11\textquoteright, \textquoteleft12\textquoteright)} \\
& \quad \text{o.shippriority IN (\text{\textquoteleftT\textquoteright, \textquoteleftw\textquoteright, \textquoteleftg\textquoteright) }
\end{align*}
\]

\[
\text{SOQ 6} \quad \begin{align*}
\text{SELECT} & \quad \text{o.order	extunderscore id, lqty} \\
\text{FROM} & \quad \text{orders, lineitems} \\
\text{WHERE} & \quad \text{o.order	extunderscore id = l.order	extunderscore id AND} \\
& \quad \text{o.orderstatus = 'F' AND} \\
& \quad \text{o.year IN (\text{\textquoteleft1994\textquoteright, \textquoteleft1996\textquoteright) AND} \\
& \quad \text{o.month IN (\text{\textquoteleft01\textquoteright, \textquoteleft02\textquoteright, \textquoteleft03\textquoteright, \textquoteleft10\textquoteright, \textquoteleft11\textquoteright, \textquoteleft12\textquoteright) \\
& \quad \text{o.shippriority IN (\text{\textquoteleftT\textquoteright, \textquoteleftw\textquoteright, \textquoteleftg\textquoteright)} 
\end{align*}
\]

By these two queries, the performance of index intersection by both bitmaps and B-trees is tested. As expected, bitSQL outperforms CDB1 for all runs of SOQ 5. The response time of bitSQL is about \( \frac{1}{2} \) of the best of CDB1, and its execution time is only \( \frac{3}{20} \) of the best that CDB1 can achieve. As a matter of fact, none of the runs of SOQ 5 by CDB1 apply B-tree intersections. For the runs marked as CDB1 and CDB1\(^+\), all the selections on \text{orders} are evaluated by a sequential scan on \text{orders}. The runs marked as CDB1\(^+\) apply the B-tree on \text{orders.month} to select the qualified tuples, and the rest of selections are evaluated on the fly of data retrieval.

In order to investigate the performance of index intersection by B-trees, directives, \text{FIRST\_ROWS}, \text{AVOID\_FULL(l)} and \text{AVOID\_FULL(q)}, are applied to SOQ 5. It causes CDB1 to apply B-tree on all selection conditions. However, it results in an unacceptable query performance. After running for over 1000 minutes (over 16 hours), we cancelled the task.\(^2\) The selectivities of individual selections in SOQ 5 are listed in Table 5.4.

<table>
<thead>
<tr>
<th>condition</th>
<th>selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{orderstatus = 'F'}</td>
<td>48.6%</td>
</tr>
<tr>
<td>\text{o.year IN (\text{\textquoteleft1994\textquoteright, \textquoteleft1996\textquoteright)}</td>
<td>30.7%</td>
</tr>
<tr>
<td>\text{month IN (\text{\textquoteleft01\textquoteright, \textquoteleft02\textquoteright, \textquoteleft03\textquoteright, \textquoteleft10\textquoteright, \textquoteleft11\textquoteright, \textquoteleft12\textquoteright)}</td>
<td>49.1%</td>
</tr>
<tr>
<td>\text{shippriority IN (\text{\textquoteleftT\textquoteright, \textquoteleftw\textquoteright, \textquoteleftg\textquoteright)}</td>
<td>0.293%</td>
</tr>
</tbody>
</table>

\[\text{Table 5.4: Selectivities of individual selections}\]

Comparing the performance of SOQ 5 and 6 by bitSQL, we see that the response time increases from 0.52 second to 7.58 seconds. The reason is that the current implementation of bitSQL rewrites all selection conditions on non-indexed attribute(s) to their dimension key(s). In the example of SOQ 6, selections on \text{shippriority} is rewritten to ones on \text{o.order.id}. Then, the selection bitmap for the condition — \(o\text{.shippriority IN (\text{\textquoteleftT\textquoteright, \textquoteleftw\textquoteright, \textquoteleftg\textquoteright)}\) — is built by bitmaps of \text{o.order.id} on the fly. (Details of query rewrites please refer to Section 4.1.2.) After the revision, the Star-Join can be omitted, and the whole query is transformed to a single selection on \text{lineitems} by the selection bitmap. However, this is done with the tradeoff of longer response time. Especially, in this example where the selection condition on \text{shippriority} is very selective, the cost of the subsequent join operation is therefore very limited. As the results show, the cost of rewriting the query is not paid back by the saving of omitting the join operation. In this case, bitSQL overdid the optimization.

As for the performance of CDB1, with \text{FIRST\_ROWS} directive CDB1 performs a sequential scan on \text{orders} to retrieve qualified tuples, followed by a nested-loop join to \text{lineitems}. This produces

\(^2\)The runs of CDB1\(^+\) for SOQ 5 are exceptionally issued with \text{FIRST\_ROWS} and \text{FULL(q)}. Otherwise, without \text{FULL(q)}, it results in an execution plan that tries to do index intersections on all selection conditions, which as stated above results in an unacceptable execution time.
the best performance of CDB1. Other runs without any optimizer directive (marked as CDB1),
or with **AVOIDFULL** (marked as CDB1⁺) perform worse than **bitSQL**.

In the above experiments, queries are varied from single selection to multiple selections and
from low selectivities to high selectivities. We also explore the cases of rewriting selections on
non-indexed attributes to dimension key(s) using bitmap indexes. In general, the performance
of bitmap indexing is very satisfying. For queries with medium to high selectivities (SOQ 1,
2 and 5), bitmap indexes dominate over B-tree in both query response and execution time.
For queries with very low selectivities where B-trees are well-suited, such as SOQ 3 and 4,
bitmap indexes still provide a comparable performance to that of B-trees. Moreover, in the case
involving Star-Joins (SOQ 3), bitmap indexes even perform better than B-trees, with respect to
query execution time.

Concerning the issues of query rewrites using bitmaps, a lesson we learn from the experiments
is that there is a need of a higher level optimizer which exploits the statistics, e.g., the selectivi-
ties, of underlying data to decide whether a query revision is profitable or not. It should avoid
the overdo of the query optimization, as the current implementation of **bitSQL** did for SOQ 6.

As one of the two post-experiments, a simple group-set bitmap index on `lineitems.order_id`
using **orders.shippriority** is defined. It takes 26.24 seconds to build the index, and the bitmaps
require 8.9M bytes. Now, we rerun SOQ 6 on **bitSQL**. The system responds in 0.66 second, and
the query is completed in 0.89 second. The query response time of SOQ 6 after defining the
bitmaps is about 11 times quicker than before, and the query completion time is only 1/10 of
the original one. The result also outperforms CDB1. Although a B-tree on **orders.shippriority**
is also defined in CDB1, we find out that its performance does not have a significant change.
The reason is simply that CDB1 issues a sequential scan on **orders** to evaluate multiple selection
conditions. Indeed, table scans perform better than B-tree intersections in this case. In practice,
if the attribute in question is a frequently asked one, the above result should be convincing
enough to add new bitmap indexes at run time. The return on investments is quite high in this
example. There is a saving of about 7 seconds in query processing which is already about 1/4
of the index building time.

Another post-experiment is to test whether the query processors of different DBMSs are able to
optimize the selection evaluation, if it retrieves an empty set. For the following example,

**SQL 11**

```
SELECT l.order_id, l.suppl_id, l.price 
FROM lineitems l, suppliers s, orders o 
WHERE l.suppl_id=s.suppl_id 
  AND o.order_id=l.order_id 
  AND s.name='Smith' 
  AND o.priority='HIGH'
```

evaluating SQL 11 on the test databases produces an empty set, since there is no supplier named
Smith. **bitSQL** takes 0.005 second to respond, while CDB1 takes over 260 seconds.³ Running
the same test on the PC platform, **bitSQL** takes 0.25 second to respond, while CDB2 takes 0.9
second, and CDB3 takes over 300 seconds.

³With **FIRST_ROWS** directive, CDB1 responds in 0.6 seconds.
5.1.2 Evaluating Group Bys and aggregates

In the following the results of evaluating grouping queries are discussed. In Section 4.3, some bitmap-based algorithms for evaluating Group Bys and aggregates are introduced [15]. Although the results of the simulation in Section 4.3.5 are quite positive for bitmap-based methods, it is still desirable to compare their performance with real-world systems to confirm our claims.

Five queries that group data of joined tables by attributes of different cardinalities are designed. The queries consist of joins, Group By, aggregate functions and Order Bys. No selection is issued in these queries, since we want to compare the performance of the bitmap-based grouping algorithms with that of conventional algorithms which use either sorting, hashing or index paths (B-trees), without the factor of selections. Especially, the results in last section show that bitmaps outperform B-trees in most cases of our testing selection queries, it is necessary to assure that if the bitmap-based grouping algorithms perform better than conventional methods, there is no influence from the selection operations. We will come back to the general scenario later where queries involve joins, Group Bys, aggregate functions and selections.

First, let us look at two simple queries, defined as follows.

**GQ 1**

```
SELECT s.nation_id, SUM(l.tax) 
FROM lineitems l, suppliers s 
WHERE l.sup_id=s.sup_id 
GROUP BY s.nation_id 
ORDER BY s.nation_id
```

**GQ 2**

```
SELECT s.sup_id, SUM(l.discount) 
FROM lineitems l, suppliers s 
WHERE l.sup_id=s.sup_id 
GROUP BY s.sup_id 
ORDER BY s.sup_id
```

GQ 1 and 2 compute summaries of tax and discount from lineitems for each nation and supplier, respectively. As described in Section 4.1.2 and 4.3, grouping by dimension attribute(s) which are not indexed will be rewritten to grouping by their dimension key(s). The grouping operator is switched from the single-value Group By to the multi-value one. To be able to rewrite such queries, it is required that bitmap indexes are defined on the dimension keys. If, however, the grouping attributes are indexed, the bitmaps can be directly used without rewriting the queries.

For the current implementation of bitSQL, regardless of whether the grouping attributes are indexed or not, query rewrite is activated by the query processor. For the example in GQ 1, although the attribute nation_id is indexed, the grouping by nation_id is revised to grouping by subsets of sup_id. The bitmaps defined on supp_id are used to construct the grouping bitmaps, one for each nation. After grouping bitmaps are ready, taxes of lineitems belonging to the same nation is aggregated. The accesses of data pages are scheduled by the grouping bitmaps (as defined in Algorithm 6).

The intention of this implementation is to explore the performance of bitmap-based algorithms under the requirement that all grouping bitmaps, other than grouping on dimension key(s), must be constructed at run-time. Since the rewrites of grouping conditions eliminate the Star-Joins, it is desirable to compare the cost of generating the grouping bitmaps with the cost

---

4 If the referential integrity between l.sup_id and s.sup_id is satisfied, the join between lineitems and suppliers in GQ 2 is redundant. However, we still like to keep the join, since for typical cases, there are very likely to have some selection conditions on the dimension table. In such cases, the join is needed. Besides, we only want to exclude the influences of selections on query processing. Joins that are generally required should stay.
of joins by conventional methods.

For GQ 1 and 2, bitmap algorithm outperforms conventional ones, as Figure 5.2 shows. The bitmap indexes defined on the test databases are listed in Table 5.3(b). As described above, instead of joining the two operand tables in GQ 1, the grouping on nation_id is rewritten to the one on supp_id, and bitmaps of those supp_ids which belong to the same nation are ORed to produce the grouping bitmap.\(^5\) Once the bitmaps are finished, they are transformed to their page-level version and are used to schedule the direct page-accesses on lineitems, followed by the aggregate function. bitSQL responds in 24.7 seconds and completes the evaluation of GQ 1 in 55.19 seconds. In the experiments, the duration of bitmap generation, including transforming tuple-level bitmaps to the page-level ones, is also measured. For GQ 1, bitSQL spends the first 8.9 seconds to construct the grouping bitmaps.

As for the performance of CDB1, the runs with sequential table scans followed by hashed-based join and grouping (marked as CDB1) have the shortest execution time among different runs of CDB1. The runs with FIRST_ROWS and AVOID_FULL directives cause CDB1 to use index scan on lineitems and/or suppliers, which degenerates the performance. Note that here, the runs tagged as CDB1 are default processing by CDB1, with no optimizer directive. Therefore, the query plans generated for these runs differ from query to query. We will reveal their query plans case by case.

\[\text{\begin{tabular}{|c|c|c|c|}
\hline
GQ 1 & & & \\
\hline
\hline
\text{10} & \text{100} & \text{1000} & \\
\hline
\text{sec. SQL} & \text{sec. CDB1} & \text{sec. CDB1*} & \\
\hline
\hline
\end{tabular}}\]

\[\text{\begin{tabular}{|c|c|c|c|}
\hline
GQ 2 & & & \\
\hline
\hline
\text{30.64} & \text{89.95} & \text{sec. SQL} & \\
\hline
\text{169.48} & \text{sec. CDB1} & \text{sec. CDB1*} & \\
\hline
\hline
\end{tabular}}\]

\[\text{\begin{tabular}{|c|c|c|c|}
\hline
GQ 3 & & & \\
\hline
\hline
\text{145.01} & \text{78.90} & \text{sec. SQL} & \\
\hline
\text{137.23} & \text{sec. CDB1} & \text{sec. CDB1*} & \\
\hline
\hline
\end{tabular}}\]

\[\text{\begin{tabular}{|c|c|c|c|}
\hline
GQ 4 & & & \\
\hline
\hline
\text{3845.35} & \text{sec. SQL} & \text{sec. CDB1*} & \\
\hline
\text{3208.86} & \text{sec. CDB1*} & \text{sec. CDB1*} & \\
\hline
\hline
\end{tabular}}\]

\[\text{\begin{tabular}{|c|c|c|c|}
\hline
GQ 5 & & & \\
\hline
\hline
\text{1816.96} & \text{sec. CDB1} & \text{sec. CDB1*} & \\
\hline
\text{874.25} & \text{sec. CDB1*} & \text{sec. CDB1*} & \\
\hline
\text{922.88} & \text{sec. CDB1*} & \text{sec. CDB1*} & \\
\hline
\hline
\end{tabular}}\]

Figure 5.2: Response time of bitSQL and CDB1 for evaluating Group Bys and aggregates

As for GQ 2, although the dimension key supp_id is indexed, the grouping bitmaps (including their page-level counterparts) are built on the fly, since the bitmap index defined on supp_id is an encoded one, and in the current version of bitSQL simple grouping bitmaps are required. The

\(^5\)Precisely speaking, since the bitmaps defined on supp_id are encoded bitmaps, the retrieval Boolean functions for those supp_ids which belong to the same nation are ORed to form the retrieval Boolean function for that nation. After reducing the Boolean function, it is used to construct the grouping bitmap.
response time of bitSQL for GQ 2 is 30.64 seconds, and the total execution time is 89.95 seconds. The first 22.05 seconds are used to construct the grouping bitmaps. The performance of CDB1 for GQ 2 is similar to that for GQ 1. The best performance is also achieved by sequential scans of both operand tables, followed by hash join and grouping (the runs tagged as CDB1 and CDB1+). Bringing index scans into query processing for GQ 2 (the runs tagged as CDB1+) degenerates the performance drastically.

The measures of the queries, including the sizes of result sets, are listed in Table 5.5. For runs by CDB1, only the query completion times are presented since either the query response times have unnoticeable differences to the completion times (the cases with small result sets), or the fraction of the response time to the completion time is nearly one (the cases with larger result sets).

<table>
<thead>
<tr>
<th>Query</th>
<th>Result set (# of rows)</th>
<th>bitSQL</th>
<th>CDB1</th>
<th>CDB1⁺</th>
<th>CDB1⁺⁺</th>
</tr>
</thead>
<tbody>
<tr>
<td>GQ 1</td>
<td>25</td>
<td>24.70-35.19</td>
<td>184.15</td>
<td>1816.96</td>
<td>874.25</td>
</tr>
<tr>
<td>GQ 2</td>
<td>100</td>
<td>30.64-89.95</td>
<td>169.48</td>
<td>163.3</td>
<td>922.88</td>
</tr>
<tr>
<td>GQ 3</td>
<td>25</td>
<td>462.54-847.33</td>
<td>374.34</td>
<td>308.07</td>
<td>304.73</td>
</tr>
<tr>
<td>GQ 4</td>
<td>7996</td>
<td>217.26-1158.98</td>
<td>3396.80</td>
<td>3402.31</td>
<td>3396.85</td>
</tr>
<tr>
<td>GQ 5</td>
<td>2403</td>
<td>145.01-789.07</td>
<td>3137.23</td>
<td>3423.35</td>
<td>3208.56</td>
</tr>
</tbody>
</table>

Remark: After UPDATE STATISTICS and by explicitly specifying FULL directives, the queries GQ 3, 4 and 5 are resubmitted to CDB1, and the performance are listed in the column CDB1++.

Table 5.5: Statistics of the testing queries

Let us move on the discussion to GQ 3, which is defined as follows.

**GQ 3**

```
SELECT p.brand, SUM(l_qty)
FROM lineitems l, parts p
WHERE l.part_id=p.part_id
GROUP BY p.brand
ORDER BY p.brand
```

The result of GQ 3 attracts our attention to reexamine the current implementation of bitSQL, and the following analysis provides a direction for future improvements. Similar to the procedures described above, bitSQL rewrites the grouping by brand, regardless of the bitmap index defined on it, to the grouping by subsets of part_id. The grouping bitmaps, together with their page-level versions, are generated using bitmaps of part_id. bitSQL responds in 462.54 seconds, and the first 156.73 seconds are used for bitmap generation. The query is completed in 847.33 seconds. Comparing the results of both GQ 1 and 3, there is a remarkable difference between the performance of both queries, although they have the same size of result sets. Of course, different operand tables with different data distribution and data cardinalities are involved in GQ 1 and 3. The measures for the two queries are supposed to be different. However, the duration of bitmap generation for GQ 3 is noticeably over 17 times longer than that for GQ 1, and both queries require generating 25 grouping bitmaps.

In order to understand where the loads go, the information about CPU utilization, numbers of disk accesses and numbers of page-faults during the execution of the two queries by bitSQL is collected and the data is illustrated in Figure 5.3. We see that bitSQL is unexpectedly CPU-bound, instead of I/O-bound. At the bitmap generation phase (at the beginning of the curves), the I/O loads are quite heavy. However, thereafter I/O has not been a bottleneck of the performance. On the other hand, CPU is almost 100% busy. Note that the tests are run on a dual-CPU
machine, and 50% of CPU load means in our case (since bitSQL runs as a single process, no parallelism or multithreading are implemented) that one of the CPU is fully loaded.\textsuperscript{6}

After reviewing the program, we find out that there are intensive in-memory lookups into the hash-table for multi-value grouping and aggregation through pointers which should have been optimized through, for example, \textit{sorted}, \textit{continuous} data types like arrays to eliminate the expensive sequential pointer-traversals. Especially, for each tuple of the fact table, the in-memory lookup is performed once, and the lengths of the linked lists depend on the cardinality of the dimension table on which the grouping condition is given, not on the cardinality of the grouping attribute. This is also the main reason why GQ 3 takes over 15 times of the processing time of GQ 1, since the cardinality of \texttt{parts} is 20 times of the cardinality of \texttt{suppliers}.

![Resource Utilization Analysis](image)

(a) Resource utilization of GQ 1

![Resource Utilization Analysis](image)

(b) Resource utilization of GQ 3

Figure 5.3: Resource utilization analysis for bitSQL

CDB1 executes GQ 3 as follows. All runs tagged as CDB1, CDB1* and CDB1$^+$ employ index scans on both tables, followed by a nested-loop join. They result in poor performance. After updating the statistics, the query with explicit \texttt{FULL(lineitems)} and \texttt{FULL(parts)} directives is issued again. It causes CDB1 to use table scans and hash joins. The execution time is reduced down to 343.43 seconds, which outperforms bitSQL.

Note that again, the goal of the experiments here is to compare the performance of bitmap-based Group By algorithms with conventional methods, which may apply index paths (B-trees), or apply sorting/hashing to evaluate Group Bys. CDB1 is used as a testbed to evaluate the same set of queries by conventional methods, just to save us time from implementing the conventional methods again in bitSQL. The knowledge about how and why some algorithms outperform the others is more valuable, since it will be applied to build a query optimizer at a higher level. Besides, we are not to argue that bitmap-based algorithms perform universally well. As a future version, bitSQL should also include conventional Group By algorithms and a strategic query optimizer above the tactical optimizer.

However, there are still great opportunities to improve the bitmap-based algorithms. As de-

\textsuperscript{6}Since there are still other system (possibly also user) processes running on the same machine, the CPU utilization is sometimes over 50\%. With another performance monitor tool, we find out that bitSQL burdens one of the CPUs 100\% for most of the time.
scribed above, the bitmap generating phase takes 156.73 seconds which can originally be saved, and from Figure 5.3(b) we also see that after about 250 seconds I/Os are not the main factor of poor performance. The first possible improvement is to use available bitmap indexes. As in this case, the attribute brand is bitmap indexed, but the bitmaps are not used in the current version of bitSQL, since (as stated at the beginning of this section) we want to explore the costs of dynamically generating grouping bitmaps. Another possible improvement is to replace unsorted linked-list structures, which are used for the aggregation hash-table, with sorted arrays. This will be further discussed in Section 5.2.2, as we talk about the system internals of bitSQL. As a summary to the discussion of GQ 3, let us look at the resource utilization during its execution in CDB1, just as a comparison with Figure 5.3(b). Figure 5.4(a) depicts the curves for runs tagged as CDB1, and Figure 5.4(b) illustrates the curves for runs tagged as CDB1++. As expected, the performance bottleneck of CDB1 is the I/O.

![Resource Utilization Analysis](image)

(a) Resource utilization of runs tagged as CDB1  
(b) Resource utilization of runs tagged as CDB1++

Figure 5.4: Resource utilization analysis of GQ 3 for CDB1

The last two Group By queries are defined as follows.

**GQ 4**

```sql
SELECT s.supp_id, p.part_id, SUM(l.qty) 
FROM lineitems l, suppliers s, parts p 
WHERE l.supp_id=s.supp_id AND l.part_id=p.part_id 
GROUP BY s.supp_id, p.part_id 
ORDER BY s.supp_id, p.part_id
```

**GQ 5**

```sql
SELECT s.supp_id, p.brand, SUM(l.tax) 
FROM lineitems l, suppliers s, parts p 
WHERE l.supp_id=s.supp_id AND l.part_id=p.part_id 
GROUP BY s.supp_id, p.brand 
ORDER BY s.supp_id, p.brand
```

They are designed to help us analyze the poor performance of the last example due to overloads of CPU in bitSQL. The results confirm our above assumption about the inefficient data structures for the hash tables used in bitSQL. Since the analysis is too specific to the current implementation, and the anomalies due to the current implementation do not invalidate the correctness and feasibility of the proposed methods, it will be described briefly only. For the time being, the multi-value Group Bys are implemented by a two dimensional linked list for each grouping attribute. The first dimension stores the values of the original grouping attribute, and the second one stores the corresponding value-set of the (dimension) key which functionally determines the original grouping attribute. In order to locate the entry in the hash
table for aggregation, for each tuple in the operand (fact) table all the two dimensional linked lists are sequentially scanned until the considered value is found. As a result, after the phase of bitmap generation, CPU is still fully loaded with these intensive in-memory searches. The resource utilization during the execution of GQ 4 and 5 in bitSQL are shown in Figure 5.5.

![Resource Utilization Analysis](image)

(a) Resource utilization of GQ 4

![Resource Utilization Analysis](image)

(b) Resource utilization of GQ 5

Figure 5.5: Resource utilization of GQ 4 and 5 by bitSQL

However, in spite of the known problem in the current implementation, bitSQL still shows the strength of bitmap-based algorithms against conventional methods in these two queries. As expected, bitSQL not only provides a very short response time (within 218 and 146 seconds for GQ 4 and 5, respectively), but also a comparable execution time comparing with some conventional grouping methods, which are optimized for execution time. The duration of bitmap generation phases for GQ 4 and 5 are 23.75 and 16.93 seconds, respectively.

As for CDB1, all runs tagged as CDB1, CDB1* and CDB1+ employ table scans on the dimension tables and index scans on the fact table, followed by a nested-loop join on parts and lineitems, and then a dynamic hashing for the second join and the aggregation. Like the case in GQ 3, the above query plans result in poor performance. Again, after the statistics is updated, the queries with full directives on all operand table are resubmitted to force CDB1 to employ sequential table scans and dynamic hashing for joins and the aggregation. The performance improves significantly. Till now, we can see that using B-trees for Group Bys and aggregation has never resulted in good performance in our tests in CDB1.

From the above experiments, bitmap-based methods do expectedly provide short response time and comparable query execution time to conventional grouping methods. The results of the experiment conform more or less to the simulation results in Section 4.3.5. Before we move on the discussion to the general scenario where queries may involve any type of query operations, let us discuss one minor experiment concerning the performance of simple bitmap indexes and encoded bitmap indexes.

Considering GQ 1 again, suppose that a simple bitmap index is defined on suppliers.supp_id instead of an encoded bitmap index. It is interesting to know how long it will take to construct the grouping bitmaps for suppliers.nation_id from simple bitmaps instead of from encoded bitmaps.
on suppliers supp.id. In the real world cases, inevitably, there are cases where the grouping bitmaps must be generated at run time, since it is not reasonable to suppose that all attributes are indexed. Therefore, it is desirable to keep the phase of bitmap generation short.

The results of re-running GQ 1 using either simple bitmap or encoded bitmaps of supp.id are listed in Table 5.6. Not surprisingly, the bitmap generation using encoded bitmaps is more efficient than using simple bitmaps, since the time needed for bitmap generation is proportional to the number of source bitmaps which needs to be read, and the number of target bitmaps which needs to be written. In this case, the source bitmaps are the bitmap index on supp.id, and the target bitmaps are the grouping bitmaps for nation.id. The number of target bitmaps in this example is equal to 25, i.e., the number of nations. As a result, the time needed for bitmap generation is simply dependent on the number of source bitmaps. Obviously, encoded bitmap indexes is more space-efficient than simple bitmap indexes by definition. In our example, the run using simple bitmaps takes about 22 seconds longer than the one using encoded bitmaps.

<table>
<thead>
<tr>
<th>Simple bitmaps on supp.id</th>
<th>Encoded bitmaps on supp.id</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitmap generation</td>
<td>[response-execution] time</td>
</tr>
<tr>
<td>31.14 seconds</td>
<td>[46.13-77.12] seconds</td>
</tr>
<tr>
<td></td>
<td>8.9 seconds</td>
</tr>
<tr>
<td></td>
<td>[24.70-55.19] seconds</td>
</tr>
</tbody>
</table>

Table 5.6: Durations of grouping bitmap generation using simple and encoded bitmaps

5.1.3 Evaluating queries involving Group Bys, aggregations and selections

In this subsection, the experimental results of running 27 queries in CDB1, CDB2, CDB3 and bitSQL are presented. The test databases are the same as those used in former subsections, and are setup as follows. The data are generated by the enclosed dbgen program of the TPC-H Benchmark. In order to simulate the overnight data loading nature of Data Warehouses, the data are generated and loaded stepwise. For each run, 32M bytes of data are generated and sorted before loading them into the databases.

The indexes built on bitSQL and CDB1 databases are as described in Section 5.1.1 (listed in Table 5.3). For the test database in CDB2, the results of [41] are applied to define the optimal indexes, as listed in Table 5.7. In [41], Graefe has conducted experiments specific to Microsoft SQL Server using the schema and data generated by the TPC-D Benchmark to find an optimal index design for decision support workloads. The proposed set of indexes should improve both hash-based and sort-based joins in Microsoft SQL Server. Although index design depends also on other factors, such as query types and data distribution, the results are simply adopted to the test databases. Two test runs of all the test queries have been conducted in [41], one on non-indexed data, and the other on indexed data in Microsoft SQL Server. As expected, there are significant performance improvements by the indexes. Since it is not the theme of this work to tune the commercial DBMSs, we apply the results from [41] in a straightforward way, without claiming that the indexes defined in the test database are optimal for our workloads.7 Two kinds of indexes are built on the CDB3 databases. The one are B-trees, and the other one are bitmap indexes supported by CDB3. The set of B-tree indexes defined in the CDB3 test databases are the same as the one defined in CDB2 (listed in Table 5.7), and the set of bitmap indexes are the same as those defined in bitSQL.8

7 The study in [41] relies on the index-tuning wizard of Microsoft SQL Server.
8 Later, we will also apply the set of indexes proposed by [41] to the CDB1 database, and compare the query...
Configurations of both testbeds and the DBMSs are listed in Table 5.1, and the definitions of the 27 queries are enclosed in Appendix E.2. The performances of the test queries by CDB1 and bitSQL on the Sun Sparc machine are presented in Figure 5.6 and 5.7, and the performances of the runs on the PC by CDB3, CDB2 and bitSQL are shown from Figure 5.8 to 5.10. All commercial DBMSs run the queries with the logging disabled.

As a brief summary, for twenty out of the 27 queries, bitSQL provides better response time than CDB1, and for sixteen out of them, bitSQL even provides shorter query execution time than CDB1. In Figure 5.7, we further compare the performance of bitSQL with that of CDB1 with the index set proposed by [41]. We see that the overall performance of CDB1 improves (except Queries 13 and 16, which degrade over 200% and 50%, respectively). For four queries, bitSQL still outperforms CDB1, and for four other queries bitSQL provides shorter response time than CDB1.

Comparing with CDB3, for nine out of the 27 queries, bitSQL beats CDB3 in all respects. Comparing with the query performances on the data set with B-trees, bitSQL outperforms CDB3 in ten queries. Comparing with the one with bitmap indexes, for 22 out of the 27 queries, bitSQL outperforms CDB3.\(^9\)

The performance of CDB2 is shown in Figure 5.10. Only for three queries, bitSQL outperforms CDB2, and for four additional queries, bitSQL provides a shorter response time than CDB2. The performance of CDB2 is noticeably outstanding. To explore the reasons why CDB2 dominates other DBMSs, the system activities and resource utilization are logged during the runs of the queries. As a matter of fact, a brute-force comparison among the systems, even if they are running on the same platform, will mislead the analysis, since one uses more system resources, while another uses less. Therefore, in the following, let us first discuss the utilization of system resources by different DBMSs.

First, the main memory utilization of CDB2, CDB3 and bitSQL during the runs of all 27 queries is illustrated in Figure 5.11(a), (b) and (c), respectively. As we see that CDB2 and CDB3 use upto 300M bytes of memory during the runs, while bitSQL uses only upto 3.5M bytes of memory. The size of the test database is only 250M bytes, *i.e.*, in principle, CDB2 and CDB3 could have cached the whole database in the main memory. Obviously, such comparisons between

---

\(^9\) According to the execution logs, CDB3 does not apply bitmap indexes defined on the data to evaluate the queries.
Figure 5.6: Performance comparison between CDB1 and bitSQL, 250 MB test data
Figure 5.7: Performance comparison between CDB1 with optimal indexes and bitSQL, 250 MB test data
**Figure 5.8:** Performance comparison between CDB3 and bitSQL (continued), 250 MB test data
Figure 5.9: Performance comparison between CDB3 and bitSQL, 250 MB test data

CDB2 and bitSQL, and between CDB3 and bitSQL, are not performed at a fair basis. Nevertheless, under such circumstances, bitSQL still shows the strength of bitmap algorithms and outperforms the commercial DBMSs in some cases.

To give a comparison between the memory usages between the runs on different machines, the memory usages of CDB1 and bitSQL are also logged on the Solaris platform, and the curves are illustrated in Figure 5.11(d). We see that although CDB1 uses more memory than bitSQL, the differences are not so enormous. CDB1 uses up to 15 MB main memory, and bitSQL up to 3 MB memory.

For the current version of bitSQL, it runs as a single, normal user process,\(^{10}\) without multithreading and parallelization. The I/O operations are performed by stdio-library, and no cache management is implemented. There are still many optimization potentials at the implementation level which are not implemented for the time being due to the shortage of time and manpower.

Second, the CPU loads and numbers of I/Os are collected during the runs of CDB1 and bitSQL on the Sun Sparc platform, to see where the loads go. The data are drawn in Figure 5.12. (The curves of CDB1 are drawn for reference purposes, since the performance of bitSQL is the center of interest.) As we have discussed in last subsection, for most of the time the bottleneck of the performance of bitSQL is CPU, instead of I/Os. However, there are still some peaks of disk accesses which are even higher than the curves of CDB1. It reveals that both CPU and I/O loads of bitSQL need to be smoothed, e.g., through multithreading, and/or better cache managements for the reading/writing of bitmaps and other system data, or reusing the bitmaps. In next section, we talk more about the implementation issues.

---

\(^{10}\)Since it runs as a normal user process, its performance is very sensitive to system loads.
Figure 5.10: Performance comparison between CDB2 and bitSQL, 250 MB test data
5.2 Implementation and Experimental Setups

The current implementation of bitSQL is aimed at evaluating the performance of the bitmap-based query processing algorithms for selections, Group Bys and aggregations. Since the existence of bitmaps changes the mechanism of query processing and optimization (as discussed in Chapter 4), a bitmap-enabled SQL engine — bitSQL — is built to perform this task. This experimental version of bitSQL exploits bitmaps to evaluate queries exclusively, i.e., no higher level optimizer is implemented to choose among bitmap-based methods and conventional methods. If bitmap indexes are not defined, they are generated on the fly. The only assumption on the set of indexes is that the foreign keys to dimension tables in the fact table(s) are bitmap

---

11 Usually, an index provides an access path to the desired data, as an alternative to sequential table scans. All query operations, such as selections, joins and Group Bys, access underlying data for their own objectives. Theoretically, all of the query operators can employ indexes to retrieve data. However, in practice, it depends on the costs of using the indexes. Since in this work, we introduce bitmaps to query processing, and bitmaps change the way of query evaluation (simply speaking, the separation of query evaluation on bitmaps and data retrieval), we need to revise the kernel of a query processor, not just simply adding an access path to the query engine. To accomplish our goal of integrating bitmaps into the query engine, we build a bitmap-enabled experimental SQL engine.
Resource Utilization Analysis

CPU utilization in %
Disk accesses in % over 320 pages

Percentage (%)
Time dimension (seconds)

(a) CDB1

(b) bitSQL

Figure 5.12: Resource utilization of CDB1 and bitSQL

indexed.

5.2.1 Architecture of bitSQL

bitSQL consists of two major system components — a SQL compiler and a SQL engine, as Figure 5.13(b) shows.

The flow of query execution is not much different from the conventional one. The differences are within step 2, 3 and probably 5 in Figure 5.13(a). We describe the flows of query execution in bitSQL, and how the components contribute to query processing by the example of Query 1, which is listed below for ease of reference.
SELECT c.mktsegment, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id=o.cust_id AND
    o.order_id=l.order_id AND
    o.year IN ('1995', '1996') AND
    c.nation_id IN ('01', '02')
GROUP BY c.mktsegment
ORDER BY c.mktsegment

By default, bitSQL accepts users’ input from the standard input device, and while reading in the inputs, the compilation of the SQL statements begins. It begins with syntax checking and content checking. Syntax checking means simply the verification of the correctness of the SQL statements. As for content checking, schema information is consulted to check the existence of query operands, for example, whether the given attribute exists in the given table. After both checks are done, the identified clauses are decomposed into an internal representation, which is further submitted to the SQL engine for execution. The syntax of admitted SQL statements in bitSQL is included in Appendix E.3 for references. Worth mentioning here is, only statements relevant with data loading, index definition and querying are accepted by bitSQL for the time being. Regarding index definition, naturally, statements for defining bitmap indexes are introduced. Both simple and encoded bitmap indexes are allowed, however, bit-sliced indexes are not implemented in the current version.

Once the compilation phase ends, the system either rejects the query, or accepts the query and moves on to the next stage — query rewriting. Query rewriting involves two subtasks. The first is to rewrite selection conditions, and the second is to rewrite the Group By clause. As discussed in Section 4.1.2, the philosophy of the query rewrites in bitSQL is to revise the query in such a way that Star-Joins between fact table(s) and dimension tables can be omitted. This can be achieved by applying functional dependencies between key and non-key attributes.

The system catalog is consulted at this stage again, to check whether the selecting and grouping attributes are indexed. Unlike conventional query processing, where usually only the index of the most selective attribute is chosen, all the bitmap indexes will be used. If a selecting attribute is not indexed and it is an attribute in a dimension table, the selection condition is rewritten as follows. Suppose that a selection of the form — \( A \ op \ V \), where \( A \) denotes an attribute, \( op \) is one of the relation operators (\( =, \ IN \)), \(^{12}\) and \( V \) denotes an operand set. The selecting attribute is changed to the key attribute of its residing table, and the operand set is rewritten to a subset of key values such that for any values in \( V \) all the key values which functionally determine it are included in the revised operand set. That is, \( A \ op \ V \) is revised to

\[
D \ IN \ V',
\]

where \( D \) is the key attribute of the table which contains \( A \), and

\[
\forall d \in D, \text{ if } \exists v \in V, \exists d \rightarrow v \\
\implies \quad d \in V'
\]

\( (d \rightarrow v \text{ means } d \text{ functionally determines } v. )\)

\(^{12}\)The same principle with minor modifications can be applied to include all other relation operators, such as \( <, \), \( >, \), \( \leq \), \( \geq \) and \( \neq \).
Taking the example of Query 1, the following selection — `c.nation_id IN ('01', '02')` — on the snow-flaked dimension table, customers, will be first rewritten to

\[ o.cust_id IN C, \]

where \( C \) is the set of the customers' ids who come from the nations numbered '01', or '02'. Since the goal of query rewrites is to omit Star-Joins to dimension table(s), the above rewriting procedure is applied recursively until the dimension key attribute, which is also a foreign key attribute in the fact table, is reached. In this example, since \( cust_id \) is not a foreign key on lineitems, we have to further rewrite the revised selection to

\[ l.order_id IN O, \]

where \( O \) is the set of orders' ids which are submitted by customers whose ids are members of \( C \), i.e., those customers who come from nations numbered '01', or '02'. The rewriting is done, since \( order_id \) is a foreign key of lineitems. The above revised condition can now be evaluated by \( ORing \) all the bitmaps of \( order_id \)s in the set \( O \), if the index defined on \( order_id \) is a simple bitmap index. In our test database, an encoded bitmap index is defined on \( order_id \). Then, the selection bitmap for the above revised condition can be obtained by \( ORing \) all the retrieval Boolean functions of the \( order_id \)s in the set \( O \). The Boolean function is first reduced to its minimal form, before the selection bitmap is materialized.

After all the selection conditions are either revised, or the selecting attributes are indexed, the final selection bitmap can be obtained by \( ANDing \) selection bitmaps for all individual selections. As for selections on the fact table, they are evaluated at the time of data retrieval. At this point in time, if any selection is evaluated as FALSE, it will be reported to the query processor, such that it can take proper action, e.g., for the current implementation, since only conjunctive conditions are accepted, any contradicting condition will terminate the execution and return an empty set as the result.

The procedure of rewriting Group By clause is similar to the one of rewriting selections, since grouping data by some attribute(s) can be treated as a union of selections on each value of the grouping attribute(s). For the example in Query 1, data are to be grouped by mktsegment. The domain of mktsegment is \{ 'AUTOMOBILE', 'BUILDING', 'FURNITURE', 'HOUSEHOLD', 'MACHINERY' \}. The above grouping can be treated as a union of selecting tuples of 'AUTOMOBILE', 'BUILDING', 'FURNITURE', 'HOUSEHOLD' and 'MACHINERY', one group after another. In other words, we want to rewrite selection conditions, like \( c.mktsegment='AUTOMOBILE' \), to

\[ o.order_id IN O_A, \]

where \( O_A \) denotes the set of order ids which are submitted by customers who belong to the automobile market-segment. For each group, one grouping bitmap is generated.

At the end of query rewriting phase, two kinds of bitmaps are ready for subsequent processing: the selection bitmaps and the grouping bitmaps. An important point that needs to be stated here is the group-set reduction by selections. The ideas behind are straightforward. For example, if we have the following selection condition in Query 1,

\[ c.mktsegment IN ('AUTOMOBILE', 'BUILDING'), \]

then the number of groups, or named groupsets, is at most two. At the phase of rewriting Group By clauses, the condition will be used to reduce the number of groupsets, and we only need to generate grouping bitmaps for 'AUTOMOBILE' and 'BUILDING'. 
Another less obvious reduction can be probably achieved at the time of selection revision. The reduction depends on the distribution of underlying data. For example, if at the time of evaluating Query 1 there is no customer from nations ’01’, or ’02’ who belong to the market-segment ’BUILDING’ in our database, then the number of groupsets can be at least reduced by one. Note that, however, such optimization can be overdone. It does not necessarily result in better performance. In the current implementation of bitSQL, the above group-set reduction mechanism have been applied.

From the above discussion, we see that query rewrites are not done without costs. For the time being, each rewriting on selections, or grouping, requires scanning the dimension table(s) once. Bitmap indexes are not applied to dimension tables in the current version. However, for big dimensions, indexes may speed up the searching in dimension tables.

Now, after the selection and grouping bitmaps are generated, we are ready to move on to step 3 of query processing in Figure 5.13(a). At this stage, bitmaps from former steps can be passed to bitmap algorithms to evaluate the query. In bitSQL, bitmap-based selection, grouping and aggregation and some other helper functions for bitmap processing, such as the mapping from bit positions to physical tuple addresses, Boolean function reduction for encoded bitmap indexes, and tuple/page-level bitmap conversion, have been implemented. The bitmap-based selection and grouping algorithms implemented in bitSQL have been discussed in Chapter 4.

After results are retrieved from the database and before feeding them to users, there is probably some post-processing needed. Typical post-processing is projection expansion. It means that the projection of dimension attribute(s), which are not involved in other query operations, are postponed to the end of query processing by simply expanding the attribute values to the result set. Another example of projection expansion occurs when the standard single-value Group Bys are replaced with the multi-value Group Bys. The attribute values of the original grouping attribute(s) are expanded over the result set. For the above example, Query 1 groups data by mktsegment, which is rewritten to grouping by subsets of order_ids. After the grouping and aggregation is done, the corresponding values of mktsegment have to be inserted into the result set.

5.2.2 More about the system internals of bitSQL

In the following, some more implementation details of bitSQL will be discussed. First of all, in Algorithm 6 the set of grouping bitmaps are given as parameters, one for each group. That is, if the grouping condition is given on multiple attributes, the number of bitmaps is the product of cardinalities of all attributes. For example, given the following Group By clause — “Group By c.mktsegment, o.clerk”, if the cardinalities for c.mktsegment and o.clerk are 5 and 100, respectively, then there are at most $5 \times 100$ groups in the result set. In other words, if we use one bitmap for one group, there will be at most 500 bitmaps of length equal to the cardinality of the fact table. Intuitively, the costs of reading these bitmaps will overwhelm their benefits. The situation will even exacerbate as the sparsity of bitmaps become higher. Therefore, in the current implementation only grouping bitmaps for the first grouping attribute are generated, i.e., all subgroups of the first grouping attribute are processed at the same time. This setting can be altered by commenting out the following compiler directive.

```c
#define ONLY_FIRST_GROUP
```
By doing so, all the grouping bitmaps will be generated. Experiments show that generating bitmaps for all groups not only degenerates the response time by the high start-up costs of bitmap generation, but also the execution time by the negative effect of sparse bitmaps and page I/Os.

Furthermore, once a page is read, Algorithm 6 processes all the tuples in this page such that no page is read twice. As a result, in processing latter groups only very few pages are hit for each individual group, comparing with the number of I/Os needed to scan a bitmap. In order to minimize the total I/O time, it would be favorable to combine the processing of some groups such that both the sparsity and the number of bitmap reads is reduced. In addition, with less sparse bitmaps, random I/Os of reading neighboring pages are more likely to be optimized by sequential reads. In addition to the ONLY_FIRST_GROUP directive, another compiler directive is introduced, which will generate one bitmap for \( n \) next group(s) where \( n \)'s are Fibonacci numbers, starting with 1. For example,

\[
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, \ldots
\]

denote that the first bitmap groups data of the first group, the second bitmap groups the data of the second group, the third bitmap groups the data of the third and the fourth groups, and so on. Undefining the directive

#define USING_FIBONACCI

will cause the system to generate one bitmap for each group. All the experimental results so far are obtained by commenting out the USING_FIBONACCI directive. About how this directive surprisingly affects the performance will be discussed in the next section where tests on larger data sets are conducted. Moreover, the choice of the Fibonacci numbers is just an intuitive decision. It is not based on any technical analysis.

Another implementation consideration is the generation of selection bitmaps. Theoretically, after evaluating and minimizing the selection conditions at algebraic level, bitmaps of each individual attributes are read and combined to form the final selection bitmap according to the minimized algebraic expression. An intuitive way of implementing this process is to read blocks of bits, say 32 bits, into unsigned long integers and perform the logical combinations on the integers. The whole process is repeated integer by integer. However, in practice, this kind of implementation will lead to poor performance, especially when bitmaps are generated at run time. The following compiler directive is used to toggle between generating one integer of bits at a time or one page of bits at a time, to let us explore the performance differences.

#define PAGEMODE_FOR_BITREAD

As expected, using page-mode for bitmap generation is more efficient than one integer at a time. As a matter of fact, the total number of I/Os of both approaches is the same. What makes performance different is the CPU load. Using page-modes, the condition expression of bitmap construction is tested once for one page rather than once for one integer. This is one of the reasons why bitSQL appeared to be CPU-bound in the tests conducted in Section 5.1.2. At that time, the page-mode bitmap generation was not implemented. In order to keep the lineage of development of bitSQL, we retain the test results done by former implementations. We will compare the performance of the two different approaches and investigate the changes of system loads, after we discuss the next issue. (The same PAGEMODE_FOR_BITREAD directive is also applied to grouping bitmap generation.)
As discussed in Section 5.1.2, another phenomenon observed during the running of bitSQL is that the system is CPU-bound for most of the time. The reasons are: For grouping and aggregation, in-memory operations are not optimized, due to the assumption that the program would be I/O-bound instead of CPU-bound. A two-dimensional linked-list was used to hash the aggregate results. For each qualified tuple in the fact table, the pointer-traversal throughout the two-dimensional linked-list must be performed. Obviously, it is quite costly. To improve the performance, the linked-list data structures are replaced with sorted arrays, and the searching was replaced with an improved binary search. The following directives are used to toggle between either alternatives.

#define LINKED_LIST
#define SORTED_ARRAY

After re-running the test queries, it shows that the performance was improved drastically. The performance results verify our above observation. In Figures 5.15 and 5.16, the performance of new and old versions of bitSQL is compared, with 250MB and 1GB datasets respectively. The new version is made by defining PAGEMODE_FOR_BITREAD and SORTED_ARRAY directives, tagged as bitSQL+. On the other hand, the old version creates bitmaps integer by integer and uses linked-list structures for aggregation, tagged as bitSQL.

In addition, the loads of both CPUs and I/Os are collected during the runs, illustrated in Figure 5.14. The data shows that the improved version of bitSQL balances both CPU and I/O loads better than its predecessor, compared with Figure 5.12(b), 5.5 and 5.3. The gaps between CPU and I/O loads are reduced.

![Figure 5.14: System loads during the runs of bitSQL+](image)

Finally, another issue which itself is also an optimization problem is the usage of selection bitmaps. Consider the following scenario: Taking Query 1 as an example, suppose the attribute c.nation_id is indexed, but c.mktsegment is not. To rewrite the grouping by c.mktsegment, the customers table must be read. Intuitively, regardless of being indexed or not, all selection conditions given on customers can be piggy-backed on the table scans to eventually reduce the size of groupsets, and thus reduce the costs of aggregation later. In the current implementation, all selection conditions on a table are evaluated during the table scans. Experiments show that if the selection is very selective, it will reduce the costs of grouping bitmap generation consid-
| Query 1 | 30.15-37.30 sec. bitSQL | 31.10-38.67 sec. bitSQL* |
| Query 2 | 9.24-11.17 sec. bitSQL | 11.56-13.08 sec. bitSQL* |
| Query 3 | 4.76-5.71 sec. bitSQL* | 6.74-10.02 sec. bitSQL |
| Query 5 | 8.65-9.38 sec. bitSQL* | 51.56-91.13 sec. bitSQL |
| Query 6 | 19.89-78.85 sec. bitSQL | 53.10-76.85 sec. bitSQL* |
| Query 7 | 18.57-64.67 sec. bitSQL* | 78.47-168.84 sec. bitSQL* |
| Query 8 | 55.56-80.21 sec. bitSQL* | 80.02-90.51 sec. bitSQL |
| Query 9 | 5.92-9.83 sec. bitSQL* | 6.42-81.60 sec. bitSQL |
| Query 10 | 22.04-3.17 sec. bitSQL | 9.26-71.14 sec. bitSQL* |
| Query 11 | 54.45-67.92 sec. bitSQL* | 19.18-83.11 sec. bitSQL |
| Query 12 | 8.03-64.26 sec. bitSQL* | 73.17-301.11 sec. bitSQL |
| Query 13 | 6.93-11.65 sec. bitSQL | 49.67-7.67 sec. bitSQL* |
| Query 14 | 6.82-11.70 sec. bitSQL* | 18.35-42.93 sec. bitSQL* |
| Query 15 | 6.64-11.39 sec. bitSQL* | 20.74-41.55 sec. bitSQL |
| Query 16 | 28.32-41.29 sec. bitSQL* | 100.16-127.67 sec. bitSQL |
| Query 17 | 49.97-69.97 sec. bitSQL* | 48.85-118.29 sec. bitSQL |
| Query 18 | 64.33-100.27 sec. bitSQL* | 25.25-30.95 sec. bitSQL |
| Query 19 | 26.85-30.16 sec. bitSQL* | 24.82-57.84 sec. bitSQL |
| Query 20 | 1.14-27.29 sec. bitSQL* | 418.09-396.78 sec. bitSQL |
| Query 21 | 5.96-14.21 sec. bitSQL* | 149.97-216.92 sec. bitSQL |
| Query 22 | 126.97-140.91 sec. bitSQL* | 131.73-293.96 sec. bitSQL |
| Query 23 | 138.08-197.22 sec. bitSQL* | 376.34-415.64 sec. bitSQL |
| Query 24 | 26.07-143.69 sec. bitSQL* | 19.19-11.23 sec. bitSQL |
| Query 25 | 3.74-5.44 sec. bitSQL* | 1226.04-57.15 sec. bitSQL |

Figure 5.15: Performance comparison between two versions of bitSQL, 250MB test data
Figure 5.16: Performance comparison between two versions of bitSQL, 1GB test data
erably. For the above example, even though \texttt{c.nation.id} is indexed, applying the condition may reduce the size of groupsets or even the number of groupsets. However, if the selectivity is very high, it induces processing costs. As we see, to apply or not to apply the selections, even if they are indexed, is simply a question of selectivity. Currently, no statistics is applied to perform query optimization in bit\texttt{SQL}. However, the \textit{probability of each value} in the attribute domain is maintained in the mapping table of a bitmap index for later extension.\footnote{The probability of an attribute value is defined by the fraction of the number of its occurrences in the table over the table’s cardinality. In bit\texttt{SQL}, both simple bitmap indexes and encoded bitmap indexes have mapping tables.} Another question arises: If an indexed selection is evaluated during the table scans, should its bitmaps be reused in the phase of generating selection bitmap? At first thought, it seems that after a condition is evaluated, reusing its bitmaps will be redundant. In fact, it \textit{is}. However, due to the current implementation, reusing its bitmaps may improve the performance, especially when data is to be grouped by multiple dimensions. As discussed above, grouping bitmaps are generated for the first grouping attribute only. That is, even though a selection on the second dimension is used to restrict the size of groupsets of the second dimension, it is \textit{not} used to filter out unnecessary 1s in the grouping bitmaps. As a result, reusing those bitmaps may be profitable. Again, it is a question of selectivity. To disable reusing bitmaps of evaluated selections, one should comment out the following compiler directive.

\begin{verbatim}
#define REUSE_BITMAP_FOR_SELECTION
\end{verbatim}

\textbf{Restrictions}

The current version of bit\texttt{SQL} has some simplification in its implementation, and thus infers some restrictions in its usage. For example, only natural-joins are allowed, and self-joins are disabled. The number of attributes in a table is not unlimited, neither is the number of columns in a table. Among the aggregate functions, only \texttt{SUM()} is implemented for the time being. In addition, the maximal number of opened files at a time is limited to 253 (\texttt{stdin}, \texttt{stdout} and \texttt{stderr} are reserved), due to the usage of \texttt{stdio-library}.$^{14}$ In addition, bit\texttt{SQL} runs as a normal-user, single process. It does not support multiple processors. Algorithms introduced in Section 4.2 exploiting statistics to optimize the query execution are not implemented. Algorithms which handle Order By aggregates and aggregation on dimension attributes (discussed in Section 4.3.2 and 4.3.3) are not implemented, either. There is no buffer management, no multithreading and no concurrency control. Data are stored in flat files, no raw devices/partitions can be used by bit\texttt{SQL}.

Among the above mentioned restrictions, the \textit{file descriptor limit} causes severe problems in performance evaluation. Since tables or bitmaps are both stored in files, the limit of 253 will soon be exceeded, especially, when we want to read all the grouping bitmaps to schedule the page access sequence. To work around this problem, currently the number of groupsets of the first grouping attribute, \textit{not total number of groupsets}, is limited to 253. Furthermore, by applying the approach of merging trailing groups (the numbers of merged groups are Fibonacci numbers), the consumption of file descriptors is reduced. For example, for $F_{n+2} - 1$ groups only $n$ bitmaps are created, \textit{i.e.}, only $n$ file descriptors are needed, where $F_i$ denotes the $i$-th Fibonacci number.$^{15}$

\footnote{The \texttt{stdio-library} only allows 256 file descriptors. Although the hard/soft-limits of the system are larger than 256, the program still halts due to failure of opening files.}

\footnote{Sums of Fibonacci numbers can be derived by $\sum_{k=1}^{n} F_k = F_{n+2} - 1$.}
Another way to work around the problem due to stdio-library, one can use SFIO (a Safe/Fast I/O Library) by AT&T, or abandon file I/Os and use the mmap function, memory mapped I/Os between process’s address spaces and the mapped files. Memory mapped I/Os are faster than file I/Os since the I/Os are done directly between the mapped memory buffers and the target files, without the extra transfers to and from the system buffers. According to the figures in [81], the total elapsed time of memory mapped I/Os is about \( \frac{1}{3} \) or \( \frac{1}{2} \) of file I/Os on SPARC and 80386 machines, respectively.

5.2.3 Experimental setups

In this subsection, the setups of our experimental environments are introduced, including the schema of test databases, populating the databases, testing workloads (the testing queries) and the types of tests.

Schema

The schema of the test database is adopted from the TPC-H Benchmark with slight simplifications. The partsupp relationship table and the normalized hierarchical table region in the TPC-H Benchmark are removed from our test databases. The simplifications are made such that we have both flattened dimensions and snow-flaked dimensions, whereas all dimensions in the TPC-H Benchmark are normalized. In addition, our goal here is to evaluate the performance of bitmap-based algorithms and to compare it with conventional query operators, not to provide a pricing report on overall system performance. The current version of bitSQL is so implemented that it applies bitmap-based methods to evaluate all queries, in order to allow us to compare performance of every individual query run by different algorithms. As stated before, the experiment is designed to help us with building of a higher level strategic query optimizer in the future which decides what tactic query optimizer to use. The schema of the test database is re-included in Table 5.8 for the ease of reference.

Data population

The enclosed program, dbgen, of the TPC-H Benchmark is used to populate the test databases with multiple runs. Each run contains about 32M bytes of data. The generated data are conformed to our schema, and are sorted on \((\text{order}_\text{id}, \text{part}_\text{id}, \text{supp}_\text{id})\), before loading into the warehouse. This way of data loading, instead of generating one single large set of test data, is to simulate the batch mode data loading nature.

Power-test queries

For the purpose of evaluating bitmap-based selection, grouping and aggregation algorithms, 27 queries over the above schema are designed. (Some of them are adopted from the TPC-H Benchmark with corresponding modifications to conform to our schema.) The queries are divided into two categories — one involves selections, while the other does not. In addition, all the queries (of both categories) contain Star-Joins, aggregation, Group Bys and Order Bys, since in the Data Warehousing environment most queries probably aggregate data of the fact
tables and group the results by dimensions. It is reasonable to claim that typical OLAP queries contain all the above operations. What really differs from time to time is the subsets of data of interest, which are determined by the selection conditions.

For the interest of bitmap processing, the queries with selections are further classified into two sub-categories. In one subcategory, all selecting attributes are bitmap indexed, while in the other, some attributes are not indexed. According to the current implementation, non-indexed selections on dimension attribute(s) will be automatically rewritten to selections on dimension key(s). Later, we will also discuss how performance improves, if additional bitmap indexes are defined to skip query revision, comparing with the performance of other DBMSs with additional B-trees defined on the same attribute(s), to see how different systems react to additional indexes.

In addition, since our goal is to examine how bitmap-based algorithms perform in every individual case, we do not claim that the test query set represent the typical workload of the OLAP environment. Instead, the queries are varied in such a way that we gain the knowledge of how to tune the bitmap indexes to improve the performance, or of when bitmap indexes should be abandon at all. No comparison on the overall average response time or average execution time among different DBMSs will be made, since if the workload is not representative, the average values do not make much sense.

In the TPC-H Benchmark, two kinds of tests are defined, one is the power test and the other is the throughput test. The power test is to measure the raw query execution power of the system when connected with a single active user, while the throughput test is to measure the performance of the system against a multi-user workload. In this work, we are only interested in power tests. In the tests, the side-effects of caching have been considered and eliminated through a brute-force method — shutting down the DBMSs and restarting them.
Setting up commercial database management systems for performance tracing

The data set loaded into CDB1, CDB2 and CDB3 is the same as the one loaded into bitSQL. The set of indexes (compound B-trees, listed in Table 5.7) defined in CDB1, CDB2 and CDB3 is adopted from [41]. The commercial systems are installed and used as they are. No tuning is performed on them. As stated above, the purpose of the experiments is to compare the performance of each individual query and to find out the reasons why one method outperforms another, not the overall performance of the systems. In the performance measurements presented in this work, they represent the performance of conventional query processing methods without using bitmap indexes, not the absolute performance of the systems. There are surely other tuning/improvement techniques, such as using materialized views to answer queries or partitioning/striping the data, to tune the performance of the commercial systems, but these are beyond the scope of this work.

5.3 Experimental Results

First, the results of running the queries on 1 GB test data are presented. In Figure 5.17, 5.18, 5.22 and 5.23, the measures for bitmap-based methods are obtained by the version of bitSQL that is made without PAGEMODE_FOR_BITREAD, without USING_FIBONACCI, and with LINKED_LIST compiler directives. This version of bitSQL is also the one used so far, other than noted otherwise. Two other versions of bitSQL will also be introduced to explore how different implementations affect the performance. Then, the experimental results on 2 GB test data under Windows NT, and results on 4 GB test data under Solaris are reported. For 2 GB test data and up, bitSQL ++ (defined later) is used to test bitmap-based algorithms only.

5.3.1 Test results of CDB1 and bitSQL under Solaris with 1 GB test data

In Figure 5.17 and 5.18, the performance of CDB1 and bitSQL is compared. Again, two set of indexes are defined on CDB1 data. One defines indexes on all key and foreign key attributes, and the other defines the set of indexes proposed by [41]. The performance of CDB1 on the two index sets does not show significant differences. Therefore, we concentrate the discussion on those cases where bitSQL performs considerably worse than either of the two CDB1 runs.

Let us look at Query 2 first. bitSQL spent about 800 seconds for query rewrites and bitmap generation until the first result is available, and it spent about another 700 second to finish the query. The long response time is due to the costly Group By rewrites (from c.nation_id to l.order_id) and the generation of grouping bitmaps. From the program logs, we see that about the first 620 seconds are needed for grouping bitmap generation and only approximately 5 seconds for selection bitmap generation (since o.year is indexed). The interval of 175 seconds between 625-th second and the response time are used to produce the first aggregate. Five rows are in the result set of Query 2. That means, on the average, another 175 seconds are needed to finish an aggregate.

Three things can be learned from the execution of Query 2 — the inefficiency of bitmap generation phase, the long response time between the timepoints when bitmaps are ready and the first result is available, and the constant time interval among subsequent aggregate results.
Figure 5.17: Performance comparison between CDB1 and bitSQL, 1 GB test data (continued)
In order to rewrite the grouping by `c.nation_id` to the one by `l.order_id`, both customers and orders are scanned sequentially. This task should have been improved by defining a group-set bitmap index on `c.nation_id` or secondary access paths on both dimension tables. However, neither of them is available in the experimental version of `bitSQL`. No index, neither bitmaps nor B-trees, is allowed on snow-flaked dimension tables. This restriction is intended to allow us to measure and to compare the time required by our query rewrite mechanisms with the time required by join operations of conventional query processing techniques. Without removing the above restrictions, we apply the directives — `PAGEMODE FOR BITREAD` — to improve the efficiency of bitmap generation. As for the phase of aggregation, `LINKED_LIST` is replaced with `SORTED_ARRAY` to reduce the costs of in-memory operations. The performance of `bitSQL` improves significantly after the above two directives have been applied, tagged as `bitSQL+` in Figure 5.19. As expected, the page-mode bitmap generation improves the efficiency of generating grouping bitmaps. In addition, replacing the linked-list traversals with improved binary searches reduces the total execution time by a factor of about 1.6.
helper functions, not due to the bitmap algorithms themselves, since the bitmap algorithms are not changed. As for the third observation, the constant time intervals of subsequent aggregates are still satisfied, which is supposed to be dependent on the distribution of the grouping attribute. The only change is that the intervals are reduced from 128 seconds downto ca. 50 seconds, due to the more efficient searching algorithm.

For the first time, **USING_FIBONACCI** is applied for grouping bitmap generation. (Both the **PAGEMODE_FOR_BITREAD** and **SORTED_ARRAY** remain.) Its performance is tagged as bitSQL++ in Figure 5.19. The performance is better than ever. It even outperforms CDB1. The reason of the good performance is simply the reduced costs of bitmap processing at the beginning of query processing, since we see that the time intervals between the response time and the completion time of both bitSQL+ and bitSQL++ only have minor differences. However, query rewrites still take long to accomplish. As stated above, this can be reduced through a group-set bitmap index on the snow-flaked dimension table.

Queries 4, 5, 9, 12, 14, 16, 19, 20, 23, 25 and 26 have similar problems of long periods of bitmap generation and slow aggregation due to inefficient in-memory searches (the reason of high CPU loads). Among them, the response time of Queries 4, 5, 9, 12, 19, 23 and 26 are even longer than the execution time of the same runs by CDB1.

After applying **PAGEMODE_FOR_BITREAD** and **SORTED_ARRAY** to re-make bitSQL, the above queries are rerun. Queries 9, 12, 14 and 23 complete before the same runs by CDB1, not to mention the faster response time. Furthermore, after enabling the directive **USING_FIBONACCI**, the time measures of other four queries (Queries 16, 19, 20 and 25) are also better than those for CDB1. The performance of CDB1 and two variants of bitSQL are depicted in Figure 5.20 and 5.21. Later in the text, bitSQL+ and bitSQL++ are used to denote the version of bitSQL applying **PAGEMODE_FOR_BITREAD** and **SORTED_ARRAY**, and another version applying **USING_FIBONACCI**, respectively.

From the experiments conducted on bitSQL+ and bitSQL++, we see that, comparing with bitSQL, the performance of bitSQL+ is improved by about a factor of 1.7 regarding the response time, and by a factor of 1.5 regarding the query execution time. As for bitSQL++, its performance is improved by a factor of 2.8 regarding the response time, and by a factor of 2.7 regarding the query execution time. Surprisingly, bitmap algorithms used in bitSQL++ outperform conventional methods (represented by CDB1) for 24 out of the 27 queries. This result is far beyond our expectation.

However, let us take a look at the rest three queries, where bitSQL++ still performs poorly. Queries 4 and 5 are similar. They differ from each other only in selection conditions. From the logs of bitSQL++, it shows that both queries spent a large fraction of time between the response time and the completion time, so do Queries 7 and 13. However, Queries 7 and 13 involve 3-table Star-Joins, whereas Queries 4 and 5 involve only 2-table joins. Obviously, the costs of query rewrites are better paid off when joining more than two tables. Especially, in Queries 4 and 5, the small table parts is even further reduced by selection conditions, such that even a nested-loop join may provide good performance.

---

16 Originally, Queries 4 and 5 are designed to investigate the behavior of bitmap-based selections on different selectivities. And, Query 23 is to further examine the changes in performance without effects of selection bitmaps. Although the results do not reveal significant differences between the performance of the first two, we still keep them both in the set of test queries since negative results do not necessarily invalidate the value of the tests.
Figure 5.20: Performance comparison between CDB1 and different versions of bitSQL, 1 GB test data (continued)
Last, Query 26 reveals the very case where the costs of query rewrites (rewriting the grouping by the snow-flaked dimension customers all the way down to the fact table and rewriting the non-indexed selection on the dimension orders to its foreign key in the fact table) bring the system to its knees. We see that after the bitmaps are ready, the system needs only ca. 40 seconds to complete the execution. However, it needs ca. 440 seconds to get the bitmaps ready. In such cases, defining group-set bitmap indexes on c.cust_id of the snow-flaked dimension customers will greatly reduce the startup time and improve the performance since query rewrites are replaced with bitmap processing. Queries, which should have performed even better if group-set bitmaps are allowed on snow-flaked dimensions, are Queries 1, 2, 6, 11, 14, 17, 18, 19, 20, 21, 24 and 25.

### 5.3.2 Test results of CDB3, CDB2 and bitSQL under Windows NT with 1 GB test data

In Figures 5.22 and 5.23, the performance results of CDB3, CDB2 and bitSQL running on the Windows NT platform are presented. Similar to the results on 250 MB test data, CDB2 and CDB3 provide very good performance under Windows NT, especially CDB2. Both commercial DBMSs provide outstanding performance over bitSQL because of the huge working set\textsuperscript{17} that they can acquire from the OS. It is observed that during the runs of the queries, CDB2 used up to 450 MB main memory, and CDB3 used up to 300 MB memory, while bitSQL used only 2 to 3 MB memory. Nevertheless, under such circumstances, there is still one case (Queries 10 and

\textsuperscript{17} Microsoft defines the working set of a process by the set of memory pages in the main memory touched recently by the threads in the process. If free memory in the computer is above a threshold, pages are left in the working set of a process even if they are not in use. When free memory falls below a threshold, pages are trimmed from working sets. If they are needed they will then be soft-faulted back into the working set before they leave main memory.
Figure 5.22: Performance comparison between CDB2, CDB3, bitSQL and bitSQL++, 1 GB test data (continued)
Figure 5.23: Performance comparison between CDB2, CDB3, bitSQL and bitSQL++, 1 GB test data

27, but they are considered as the same type of query) where bitSQL outperforms both CDB2 and CDB3, and there are five other queries (Queries 5, 7, 13, 22 and 23) where bitSQL provides comparable performance with either of the DBMSs. For bitSQL++, there are four additional cases (Queries 9, 12, 18 and 21) where it outperforms both CDB2 and CDB3 and eight additional queries (Queries 4, 5, 7, 13, 16, 20, 22 and 23) where it provides comparable performance with either of the DBMSs.

The execution of every single query will not be analyzed here further, since without a fair basis the analysis will be misleading. Until bitSQL has implemented its own buffer management (for the improvement of I/O operations), the brute-force comparisons between bitSQL and CDB3, or between bitSQL and CDB2 will not help us further with the analysis of the performance of bitmap-based algorithms. In the next section (where tests on 2 GB data are conducted), the effects of memory usage will be discussed in more detail. In addition, the reasons why bitSQL++ performs well or poorly are explained, and solutions for the poor-performing cases are suggested.
Figure 5.24: Performance comparison between CDB2 and \texttt{bitSQL++}, 2 GB test data
5.3.3 Test results on 2 GB test data

Figure 5.24 shows the run times of the 27 queries by CDB2 and bitSQL++ on 2 GB data. In Table 5.9, the selectivities of all the 27 queries are listed. The selectivities are calculated by the fraction of the numbers of selected tuples over the cardinalities of the (joined) tables. Since all the joins are Star-Joins, the cardinality of the joined table is equal to the cardinality of the fact table, lineitems.

<table>
<thead>
<tr>
<th>Query</th>
<th>Selectivity (%)</th>
<th>Query</th>
<th>Selectivity (%)</th>
<th>Query</th>
<th>Selectivity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.80</td>
<td>10</td>
<td>0.07</td>
<td>19</td>
<td>0.88</td>
</tr>
<tr>
<td>2</td>
<td>10.47</td>
<td>11</td>
<td>2.54</td>
<td>20</td>
<td>1.06</td>
</tr>
<tr>
<td>3</td>
<td>0.09</td>
<td>12</td>
<td>3.68</td>
<td>21</td>
<td>1.60</td>
</tr>
<tr>
<td>4</td>
<td>9.20</td>
<td>13</td>
<td>3.208</td>
<td>22</td>
<td>100.00</td>
</tr>
<tr>
<td>5</td>
<td>0.06</td>
<td>14</td>
<td>3.58</td>
<td>23</td>
<td>33.58</td>
</tr>
<tr>
<td>6</td>
<td>1.79</td>
<td>15</td>
<td>0.00</td>
<td>24</td>
<td>7.28</td>
</tr>
<tr>
<td>7</td>
<td>15.23</td>
<td>16</td>
<td>100.00</td>
<td>25</td>
<td>7.25</td>
</tr>
<tr>
<td>8</td>
<td>0.22</td>
<td>17</td>
<td>0.22</td>
<td>26</td>
<td>0.86</td>
</tr>
<tr>
<td>9</td>
<td>4.64</td>
<td>18</td>
<td>0.07</td>
<td>27</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Table 5.9: Selectivities of all test queries

As we know, B-trees are well-suited for selections with very low selectivities. For those cases in bold face in Table 5.9, CDB2 does provides very good performance, except for Queries 10, 18 and 27. Queries 3, 8, 15, 17, 19 and 26 select less than 1% of the underlying tuples, and these are definitely cases where B-trees are better suited than bitmap indexes. It is thus not surprising that CDB2 outperforms bitSQL++ in these cases. However, CDB2 performs poorly on Queries 10, 18 and 27. This is simply because there does not exist any index on the selection attributes, suppliers.nation_id and customers.nation_id, and all the three queries involve selections on these two attributes.

It is now interesting to know the performance of CDB2 if CDB2 cannot obtain the huge memory space it defaults to. In order to obtain comparable results for those queries with medium to high selectivities, the server memory of CDB2 is deliberately limited to 15 MB. Figure 5.25 shows the run times of the 27 queries on 2 GB data by bitSQL++ and CDB2 with maximal server memory equal to 15 MB.

From Figure 5.25, we see that the performance of CDB2 degrades for Queries 4, 5, 7, 12, 13, 14, 16, 22 and 23. In addition, for Queries 4, 5, 12, 22 and 23, bitSQL++ even provides better query execution times than CDB2. For other queries, such as Queries 2, 24 and 25, if additional group-set bitmap indexes on the snow-flaked dimension tables are defined, the performance of these queries will be improved considerably. From the above results, we see that for queries with medium to high selectivities, bitmap indexes and bitmap-based query processing techniques are better suited than B-trees and conventional methods.

Although disk space is nowadays no longer a problem, it is worth mentioning that defining the set of bitmap indexes, shown in Table 5.3(b), on the 2 GB database requires ca. 500 MB, while defining the set of B-trees, shown in Table 5.7, on the 2 GB test database, CDB2 uses 6.5 GB for the indexes. For this amount of space, we can define bitmap indexes on all attributes.

---

18 We still give CDB2 memory five to six times than of bitSQL++ since such a giant system it requires surly more memory by nature.

19 CDB2 uses the option fill factor to specify how full a page should be when creating an index. The default fill factor is 0, which means CDB2 will create clustered indexes with full data pages and nonclustered indexes with full leaf pages, but leave some space in the upper level pages of the index.
Figure 5.25: Performance comparison between CDB2 (using up to 15 MB memory) and bitSQL++, 2 GB test data.
5.3.4 Test results on 4 GB test data

Figure 5.26 shows the performance results of running the test queries on 4 GB test databases under the Solaris platform. Comparing with the results on 1 GB test databases, the reason why the execution times of some queries by CDB1 on 4 GB data are even shorter than on 1 GB data is that UPDATE STATISTICS has been applied carefully on the 4 GB CDB1 database to refresh the statistical information such that the query optimizer works more precisely. Even though the statistical information is refreshed with high precision, there are still 12 queries where bitSQL++ outperforms CDB1 and 5 additional queries with shorter response times and comparable execution times. Since the results conform to the results on 1 GB test data, we do not discuss every single query any further.

To conclude the experiments, another compiler directive — BIT HASHING — is introduced to remake bitSQL. With this directive, bitSQL applies bitmap indexes for query rewrites and selection evaluation only. Aggregation and grouping are evaluated by hashing. The purpose of this version of bitSQL, tagged as bitSQLΔ, is to understand how bitmap-based algorithms perform if we wish to optimize the query execution time, instead of query response time. The performance results on 4 GB test data, compared to CDB1, are illustrated in Figure 5.27. As expected, compared with bitSQL++, bitSQLΔ provides shorter query execution times, but it is less responsive.

5.4 Experiences and Perspectives

In the former chapters, query processing and optimization techniques using bitmaps have been introduced. In order to understand how these bitmap-based algorithms work, a bitmap-enabled SQL engine is implemented, and some performance measurements are presented. During the tests, some practical experience on (bitmap) index tuning are gained, and potential improvements or extensions for future versions of bitSQL are recognized.

5.4.1 Index tuning

Simple bitmap indexing versus Encoded bitmap indexing

In Section 5.1.2, we discussed the case where the grouping bitmaps of non-indexed grouping attribute(s) must be generated at run time. Remember that such rewrites of Group By clauses are based on the functional dependencies between key and non-key attributes. That is, the bitmaps of the key attribute of the containing table of the grouping attribute(s) are exploited to construct the grouping bitmaps. Thus, the costs of the bitmap generation depend highly on the number of bitmaps that must be read in and written out. The result in Section 5.1.2 shows that it is more efficient to construct the bitmaps from encoded bitmap indexes than from simple bitmap indexes.

Here, we would like to discuss another scenario where the (non-key) grouping attributes are indexed. For example, Queries 4, 5 and 23 group data by p.brand which has an encoded bitmap index defined. According to the current implementation, simple bitmaps are required by the bitmap grouping algorithms, therefore, although p.brand is indexed, the bitmaps of p.part_id...
Figure 5.26: Performance comparison between CDB1 and bitSQL++, 4 GB test data
<table>
<thead>
<tr>
<th>Query</th>
<th>SQL Time (sec.)</th>
<th>CDB1 Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[833.29]</td>
<td>[889.35]</td>
</tr>
<tr>
<td>2</td>
<td>[1288.60]</td>
<td>[1660.42]</td>
</tr>
<tr>
<td>3</td>
<td>[89.66]</td>
<td>[805.38]</td>
</tr>
<tr>
<td>4</td>
<td>[879.32]</td>
<td>[813.02]</td>
</tr>
<tr>
<td>5</td>
<td>[871.58]</td>
<td>[826.26]</td>
</tr>
<tr>
<td>6</td>
<td>[819.34]</td>
<td>[826.34]</td>
</tr>
<tr>
<td>7</td>
<td>[1273.69]</td>
<td>[1273.69]</td>
</tr>
<tr>
<td>8</td>
<td>[380.39]</td>
<td>[829.64]</td>
</tr>
<tr>
<td>9</td>
<td>[265.94]</td>
<td>[898.66]</td>
</tr>
<tr>
<td>10</td>
<td>[262.86]</td>
<td>[946.56]</td>
</tr>
<tr>
<td>11</td>
<td>[855.55]</td>
<td>[855.55]</td>
</tr>
<tr>
<td>12</td>
<td>[272.12]</td>
<td>[624.90]</td>
</tr>
<tr>
<td>13</td>
<td>[435.39]</td>
<td>[1745.94]</td>
</tr>
<tr>
<td>14</td>
<td>[638.25]</td>
<td>[886.03]</td>
</tr>
<tr>
<td>15</td>
<td>[478.46]</td>
<td>[2590.72]</td>
</tr>
<tr>
<td>16</td>
<td>[300.75]</td>
<td>[2590.72]</td>
</tr>
<tr>
<td>17</td>
<td>[127.52]</td>
<td>[816.01]</td>
</tr>
<tr>
<td>18</td>
<td>[300.82]</td>
<td>[833.58]</td>
</tr>
<tr>
<td>19</td>
<td>[744.89]</td>
<td>[888.85]</td>
</tr>
<tr>
<td>20</td>
<td>[244.97]</td>
<td>[607.75]</td>
</tr>
<tr>
<td>21</td>
<td>[238.27]</td>
<td>[699.45]</td>
</tr>
<tr>
<td>22</td>
<td>[307.68]</td>
<td>[2585.40]</td>
</tr>
<tr>
<td>23</td>
<td>[241.67]</td>
<td>[2586.20]</td>
</tr>
<tr>
<td>24</td>
<td>[1155.67]</td>
<td>[1225.45]</td>
</tr>
<tr>
<td>25</td>
<td>[1100.76]</td>
<td>[1225.45]</td>
</tr>
<tr>
<td>26</td>
<td>[1844.76]</td>
<td>[847.30]</td>
</tr>
<tr>
<td>27</td>
<td>[344.32]</td>
<td>[847.30]</td>
</tr>
</tbody>
</table>

Figure 5.27: Performance comparison between CDB1 and bitSQL\textsuperscript{A}, 4 GB test data
are applied to construct the grouping bitmaps and abandon `p.brand`'s encoded bitmaps. Intuitively, if we define a simple bitmap index on `p.brand`, the performance should be improved. To understand the changes, the encoded bitmap index is dropped and a new simple bitmap index on `p.brand` is created. After rerunning the queries, all Queries 4, 5 and 23 respond much earlier, since no grouping bitmap needs to be generated at run time, as Figure 5.28 shows. However, Queries 4 and 5 do not have equivalent improvements on the execution time. Query 4 even takes slightly longer to finish than before. The reason is the influence of selection bitmaps. Whereas Query 23 does not involve any selection bitmap, it responds and ends much earlier than before. (The non-indexed selection on the fact table is evaluated at the time of data retrieval.)

Figure 5.28: Performance comparison between simple bitmaps (SBI) and encoded bitmaps (EBI)

Another two queries, which are also affected by the redefinition of `p.brand`'s index, are Queries 7 and 13. However, we see that the performance of the two queries is not remarkably affected by the types of bitmaps. Query 13 performs even slightly better with encoded bitmaps than with simple bitmaps. The reason is the range of the selection. Query 13 selects eight brands of products. Using simple bitmaps, eight bitmaps are read and ORed to obtain the final selection bitmap, while with encoded bitmap indexes the upper limit of the numbers of bitmap scans is a logarithmic function of the attribute’s cardinality of base 2. For wide range selections, encoded bitmap indexes will perform better than simple bitmap indexes, since the number of bitmap scans will not grow with the range of selections. In the example of Query 13, the cardinality of `brand` is 25, i.e., to evaluate the selection on `p.brand`, at most five encoded bitmaps are read. Obviously, it takes less time than reading eight simple bitmaps.

The following rules are proposed as simple guidelines for index design: For dimension keys and other high cardinality attributes, encoded bitmap indexes are well-suited. While for low cardinality attributes, or attributes on which grouping conditions are often given, simple bitmap indexes are better suited than encoded bitmap indexes.

**Bitmap creation on demand**

In the current implementation, bitmaps on demand are generated using existing bitmaps. However, sometimes it is even more efficient to create the bitmaps on their own. In Table 5.10, the time of creating each bitmap index on the 4 GB data is listed. We can compare the time of
bitmap creation with the time of dynamically generating bitmaps using other existing bitmaps. For example, according to the logs of bitSQL++, Query 2 took over 1300 seconds to generate the grouping bitmaps for customers.nation_id from the bitmaps of orders.order_id. From Table 5.10, we see that it takes ca. 407 seconds to build a simple bitmap index on suppliers.nation_id. Although it does not directly imply that it takes the same time to create a bitmap index on customers.nation_id, it should be a good approximation of the actual elapsed time.

<table>
<thead>
<tr>
<th>Type of index</th>
<th>Indexed attributes</th>
<th>Time elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoded bitmap</td>
<td>lineitems.part_id</td>
<td>414.22</td>
</tr>
<tr>
<td>encoded bitmap</td>
<td>lineitems.supp_id</td>
<td>305.21</td>
</tr>
<tr>
<td>encoded bitmap</td>
<td>lineitems.order_id</td>
<td>556.52</td>
</tr>
<tr>
<td>encoded group-set bitmap</td>
<td>parts.brand</td>
<td>276.43</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>parts.type</td>
<td>261.08</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>suppliers.nation_id</td>
<td>407.56</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>orders.priority</td>
<td>308.20</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>orders.orderstatus</td>
<td>309.17</td>
</tr>
<tr>
<td>encoded group-set bitmap</td>
<td>orders.cust_id</td>
<td>441.07</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>orders.year</td>
<td>331.26</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>orders.month</td>
<td>377.78</td>
</tr>
<tr>
<td>simple group-set bitmap</td>
<td>orders.day</td>
<td>414.91</td>
</tr>
</tbody>
</table>

Table 5.10: Time elapsed for index creation on 4 GB data

In addition, bitmaps generated on the fly are not made persistent in the current implementation. After the query execution, they are simply discarded. This can be enhanced by a caching mechanism. With help of usage profiles, the system can reserve an area for storing the generated bitmaps before either discarding them, or making them permanently persistent. The bitmap indexes defined in the test database use about \( \frac{1}{5} \) of the total database size. Comparing with the space requirements of B-trees, bitmap indexes are quite space efficient.

**Improving the merging of trailing groups**

In Section 5.2.2, we discuss about merging trailing groups to minimize the total I/Os, and the Fibonacci numbers are simply applied to determine the number of groups to merge. However, this task can be optimized through the usage of statistics stored in the mapping table of each bitmap index, such as the selectivity of each attribute value, to keep the sparsity of each grouping bitmap from dropping below a threshold. This task is left to the future version of bitSQL. The definition of such a threshold will be dependent on the size of available main memory and the performance criteria.

**5.4.2 Perspectives**

The experimental version of bitSQL simplifies many implementation details, and applies bitmap-based algorithms exclusively for query evaluation. Future versions of bitSQL should integrate conventional methods into its SQL engine. In the following, some extensions and improvements which should be included into bitSQL are listed.
Issues related to bitmap processing

First of all, the types of bitmap indexes admitted by the experimental version of SQL are restricted to simple bitmaps and encoded bitmaps. Another important type of bitmap indexes, which is not implemented for the time being are bit-sliced indexes. Bit-sliced indexes are especially well-suited for numeric data types and continuous range selections. Query evaluation algorithms, including selection evaluation and aggregation, which perform on bit-sliced indexes have been introduced in the literature [65, 16, 17, 92]. In Chapter 4, we have also introduced some further optimization techniques, such as tree reduction and the principle of inclusion and exclusion, for bit slices [92, 15]. All these techniques will extend the system to be able to optimize selections on numerical data, or aggregation.

Second, concerning encoded bitmap indexing, the encoding function plays an important role of this indexing scheme. For selection evaluation, the numbers of bitmap scans can be minimized through a well-defined encoding. For example, for typical selections along a dimension hierarchy, they can be optimized through a hierarchy encoding. Note that such an encoding is not only good for selection, but also helpful for grouping by any attribute along the dimension hierarchy. Currently, no scheme for encoding definition is implemented. The attribute values are encoded in ordinal order as they appear in the table. However, the number 0 is reserved for non-existent tuples.

Third, the optimization techniques introduced in Chapter 4, which exploit statistics information of the underlying data, are not implemented. The information about, e.g., the selectivities, or the number of 1 bits in both page-level and tuple-level bitmaps, are stored in the mapping table for future extension. This information is gathered on the fly of index construction, and is used by the optimization techniques, such as the principle of inclusion and exclusion and the merging of trailing groups discussed in Section 4.2.2.

Fourth, regarding grouping and aggregation, bitmap indexes on snow-flaked dimension tables are not allowed, aggregation on dimension table is not yet supported, and Order By clauses can not contain aggregated attributes. All these functionalities are to be extended.

Last, but not least, in this work bitmaps and bitmap-based query processing and optimization techniques are introduced and they are proven to be useful in the Data Warehousing environment. However, as a running system, SQL should provide a strategic query optimizer which chooses among bitmap-based algorithms and conventional methods. In addition, it also requires careful investigation whether or not a hybrid approach can further improve the performance. For example, the drawbacks of hash methods are the long startup time, since huge tables must be first recursively partitioned into small pieces, such that the hashed results of each individual run can fit into memory. On the other hand, the bitmap grouping suffers from slow random page I/Os, compared with sequential I/Os of the hash methods. What if we choose an average approach, by replacing the recursive partitioning with grouping bitmaps? That is, instead of actually partitioning the data, data is read and hashed according to grouping bitmaps. One grouping bitmap is generated for groups such that the hash table for these groups can fit into memory. The choice of n is similar to determining the recursion depth of the hash methods.
Issues related to system architecture and functionalities

The only aggregate function supported in the current version of bitSQL is $\text{SUM}()$. Future versions of bitSQL should provide other aggregate functions which are usual to the OLAP applications. Other improvements concerning programming details include: multithreading bitmap provider and bitmap consumer, optimizing page access sequences [77], supporting multiprocessors, providing buffer management, using virtual memory objects provided by the OS, or even providing its own I/O subsystem, exploring parallelism among physical operations (e.g., pipelining bitmap provider and bitmap consumer), and applying compression techniques to minimize I/O costs of bitmap scans.

SUMMARY In this chapter, the architecture and some implementation issues of a bitmap-enabled SQL engine, bitSQL, are introduced, and the experimental results of running a set of queries on bitSQL and three other commercial DBMSs on two different platforms are presented. The experimental version of bitSQL enables us to explore the performance of the bitmap-based algorithms introduced in Chapter 4.

The aims of the experiments are 1) to justify the introduction of bitmap indexes and bitmap-based query processing algorithms to query processing in Data Warehousing and 2) to identify the situations where bitmap algorithms perform better than conventional methods, or vice versa, such that a higher-level query optimizer can be implemented. For the first objective, the results are quite satisfying and they do show the strength of bitmap algorithms. For the second objective, we recognized the potential to improve bitSQL through some advanced programming techniques, and identified queries by their selectivities, that can be more efficiently processed using bitmap algorithms.
Chapter 6

Conclusions and Future Work

Indexes indicate where to locate items of interest. In databases, indexes provide the addresses of data objects that contain the values in question. Different index structures are designed for different purposes, and the more complex the index structures are, the less cooperative they are with indexes designed for other purposes.

Bitmap indexing on the contrary, since its emergence in the 1960s, has shown its strength based on its simplicity. No matter what kind of bitmaps the indexes are, and what purpose the bitmaps have, they are all common in their simple data structure — bit string. Its simple data structure allows different bitmap indexes to be efficiently combined by logical operators, such as AND, OR and NOT. Moreover, the results are still bitmaps that preserve the address information of desired tuples. However, the original bitmap indexes have their limitations. That is, as the cardinality of the indexed attribute increases, the large space requirements cause another severe problem of sparsity in bitmaps. In order to solve these problems, but at the same time retaining the simplicity of bitmap indexes, a new design scheme of bitmap indexing — encoded bitmap indexing — and its query processing algorithms are proposed in [91, 92] and [15]. Encoded bitmap indexing reduces the space requirements to an order of a logarithm of attributes’ cardinalities through domain encoding and minimizes the numbers of bitmap scans at query processing time through well-defined encodings.

Among other variants of bitmap indexes, encoded bitmap indexes and bit-sliced indexes, proposed in [65, 16, 17], provide solutions, which cover discrete and continuous range selections, to query processing in Data Warehousing. Discrete range selections are selections expressed as IN-lists, usually, on alphanumeric attributes, while continuous range selections are selections on the continuous ranges of, usually, numeric attributes. In this work, the design space of bitmap indexing is identified, and the design issues of encoded bitmap indexes and bit-sliced indexes subject to query types and resource limitations are explored. In addition, how to use bitmap indexes to improve the tasks of query processing and optimization in Data Warehouses is discussed. Static and dynamic optimization techniques are introduced for selection evaluation using encoded bitmap indexes and bit-sliced indexes. Furthermore, bitmap-based query processing techniques for aggregation, Group Bys and Order Bys, which minimize the query execution time subject to minimization of query response time, are proposed.

Through the uses of bitmap indexes, query processing and optimization in Data Warehouses change a bit.
• First, the fact that the results of bitmap processing are still bitmaps postpones the data retrieval to the end of query processing. That means, most of tasks of query evaluation are performed on bitmaps without accessing the underlying data. Bitmaps that resulted from preceding query operations are passed on to subsequent operators. Pipelining individual bitmap processing algorithms even reduces the costs of materializing the intermediate bitmaps.

• Second, through the introduction of group-set bitmap indexes, which can be realized as simple or encoded bitmap indexes, Joins and Group Bys are optimized. Group-set bitmap indexes are indirect indexes, i.e., a referenced tuple does not contain the indexed value, instead it is associated with the indexed value through a Join to another table. In other words, group-set bitmap indexes are some sort of bitmapped join indexes. However, they are not used to ease the task of joining two tables; they are used to skip the Join operations. They usually serve as access plans to fact table tuples which belong to a group by some dimension attribute(s).

Furthermore, not just for Group By operations, the group-set bitmap indexes can also be used for selections.

• Third, with the help of bitmap indexes on the foreign keys of the fact table(s), even though no group-set bitmap index is defined on the grouping attribute(s), both the Group By clauses and selections on dimension table(s) can be rewritten such that the Star-Joins to the dimension table(s) are skipped.

However, as the experimental results in the last chapter shows, the query rewrites can be sometimes very expensive, especially when the dimension tables are snow-flaked. Since bitmap indexes, especially encoded bitmap indexes, are quite space efficient and memory capacity is nowadays no longer a big deal, it is preferable to define group-set bitmap indexes on attributes on which grouping or selection conditions are likely to be given.

• Last, but not least, all the above properties and behaviors of bitmap indexes simplify the phase of query optimization. All selections are evaluated directly on bitmaps; no join orders need to be investigated, and no join operations need to be optimized. As a result, it reduces the cost of query optimization and increases the net benefits of query optimization.

In order to show the feasibility and the advantages of bitmap indexes and bitmap-based algorithms, a bitmap-enabled SQL engine, bitSQL, is implemented, and experiments are conducted on it and on three other commercial DBMSs. The experimental results confirm our analysis and show that the bitmap-based query processing techniques provide significant improvements to query processing in Data Warehousing. Based on the results of the experiments and the experiences gained from implementation of bitSQL, we think that bitmap indexing plus bitmap-based query processing techniques is one of the key solutions to query processing/optimization in the Data Warehousing environment.

The good performance of the current version of bitSQL encourages us to build a full functional bitmap-enabled query processor and optimizer for Data Warehousing. As future work, we are looking forward to integrate bitmap indexing and the bitmap algorithms into a database kernel. Favorable extensions to bitSQL are included as follows.
A cost-based or rule-based strategic query optimizer can help to choose the optimal query execution plan among conventional query processing methods and bitmap algorithms. The preliminary classification by query selectivities used in the previous chapter is just for explanatory purposes of why one method outperforms another. A more precise cost-model or heuristic rules are needed.

Considering the optimization of query processing only (without the effort of minimizing the space requirements or of defining dense indexes), there is a number of variants of bitmap indexes that can further optimize query processing.

First, for attributes with wide domains, if only a small number of subdomains are frequently queried, a simple bitmap index can be defined on these subdomains only, instead of on the whole attribute domain. For example, suppose that attribute \( A = \{a, b, c, d, e, f, g, h, \ldots, x, y, z\} \), and only subdomains \( \{a\}, \{a, b\} \) and \( \{g, h\} \) are frequently queried. We can define three simple bitmaps on \( \{a\}, \{a, b\} \) and \( \{g, h\} \) and leave other values un-indexed. Obviously, this index can quickly locate the desired tuples if the ranges of selections match any of the above subdomains. However, the index is not dense since the index cannot be used to retrieve every value in the attribute domain. For example, predicates like \( A \in \{b\} \) and \( A \in \{f\} \) cannot be evaluated using this index. In addition, some key value(s) may be indexed more than once, e.g., the value \( a \).

To remedy this drawback and to ensure that all selection predicates on \( A \) can be evaluated by the index, we can apply a hybrid approach by combining simple and encoded bitmap indexing. Following the above example, in addition to the three simple bitmaps on \( \{a\}, \{a, b\} \) and \( \{g, h\} \), encoded bitmaps can be defined on the subdomain \( \{b, c, \ldots, x, y, z\} \). Naturally, the query optimizer should be enhanced to be able to choose between simple or encoded bitmaps based on the selection ranges.

Second, the current definition of encoded bitmap indexes and well-defined encoding are based on the condition that the number of bitmaps is minimized and is determined by the logarithmic function of base 2 of the attribute’s cardinality. Under such a condition, finding a well-defined encoding is an NP-problem. To ease the task of finding a well-defined encoding, we can release the condition of minimized space requirements of encoded bitmap indexes. In addition, while defining the encoding, we should take the frequencies of selection predicates into consideration.

For example, suppose that attribute \( A = \{a, b, c, d, e, f, g, h\} \), and there are three predefined range selection predicates on \( A \) — \( A \in \{a, b\} \), \( A \in \{c, d, e, f\} \) and \( A \in \{g, h\} \) — with the probabilities of occurrence 0.5, 0.2 and 0.2, respectively. The rest 10% of queries select single attribute values randomly. To find a well-defined encoding for those three predefined selections, we may define the mapping table as Figure 6.1(a) shows. The idea can be easily illustrated by the Karnaugh map, shown in Figure 6.1(b).

Originally, the number of bitmaps of an encoded bitmap index on \( A \) is determined by \( \lceil \log_2(|A|) \rceil \), which is 3. By releasing this condition and increasing the number of bitmaps to 4, we can easily define a well-defined encoding by adding don’t-care conditions to proper positions, as Figure 6.1(a) shows. This can be systematically done by assigning “don’t-care” to the neighboring cells of some given cells to form a prime chain (defined in Section 3.1.2). By this example, we see that by increasing the
Figure 6.1: An encoding that releases the minimized space requirement and takes probabilities of query occurrence into account ($\times$ denotes don’t-care conditions)

number of bitmaps by 1, the number of bitmap scans needed to evaluate predicate “$A \in \{a, b\}$” is reduced from 2 to 1.

- Third, the compression technique — **Huffman coding** — introduced in the field of data communication can be exploited to define the encoding on the attribute domain [50], especially for those attributes on which single-value selections are the usual case. Huffman coding results in variable length encoding, instead of fixed length encoding used so far. The idea of how to exploit it for the encoding of an attribute is as follows. First, every value in the attribute domain is assigned a weight, e.g., the probability of occurrence or the probability of access. Then, the coding operation begins with constructing an unbalanced binary tree in a bottom-up manner. The pair of values with the smallest two weights are combined to form a sub-tree, and the weight of the sub-tree is the sum of the two values’ weights. Repeat combining two values, or sub-trees, until there is only one tree left. By assigning the left branches and the right branches all the way down to the leaf-nodes, we can obtain the codes for each value by traversing from the root down to the leaf-node. Since the operation begins with values with smaller weights, the values with the larger weights are encoded with fewer bits than those values with smaller weights. The average number of bitmap scans for evaluating single-value selections can be determined by

$$-\sum_{i=1}^{n} w_i \log_2 w_i,$$

which is known as the **entropy** per key [84], where $n$ is the number of values in the attribute domain and $w_i$ is the weight of the $i$-th value in the domain. This value is no larger than $\lfloor \log_2(\vert A\vert) \rfloor$, which is the number of bitmap scans while evaluating a single-value selection using an encoded bitmap index.

- In the literature, it has been shown that performance can be improved by orders of magnitude through query rewrites. In this work, a straightforward query revision technique based on referential integrity and functional dependencies is introduced to transform Star-Joins into unions of selections. However, only non-nested SQL statements are allowed in the current version of bitSQL. It will be a desirable extension in the future to include query rewriting rules which also take nested SQL statements into account. Nested SQL statements should first be transformed into joins, if possible, and the joins are further transformed to unions of selections on single table.

- Another technique that may reduce the storage requirements of bitmap indexes and the
time of reading bitmaps into memory is *compression*. However, compression techniques should be applied with care. Compressed bitmaps deform the address information of the indexed tuples. Therefore, bitmap algorithms and design criteria based on uncompressed bitmaps should be carefully re-examined before applying compression. If compressed bitmaps can be processed without uncompressing them, there will be performance gains with respect to both time and space. In the literature, algorithms that perform Boolean operations on compressed bitmaps have been proposed [61]. (Detailed performance analysis on compressed bitmaps can be found in [54].) However, bitmap-based query processing requires operations more than Boolean operations. For example, if data is striped horizontally on the fly, the compressed bitmaps should also be cut at the same position as data is partitioned.

- Last, the issue concerning *parallelism* and *distributed computing* has not been addressed in this work. Both bitmap processing and data fetching using bitmaps provide great opportunities to be parallelized. For example, in a multi-processor environment with bitmap indexes vertically striped among multiple disks, different selection predicates can be evaluated concurrently using bitmap indexes, followed by a logical combination of the resulting bitmaps. For another example, in a distributed environment data and its bitmap indexes can be striped horizontally on the fly to balance loads and to increase parallelism, since bitmaps do not lose the address information if data and bitmaps are partitioned consistently, *i.e.*, if the cuts are at the same position. Or, if data is by nature horizontally partitioned based on the time dimension, or the geographical dimension, the results of bitmap processing can be easily combined with union operations to construct the final results.
Appendix A

Proofs

A.1 Conformation of Equation (3.5) and (3.6)

Storage utilization of B-trees and their variants have been intensively studied in the past, since the performance of B-trees depends highly on the heights of the trees, which in turn are determined by their storage utilization factors. In this work, in order to compare B-trees with bitmap indexes, we have to apply results of former works about storage utilization of B-trees to analyze their performance. Here, in this appendix, we will show the conformation of different results from [23] and [57], before we apply one of the results in our analysis.

In [23], Chu and Knott have defined the storage utilization factor at leaf level as

$$\mu = \frac{\phi}{1 - \phi} \ln \frac{1}{\phi} + O\left(\frac{1}{M}\right),$$

where $\phi = \frac{|M_t + 1|}{M}$. For large value of $M$, the above equation conforms to the work done by Küspert [57], since

$$\lim_{M \to \infty} \left(\frac{\phi}{1 - \phi} \ln \frac{1}{\phi} + O\left(\frac{1}{M}\right)\right) = \lim_{M \to \infty} \frac{tM + 1}{M - 1} \cdot \ln \frac{tM + M}{tM + 1} + \lim_{M \to \infty} O\left(\frac{1}{M}\right) = t \cdot \ln \frac{t + 1}{t}.$$

A.2 Effects of Non-Homogeneous Splitting Factors in both Leaf and Non-leaf Nodes

In Section 3.3.2, both the split factors at leaf and non-leaf nodes are the same, i.e., $t = t^*$. Here, we explore the effect of releasing this assumption and explain our reading of it.

To reflect the different split factors at both non-leaf and leaf nodes, Equation (3.8) should be rewritten as

$$\frac{n}{M \cdot \mu} + \frac{(M^* \cdot \mu^*)^{h^* - 1} - 1}{M^* \cdot \mu^* - 1} = \frac{1}{M \cdot \mu} \left(n + \frac{n - M \cdot \mu}{M^* \cdot \mu^* - 1}\right) \quad (A.1)$$
where \( h^* = \log (M \cdot \mu^*) \left( \frac{1}{M \cdot \mu} \right) + 1 \). In Figure A.1, we show three different cases: the first one with \( t = t^* = 7 \), the second one with \( t = 7 \) (the split factor at leaf nodes) and \( t^* = 1 \) (the split factor at non-leaf nodes), and the third one with \( t = 1 \) and \( t^* = 7 \). We can see that the split factor at the leaf nodes play a dominating role for the storage utilization of a B-tree, while the split factor at the non-leaf nodes has hardly an influence on it.

![Figure A.1: Space Requirement analysis releasing \( \mu^* = \mu \)](image)

The result is reasonable, since for a large B-tree with all the data items stored at the leaf nodes, the utilization factor at the leaf level will drastically affect the height and the width of the tree.

### A.3 Best Case Analysis

In Section 3.3.2, we have used Equation (3.9) to approximate the space requirements of best-case B-trees, however, it underestimates the number of the non-leaf nodes slightly. It is difficult to have a precise approximation for average space requirements. Nonetheless, the best cases can be analyzed by the following equation.

\[
\frac{n}{M \cdot \mu} + \sum_{i=1}^{h-1} \left[ \frac{n}{M \cdot \mu} \cdot \left( \frac{1}{M \cdot \mu} \right)^i \right]
\]

(A.2)

The second term of Equation (A.2) computes the minimal number of non-leaf nodes in order to be able to hold \( \frac{n}{M \cdot \mu} \) nodes at the leaf level. In Figure A.2, the two functions overlap. It shows that Equation (3.9) is a very good approximation for the best cases of B-trees. In Table A.1 we list the results of Equation (3.9) and (A.2) and the errors with respect to different cardinality of indexed tables.

### A.4 Proof of Theorem 2

A well-defined encoding is an optimal encoding.

**Proof:** Let \( s = \{v_0, \ldots, v_{n-1}\} \), \( n \geq 2 \) be a subdomain of attribute \( A \), \( \log_2 |A| = k \) and \( M^A(\cdot) \) be a mapping on \( A \) and is well-defined.
Figure A.2: Best case analysis

I) Suppose that $|s| = 2^p$. Because $M^A(\cdot)$ is well-defined, therefore, there exists a chain in 
\{M^A(v) | v \in s\}, and \( \forall v, v' \in s, \lambda(M^A(v), M^A(v')) \leq p \), without loss of generality, let the
sequence on \{M^A(v) | v \in s\} be
\[ < M^A(v_0), M^A(v_1), \ldots, M^A(v_{2^p-1}) >, \]
and
\[
\begin{align*}
M^A(v_0) &= b_{k-1} \cdots b_{2^p-1} b_{2^p-2} \cdots b_0 = \lambda(M^A(v_0), M^A(v_0)) = 1 \\
M^A(v_1) &= b_{k-1} \cdots b_{2^p-1} b_{2^p-2} \cdots b_0 = \lambda(M^A(v_1), M^A(v_2)) = 1 \\
& \vdots \\
M^A(v_{2^p-3}) &= b_{k-1} \cdots b_{2^p-1} b_{2^p-2} \cdots b_0 = \lambda(M^A(v_{2^p-3}), M^A(v_{2^p-2})) = 1 \\
M^A(v_{2^p-1}) &= b_{k-1} \cdots b_{2^p-1} b_{2^p-2} \cdots b_0 \text{ (for all } p \text{ bits)} \\
& \Rightarrow \lambda(M^A(v_{2^p-1}), M^A(v_{2^p-1})) = 1
\end{align*}
\]

Note that, the bits, \( b_{2^p-1}, \ldots, b_0 \), are not necessarily continuous or adjacent to each other, and since \( \forall v, v' \in s, \lambda(M^A(v), M^A(v')) \leq p \) and a chain exists in \{M^A(v) | v \in s\}, we can be sure that all other bits, except \( b_{2^p-1}, \ldots, b_0 \), in \( M^A(v) \) for all \( v \in s \) are the same, i.e., the number of variant bits is restricted to \( p \). In other words, by logical reduction of the
Boolean expression, \( M^A(v_0) + \cdots + M^A(v_{2^p-1}) \), we will get
\[
\begin{align*}
&b_{k-1} \cdots (b_{2^p-1} b_{2^p-2} \cdots b_0 + \cdots + b_{2^p-1} b_{2^p-2} \cdots b_0 + b_{2^p-1} b_{2^p-2} \cdots b_0) \cdots b_0 \quad (A.3) \\
&2^p \text{ } p\text{-variable min-terms }
\end{align*}
\]

The sum of min-terms,
\[
(b_{2^p-1} b_{2^p-2} \cdots b_0 + \cdots + b_{2^p-1} b_{2^p-2} \cdots b_0 + b_{2^p-1} b_{2^p-2} \cdots b_0),
\]
is a tautology, since \( 2^p \) distinct min-terms must cover all the values in an \( p \)-variable truth
table, in addition, the equation (A.3) will be reduced to a \( (k-p) \)-variable Boolean function
\[
\begin{align*}
&b_{k-1} \cdots b_j b_h \cdots b_0, \\
&k-p \text{ variables }
\end{align*}
\]
Table A.1: Comparison of Equations (3.9) and (A.2), Error = (Approximation − Best Case), Error in % = \frac{Error}{Best Case}

i.e., to process the queries with selection “A \text{IN} \{w_0, \ldots, v_{n-1}\}”, (k-p) bitmap vectors are accessed.

According to Property 2, we know that for the best cases, $\lceil \log_2 |A| \rceil - \theta(\delta)$ bitmap vectors must be accessed. In our case, $\theta(\delta) = p$, $\lceil \log_2 |A| \rceil = k$, therefore,

$$\lceil \log_2 |A| \rceil - \theta(\delta) = k - p$$

The result follows.

II) If $2^p < n < 2^{p+1}$, and n is even

$$\exists \text{ a chain in } \{\mathcal{M}^A(v) | v \in s \} \text{ and } \forall v, v' \in s, \exists \lambda(\mathcal{M}^A(v), \mathcal{M}^A(v')) \leq p + 1$$

$$\text{the number of variant bits in } \mathcal{M}^A(v), \text{ for all } v \in s, \text{ is restricted to } p + 1,$$

i.e., Equation (A.3) can be rewritten as

$$b_{k-1} \cdots \left( b_{h_p} b_{h_p-1} b_{h_p-2} \cdots b_{h_0} + \cdots + b_{h_p} b_{h_p-1} b_{h_p-2} \cdots b_{h_0} + b_{h_p} b_{h_p-1} b_{h_p-2} \cdots b_{h_0} \right) \cdots b_0$$

(A.4)

In addition,

$$\exists s' \subset s, |s'| = 2^p, \text{ and there exists a prime chain in } \{\mathcal{M}^A(v) | v \in s' \}$$

$$\exists \text{ Those } 2^p \text{ min-terms (representing the set } s' \text{) from the } n \text{ min-terms (within the parenthesis in Equation (A.4)) can be reduced to a one-variable Boolean function}$$

Note that there are still $k - (p + 1)$ common variables in Equation (A.4), which should be added to the total number of variables in the final Boolean function.
Now, we have to determine how many variables the rest \((n - 2^p)\) min-terms contribute to the final retrieval function.

**II.1** Let \(|s - s'| = 2^{p-1} \ (1 < |s - s'| < 2^{p+1} - 2^p)\). According to the definition, we know that \(M^A(\cdot)\) is well-defined w.r.t. \((s - s')\). That is, there exist prime chains in both \(\{M^A(v)|v \in s - s'\}\) and \(\{M^A(v)|v \in s - s'\}\). Let us express the retrieval Boolean functions for \(s'\) and \(s - s'\) as the sums of min-terms,

\[
\sum m(s') = \sum m(M^A(v_{2^p-1}), \ldots, M^A(v_0)), \text{ where } s' = \{v_0, \ldots, v_{2^p-1}\},
\]

\[
\sum m(s - s') = \sum m(M^A(u_{2^p-1-1}), \ldots, M^A(u_0)), \text{ where } s - s' = \{u_0, \ldots, u_{2^p-1-1}\}
\]

Without loss of generality, let

\[
\sum m(s') = b_{k-1} \cdots b_p \left( \sum T(b_{p-1}, \ldots, b_0) \right) = b_{k-1} \cdots b_p,
\]

(A.5)

where \(\sum T(b_{p-1}, \ldots, b_0)\) denotes the sum of all \(p\)-variable fundamental conjunctions, i.e., \(\sum T(b_{p-1}, \ldots, b_0) = 1\). And,

\[
\sum m(s - s') = b_{k-1} \cdots b_{p+1} \overline{b}_p \left( \sum T(b_{p-1}, \ldots, b_1) \right) b_0,
\]

\((: \forall v, v' \in s, \ \lambda(M^A(v), M^A(v')) \leq p + 1, \ i.e., \ the \ number \ of \ variant \ bits \ in \ \{M^A(v)|v \in s\} \ is \ no \ larger \ than \ p + 1)\)

Then,

\[
\sum m(s') + \sum m(s - s') = b_{k-1} \cdots b_p \left( \sum T(b_{p-1}, \ldots, b_0) \right) + b_{k-1} \cdots b_{p+1} \overline{b}_p \left( \sum T(b_{p-1}, \ldots, b_1) \right) b_0
\]

Let \(F = (b_{k-1} \cdots b_{p+1})\), then the above equation is written as

\[
F \cdot b_p + F \cdot \overline{b}_p b_0, \text{ since } \sum T(b_{p-1}, \ldots, b_1) = 1, \text{ and } \sum T(b_{p-1}, \ldots, b_0) = 1
\]

\[
= F \cdot b_p + F \cdot b_0
\]

\[
= (b_{k-1} \cdots b_{p+1}) b_p + (b_{k-1} \cdots b_{p+1}) b_0
\]

Compare the above result with Equation (A.5), we see that the \(n - 2^p\) min-terms from \(s - s'\) contribute one more variable to the final retrieval Boolean function. In a word, the total number of variables in the final Boolean function is

\[
\text{number of common variables} + \underbrace{\text{two more variables by } s \text{ and } (s - s')}_{k - (p + 1) + 1 + 1} = k - p + 1
\]

According to Property 2, for the best cases, the number of bitmap vectors accessed is

\[
\lfloor \log_2 |A| \rfloor - \theta(\delta) = k - (p - 1),
\]

where \(\delta = n = 2^p + 2^{p-1}\) and \(\theta(2^p + 2^{p-1}) = p - 1\). The result follows.
II.2) Consider now the general case, where \(|s - s'| = 2^{p-\varepsilon}|\), where \(1 \leq \varepsilon < p\) and \(\varepsilon \in \mathbb{N}\). Let \(\sum m(s - s')\) denote the retrieval Boolean function for \((s - s')\).

\[
\sum m(s - s') = \sum m(M^A(u_{2^{p-\varepsilon} - 1}), \ldots, M^A(u_0)),
\]

where \(s-s' = \{u_0, \ldots, u_{2^{p-\varepsilon} - 1}\}\). Without loss of generality, \(\sum m(s - s')\) is expressed as follows.

\[
\sum m(s - s') = b_{k-1} \cdots b_p \cdot T(b_{p-1}, \ldots, b_1) \cdot b_0 \quad \text{(A.6)}
\]

where \(\sum T(b_{p-1}, \ldots, b_1) = 1\). (\(\forall v, v' \in s\), \(\lambda(M^A(v), M^A(s')) \leq p+1\), i.e., the number of variant bits in \(\{M^A(v) | v \in s\}\) is no larger than \(p+1\) )

Then,

\[
\sum m(s') + \sum m(s - s') = b_{k-1} \cdots b_p \cdot (\sum T(b_{p-1}, \ldots, b_1)) + b_{k-1} \cdots b_{p+1} \cdot \sum T(b_{p-1}, \ldots, b_1) \cdot b_0.
\]

Let \(F = (b_{k-1} \cdots b_{p+1})\), then the above equation is written as

\[
F \cdot b_p + F \cdot b_{p-1} \cdots b_{p+1} \cdot b_0
\]

\[
= (b_{k-1} \cdots b_{p+1}) \cdot b_p + (b_{k-1} \cdots b_{p+1}) \cdot b_{p+1} b_0.
\]

Compare the above result with Equation (A.5), we see that the \(n - 2^p\) min-terms from \(s - s'\) contribute \(\varepsilon\) more variable(s) to the final retrieval Boolean function. In a word, the total number of variables in the final Boolean function is

\[
\text{number of common variables} \cdot (k - (p+1)) + 1 + \varepsilon = k - p + \varepsilon
\]

According to Property 2, for the best cases, the number of bitmap vectors accessed is

\[
|\log_2 |A| - \theta(\delta)| = k - (p - \varepsilon),
\]

where \(\delta = n = 2^p + 2^{p-\varepsilon}\) and \(\theta(2^p + 2^{p-\varepsilon}) = p - \varepsilon\). The result follows.

II.3) For those cases, \(\|s - s'| \neq 2^{p-\varepsilon}\) \((1 \leq \varepsilon < p)\), \(\exists q \in \mathbb{N}\), \(1 \leq q < p\). \(\exists 2^{p-q} < |s - s'| < 2^{p-q+1}\).

According to II.2) some \(2^{p-q}\) min-terms from \(\sum m(s - s')\) can be reduced, and its contribution to the number of bitmap vectors access is \(q\). However, the contribution of \(\sum m(s - s')\) is more than \(q\), since there are still some \(|s - 2^{p-q} < 2^{p- q+1}\) min-terms not being considered.

If \(|s - 2^{p-2^{p-q}}| = 2^{p-q'} (q' < q < p)\), then again according to II.2) its contribution should be \(q' - q\). If not, i.e., \(2^{p-q} < |s| - 2^{p-q} < 2^{p-q'+1}\), then the above process will be repeated until \(|s| - 2^{p-q} - 2^{p-q'} - \cdots = 2^{p-q^{(n)}}, q^{(n-1)} < q^{(n)} < p\). Then, the total contribution of \(\sum m(s - s')\) to the number of bitmap vectors accessed is \(q + (q' - q) + (q'' - q') + \cdots + (q^{(n)} - q^{(n-1)}) = q^{(n)}\). That is, totally, there are

\[
\text{number of common variables} \cdot (k - (p+1)) + 1 + q^{(n)} = k - p + q^{(n)}
\]
According to Property 2,

\[
\therefore \quad \delta = |s| = 2^p + 2^{p-q} + 2^{p-q'} + \ldots + 2^{p-q^{(n)}}
\]

\[
\implies \quad \theta(\delta) = p - q^{(n)}
\]

\[
\implies \quad [\log_2 |A|] - \theta(\delta) = k - (p - q^{(n)})
\]

It confirms the result above.

III) If \(|s| = n\), and \(n\) is odd, then no matter how the encoding is, all the bitmaps need to be accessed during query processing. Therefore, we do not have to prove that the well-defined encoding is the optimal encoding. However, a well-defined encoding avoids any isolated cell in the Karnaugh-map, which provides more potentials for optimization at run time.

\[\Box\]

### A.5 Proof of the Correctness of Query Rewrites

In Section 4.1.2, we propose a query rewrite technique to transform Group By clauses on dimension table(s) to grouping on the fact table such that Star-Joins are redundant and can be omitted. We claim that if functional dependencies and referential integrities are satisfied, the correctness of the rewrites is guaranteed. However, if functional dependency or referential integrity is violated, it does not imply the dissatisfaction of the correctness. Let us show the above statement by the following simplified example. Suppose that we have three tables \(R, S\) and \(T\). The schema of \(R\) is \(R(A, C)\); the schema of \(S\) is \(S(A, B)\), and the schema of \(T\) is \(T(B, D)\). Attribute \(A\) is the key of \(S\) and a foreign-key from \(R\) to \(S\), and attribute \(B\) is the key of \(T\) and a foreign-key from \(S\) to \(T\). Attribute \(A.C\) is an additive attribute. We want to prove the following statement.

Referential integrities between keys and foreign-keys and functional dependencies among keys and non-keys are satisfied

\[
\Rightarrow \quad \sigma_{A \in \pi_A(\sigma_{D \bowtie_B T})} R = \pi_H(R) \sigma_D(R \bowtie_A S \bowtie_B T)
\]

We provide a proof to the above statement by contradiction. That is, if we can show that \((\rho \land \neg q) \equiv FALSE\), then \(\rho\) logically implies \(q\), denoted by \(\rho \Rightarrow q\). Let \(\rho\) and \(q\) be given as

\[
\rho : \quad \rho_1 \land \rho_2
\]

\[
\rho_1 : \quad \text{referential integrities between keys and foreign-keys are satisfied}
\]

\[
\rho_2 : \quad \text{functional dependencies among keys and non-keys are satisfied}
\]

\[
q : \quad \sigma_{A \in \pi_A(\sigma_{D \bowtie_B T})} R = \pi_H(R) \sigma_D(R \bowtie_A S \bowtie_B T)
\]

Now let \(\rho\) be TRUE, i.e., both referential integrities and functional dependencies are satisfied, and let \(q\) be FALSE, i.e., \(\sigma_{A \in \pi_A(\sigma_{D \bowtie_T})} R \neq \pi_H(R) \sigma_D(R \bowtie_A S \bowtie_T)\). For the sake of simplicity and clarity, we omit the subscripts of Joins.
According to the definition of referential integrity, we know that
\[ \forall r \in R, \exists s \in S, \exists t. A = s.A \]
\[ \forall s \in S, \exists t \in T, \exists s.B = t.B \]
In addition, by the definition of functional dependencies, we have
\[ B \rightarrow D \]
\[ A \rightarrow B \]
\[ A \rightarrow D \]
(Note that \( \Rightarrow \) denotes logical implication, and \( \rightarrow \) denotes functional dependency.) Besides, because \( A \) is the key of \( S \), and \( B \) is the key of \( T \), the referential integrities among \( R, S \) and \( T \) are revised as
\[ \forall r \in R, \exists \text{ exactly one } s \in S, \exists t. A = s.A \]
\[ \forall s \in S, \exists \text{ exactly one } t \in T, \exists s.B = t.B \]
Now we begin with the proof. The fact that \( \sigma_{A \in \pi_A(\sigma_{D \bowtie T} R)} R \neq \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T) \) can be divided into two sub-cases.

**case 1):** \( \exists \tau \in R, \exists \tau \in \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T), \) \hspace{1cm} (A.7)
and \( \tau \notin \sigma_{A \in \pi_A(\sigma_{D \bowtie T} R)} R \) \hspace{1cm} (A.8)

**case 2):** \( \exists \tau \in R, \exists \tau \in \pi_{A \in \pi_A(\sigma_{D \bowtie T})} R, \) \hspace{1cm} (A.9)
and \( \tau \notin \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T) \) \hspace{1cm} (A.10)

From Statement (A.7) we get
\[ \exists \varphi \in \sigma_{D}(S \bowtie T), \exists \varphi.A = \tau.A \]

However, it contradicts Statement (A.8), since from Statement (A.8), we know
\[ \forall \varphi \in \sigma_{D}(S \bowtie T), \exists \varphi.A \neq \tau.A \]
\[ \therefore \exists \tau \in R, \exists \tau \in \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T) \text{ and } \tau \notin \sigma_{A \in \pi_A(\sigma_{D \bowtie T})} R \]
\[ \Rightarrow \sigma_{A \in \pi_A(\sigma_{D \bowtie T})} R \supseteq \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T) \] \hspace{1cm} (A.11)

From Statement (A.10) we have
\[ \exists \varphi \in \sigma_{D}(S \bowtie T), \exists \tau.A = \varphi.A, \]

but this contradicts Statement (A.9) since
\[ \forall \varphi \in \sigma_{D}(S \bowtie T), \text{ if } \varphi.A \neq \tau.A, \text{ then } \tau \notin \sigma_{A \in \pi_A(\sigma_{D \bowtie T})} R \]
\[ \Rightarrow \sigma_{A \in \pi_A(\sigma_{D \bowtie T})} R \subseteq \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T) \] \hspace{1cm} (A.12)

By Statements (A.11) and (A.12), we conclude that
\[ \sigma_{A \in \pi_A(\sigma_{D \bowtie T})} R = \pi_{H(R)} \sigma_{D}(R \bowtie S \bowtie T) \]
However, we do not have the proof yet since the above inferences are based on the set theory, but tables are implemented in relational databases as multi-sets.\(^1\) Now, let us replace all the operands in the above inferences with multi-sets. In order to keep using set operators but at the same time to treat duplicates in multi-sets as conceptually different elements, we use the notations — \(\tau, \tau', \ldots, \tau^{(n)}\) \((n \in \mathbb{N})\) to denote \(n\) duplicates of \(\tau\).

The above result \(\sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R = \pi_H(R) \sigma_D(R \bowtie S \bowtie T)\) are still true if the results on both sides of the equal sign are sets. In other words, if the equality is dissatisfied, the relations on both sides of the equal sign can only differ from each other in either of the following ways:

1. \[
\exists \tau, \tau' \in \sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R, \tau \in \pi_H(R) \sigma_D(R \bowtie S \bowtie T), \text{ but } \tau' \notin \pi_H(R) \sigma_D(R \bowtie S \bowtie T)
\]

2. \[
\exists \tau, \tau' \in \pi_H(R) \sigma_D(R \bowtie S \bowtie T), \tau \in \sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R, \text{ but } \tau' \notin \sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R
\]

For case 1),
\[
\therefore \tau, \tau' \in \sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R \\
\Rightarrow \tau. A, \tau'. A \in \pi_A(\sigma_D S \bowtie T), \text{ and } \tau, \tau' \in R
\]

According to the definition of Joins, we know
\[
\pi_H(R) (\sigma_D S \bowtie T) \bowtie R \supseteq \{\tau, \tau'\}
\]

It contradicts \(\tau' \notin \pi_H(R) \sigma_D(R \bowtie S \bowtie T)\), i.e., case 1) does not exist. For case 2), since \(\tau'\) is a duplicate of \(\tau\),
\[
\tau \in \sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R, \text{ and } \tau' \notin \sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R \\
\Rightarrow \tau \in R, \text{ and } \tau' \notin R
\]

In addition, because \(\tau, \tau' \in \pi_H(R) \sigma_D(R \bowtie S \bowtie T)\),
\[
\Rightarrow \exists \varphi, \phi \in \sigma_D(S \bowtie T), \exists \varphi \neq \phi \text{ and } \varphi. A = \phi. A = \tau'. A \\
(\varphi \text{ can be a duplicate of } \phi, \text{ or vice versa, but } \varphi \text{ and } \phi \text{ refer to different tuples})
\]
\[
\therefore A \rightarrow B
\]
\[
\therefore \varphi. B = \phi. B \text{ (otherwise functional dependency between } A \text{ and } B \text{ is violated)}
\]

However, since \(B\) is the key of \(T\), and \(A\) is the key of \(S\),
\[
\Rightarrow \exists \text{ exactly one } t \in T, \exists \ t. B = \varphi. B = \phi. B \text{ and } \\
\Rightarrow \exists \text{ exactly one } s \in S, \exists \ s. A = \varphi. A = \phi. A
\]

According to the definition of Joins, \(\{s\} \bowtie \{t\}\) cannot result in a relation with more than one tuple, thus \(\varphi\) and \(\phi\) must refer to the same tuple. Consequently, for all duplicates of \(\tau\) (if any), if they exist in \(\pi_H(R) \sigma_D(R \bowtie S \bowtie T)\), they must also exist in \(\sigma_{A \in \pi_A(\sigma_D(S \bowtie T))} R\).

---

\(^1\)The multi-value sets discussed in the previous section are sets. The only difference is that their elements are multi-values, instead of single values. However, multi-sets are actually no longer sets; they are also called bags in the literature. Duplicates are allowed in multi-sets and are treated individually. Duplicates make two multi-sets different, e.g., \(\{a, a\} \neq \{a\}\). For the sake of simplicity, we still use the notations of the set theory for multi-sets, but they will be noted explicitly.
From the above discussions, we prove that even if the operands of the relational operators are multi-sets,

\[(\sigma_{A \in \pi_A(\sigma_D \bowtie T)} R = \pi_H(R) \sigma_D(R \bowtie S \bowtie T)) \equiv \text{TRUE}\]

Finally, we can conclude that

\[\neg q \Rightarrow \neg p\]
\[\Rightarrow (p \land \neg q) \equiv \text{FALSE}\]
\[\Rightarrow p \Rightarrow q\]

In general cases, we are also interested to know whether or not \(R = \pi_H(R) (R \bowtie S \bowtie T)\), without the effects of selection conditions. This scenario occurs, for example in cases of scalar aggregates. Analogous to the discussion above, if we can show the truth of the equality, we can use \(R\) instead of \(\pi_H(R) (R \bowtie S \bowtie T)\), i.e., the joins are redundant.

Following the above discussion, we let \(q'\) be \(R = \pi_H(R) (R \bowtie S \bowtie T)\). If we can prove that \((p \land \neg q') \equiv \text{FALSE}\), then we can say \(p \Rightarrow q'\). Let \(\neg q'\) be \(\text{TRUE}\), i.e., \(R \neq \pi_H(R) (R \bowtie S \bowtie T)\). There are two cases which make \(\neg q' \equiv \text{TRUE}\):

- case 1) \(\exists \tau, \tau \in \pi_H(R) (R \bowtie S \bowtie T)\), and \(\tau \notin R\)
- case 2) \(\exists \tau, \tau \in R\), and \(\tau \notin \pi_H(R) (R \bowtie S \bowtie T)\)

Case 1) is a contradiction. If \(\tau \notin R\), then \(\tau \notin \pi_H(R) (R \bowtie S \bowtie T)\). It is because Join operations will not join tuples that do not exist in the operand tables. For case 2), if there does exist a tuple, \(\tau\), and \(\tau \in R\) and \(\tau \notin \pi_H(R) (R \bowtie S \bowtie T)\), then

\[\Rightarrow \exists s \in S, \exists \tau.A = s.A, \text{ or } \forall s \in S, \exists t \in T, \exists \tau.A = s.A \text{ and } s.B = t.B\]

This contradicts the definition of referential integrities among \(R, S\) and \(T\). Thus, we prove that

\[\neg q' \Rightarrow \neg p\]
\[\Rightarrow p \Rightarrow q'\]

To summarize, referential integrities ensure the correctness of query rewrites if scalar aggregates are queried, while functional dependencies along the join paths to the Group By attribute(s) guarantee the correctness of the revision in general cases. In Data Warehousing, since Joins are mostly in form of Star-Joins, i.e., joins between keys and foreign-keys, the rewriting technique proposed in the previous section provides an easy and effective way to skip costly join operations and to evaluate aggregate functions directly on the fact table(s). The failure of referential integrities or functional dependencies does not directly imply the failure of rewrites. However, to ensure the correctness of rewrites under such circumstances, the approach needs to adopt multi-set operators and to include artificial filters for exclusion of dangling references.
A.6 Complexity Analysis of Algorithms 2 and 3

The number of bitmap scans by both Algorithm 2 and 3 are functions of the numbers of distinct digits in all components. In Algorithm 3, for each distinct digit, one bit vector is scanned. As a result, the time function of Algorithm 3 can be defined as counting the number of different digits in all components.

One important factor affecting the counting process is the distribution of the digits within their ranges. By assuming that the numbers are evenly distributed in their domains, and the probability of the occurrence of one value is independent from that of another value, we can derive the time complexity as follows.

Given a set of numbers \( V = \{v_1, \ldots, v_k\} \) and a base \(<b_N, \ldots, b_1>\), then any value \(v_j \in V\) is decomposed and rewritten as \(v_{j,N} \cdots v_{j,1} (1 \leq j \leq k, 0 \leq v_{j,i} < b_i, 1 \leq i \leq N)\). Now, we wish to know how high is the probability that the number of distinct digits in \(\{v_{1,1}, v_{2,1}, \ldots, v_{k,1}\}\) (1 \(\leq i \leq N\)) is equal to \(l_i\), 1 \(\leq l_i \leq b_i\). Then, the summation of the products of the probabilities and \(l_i\)’s is the expected number of bit vectors which are required to be read in query processing.

\[
\sum_{i=1}^{N} \sum_{l_i=1}^{b_i} l_i \cdot \text{prob}(l_i) \tag{A.13}
\]

We use \(\text{prob}(l_i)\) to denote the probability that the number of distinct digits in \(\{v_{1,1}, v_{2,1}, \ldots, v_{k,1}\}\) is equal to \(l_i\), where 1 \(\leq l_i \leq b_i\). To compute \(\text{prob}(l_i)\), let us transform the problem into a problem of combination with repetitions. Suppose that we have \(b_i\) numbered balls, and we wish to choose \(k\) balls from them with repetition. There will be totally

\[
\binom{k+b_i-1}{k}
\]

different kinds of combinations. Among the chosen \(k\) balls, let the number of distinct balls be \(k-j\) (1 \(\leq k-j \leq b_i\)), the scenario is equivalent to that we first choose \(k-j\) balls from \(b_i\) numbered balls without repetition, and then select \(j\) balls from the \(k-j\) balls with repetition to make the total number of selected balls equal to \(k\). There are \(\binom{b_i}{k-j}\) different ways of picking \(k-j\) balls from \(b_i\) balls without repetition, and \(\binom{k-j+j-1}{j-1}\) different ways of choosing \(j\) balls from \(k-j\) balls with repetitions. As a result, the probability of \(k-j\) distinct balls being selected is

\[
\frac{\binom{k-j+j-1}{j-1} \cdot \binom{b_i}{k-j}}{\binom{k+b_i-1}{k}}.
\]

The sum of the probability that \(k-j = 1, 2, \ldots, b_i\) should be proven to be 1. However, for
We claim that if Inequality (A.14) is satisfied, then the range $0 \leq j \leq k-1$.

\[
\sum_{j=0}^{k-1} C^{(k-j)+j-1}_{k-j} \cdot b_j^{b_k} = \frac{b_1!}{k!(b_1-k)!} + \frac{(k-1)b_1!}{(b_1-k+1)!(k-1)!} + \cdots + \frac{b_1!}{(b_1-1)!} = \frac{b_1!(b_1-1) \cdots (b_1-k+1) + (k-1)b_1!(b_1-k) \cdots (b_1-k+2) + \cdots + k!b_1!}{k!(b_1-k)!} = \frac{b_1!(b_1+1)(b_1+2) \cdots (b_1+k-1)}{k!(b_1-1)!} = \frac{(b_1+k-1)!}{k!(b_1-1)!} = C^{b_k+k-1}_{k} \Rightarrow \sum_{j=0}^{k-1} C^{(k-j)+j-1}_{k-j} \cdot b_j^{b_k} = 1
\]

Now, the expected number of bitmap scans can be defined by

\[
\sum_{i=1}^{N} \sum_{l_i=1}^{b_i} l_i \cdot \text{prob}(l_i) = \sum_{i=1}^{N} \sum_{j=0}^{k-1} \frac{1}{C^{b_k+k-1}_{k}} \left( (k-j) \cdot C^{(k-j)+j-1}_{j} \cdot b_j^{b_k} \right)
\]

As for range bit-encoding indexes, since for each digit two bit vectors are scanned and the total number of bitmap scans is upper-bounded by the number of bit vectors, the expected number of bitmap scans is

\[
\sum_{i=1}^{N} \sum_{l_i=1}^{b_i} l_i \cdot \text{prob}(l_i) = \sum_{i=1}^{N} \sum_{j=0}^{k-1} \frac{1}{C^{b_k+k-1}_{k}} \left( \min(2(k-j), b_i) \cdot C^{(k-j)+j-1}_{j} \cdot b_j^{b_k} \right)
\]

\[\blacksquare\]

### A.7 Proofs of Property 3 and Algorithm 8

In Algorithm 8, the following inequality is used to compare two numbers based on summation of partial digits.

\[
2^{k-h} \cdot (\text{sum}[\text{seq}[i]] - \text{sum}[\text{seq}[i-1]]) > c_{\text{seq}[k-1] \cdot (2^{k-h} - 1)}
\]  
(A.14)

We claim that if Inequality (A.14) is satisfied, then the seq[i]-th group’s sales summary is larger than that of the seq[i-1]-th group. To prove this statement, let us first define the following notations. Given are two sets of positive numbers, \( \{A_1, \ldots, A_m\} \) and \( \{B_1, \ldots, B_n\} \). Suppose that each number is expressed by \( k \) decimal digits, where \( k = \lceil \log_{10} \max(A_1, \ldots, A_m, B_1, \ldots, B_n) \rceil + 1 \). Then, the two sets of numbers can be expressed as follows.

\[
A_i = A_i,k-1 A_i,k-2 \cdots A_i,0 \quad B_i = B_i,k-1 B_i,k-2 \cdots B_i,0
\]

\[
\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots
\]

\[
A_m = A_m,k-1 A_m,k-2 \cdots A_m,0 \quad B_n = B_n,k-1 B_n,k-2 \cdots B_n,0
\]
Now, we want to prove the following property.

**Property 4**

\[
10^{k-1} \cdot \left( \sum_{i=1}^{m} A_{i,k-1} - \sum_{j=1}^{n} B_{j,k-1} \right) > n \cdot (10^{k-1} - 1) \quad \implies \quad \sum_{i=1}^{m} A_{i} > \sum_{j=1}^{n} B_{j}
\]

**Proof:** Without loss of generality, let

\[
10^{k-1} \cdot \left( \sum_{i=1}^{m} A_{i,k-1} - \sum_{j=1}^{n} B_{j,k-1} \right) > n \cdot (10^{k-1} - 1)
\]

Since all decimal digits \(B_{j,k-1}, \ldots, B_{j,0} (1 \leq j \leq n)\) are smaller than or equal to 9, then

\[
\sum_{j=1}^{n} \left( 9 \cdot 10^{k-2} + 9 \cdot 10^{k-3} + \cdots + 9 \right) \geq \sum_{j=1}^{n} \left( B_{j,k-2} \cdot 10^{k-2} + B_{j,k-3} \cdot 10^{k-3} + \cdots + B_{j,0} \right)
\]

In addition,

\[
\therefore \quad \sum_{j=1}^{n} \left( 9 \cdot 10^{k-2} + 9 \cdot 10^{k-3} + \cdots + 9 \right) = 10^{k-2} \cdot \sum_{j=1}^{n} \left( 9 + \frac{9}{10} + \cdots + \frac{9}{10^{k-2}} \right) = 10^{k-2} \cdot 9 \cdot \frac{1 - \frac{1}{10^{k-1}}}{1 - \frac{1}{10}} = n \cdot (10^{k-1} - 1)
\]

\[
\therefore \quad n \cdot (10^{k-1} - 1) \geq \sum_{j=1}^{n} \left( B_{j,k-2} \cdot 10^{k-2} + B_{j,k-3} \cdot 10^{k-3} + \cdots + B_{j,0} \right)
\]

\[
\therefore \quad 10^{k-1} \cdot \left( \sum_{i=1}^{m} A_{i,k-1} - \sum_{j=1}^{n} B_{j,k-1} \right) > \sum_{j=1}^{n} \left( B_{j,k-2} \cdot 10^{k-2} + B_{j,k-3} \cdot 10^{k-3} + \cdots + B_{j,0} \right)
\]

\[
\therefore \quad 10^{k-1} \left( \sum_{i=1}^{m} A_{i,k-1} - \sum_{j=1}^{n} B_{j,k-1} \right) + \sum_{i=1}^{m} \left( A_{i,k-2} \cdot 10^{k-2} + \cdots + A_{i,0} \right) > \sum_{j=1}^{n} \left( B_{j,k-2} \cdot 10^{k-2} + \cdots + B_{j,0} \right)
\]

\[
\therefore \quad \sum_{i=1}^{m} \left( A_{i,k-1} \cdot 10^{k-1} + \cdots + A_{i,0} \right) > \sum_{j=1}^{n} \left( B_{j,k-1} \cdot 10^{k-1} + \cdots + B_{j,0} \right)
\]

\[
\therefore \quad \sum_{i=1}^{m} A_{i} > \sum_{j=1}^{n} B_{j}
\]

We have the proof. \(\blacksquare\)

Property 4 is useful, since by summing up \(A_{i,k-1} (1 \leq i \leq m)\) and \(B_{j,k-1} (1 \leq j \leq n)\) only, we are able to tell whether \(\sum_{i=1}^{m} A_{i} > \sum_{j=1}^{n} B_{j}\), or not. In database systems, it means that we only
need to read some partial data and could be able to tell the ordering between numbers. For example, if bit slices are built on an attribute qty which needs to be aggregated and grouped, instead of scanning the whole table, bit slices are read (beginning with the most significant bit) and cumulated to form the partial summaries of qty of different groups. As long as the ordering among the aggregates of different groups can be established, the scanning is stopped.

Note that in Property 4, we only sum up the $k$-th digit (the most significant digit) of $A_i$ ($1 \leq i \leq m$) and $B_j$ ($1 \leq j \leq n$) to test the inequality. If, however,

$$10^{k-1} \cdot \left( \sum_{i=1}^{m} A_{i,k-1} - \sum_{j=1}^{n} B_{j,k-1} \right) \neq n \cdot (10^{k-1} - 1),$$

it does not imply

$$\sum_{i=1}^{m} A_i \leq \sum_{j=1}^{n} B_j.$$

In this case, we can further sum up the $(k-1)$-th digits of $A_i$ and $B_j$, and revise it to Property 5.

**Property 5**

$$10^{k-2} \cdot \left( \sum_{i=1}^{m} (A_{i,k-1} \cdot 10 + A_{i,k-2}) - \sum_{j=1}^{n} (B_{j,k-1} \cdot 10 + B_{j,k-2}) \right) > n \cdot (10^{k-2} - 1) \Rightarrow \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j$$

The same procedure can be repeated by summing up one more digit each time until the ordering between $\sum_{i=1}^{m} A_i$ and $\sum_{j=1}^{n} B_j$ can be told. Property 6 describes the general cases.

**Property 6** For $1 \leq h \leq k$,

$$10^{k-h} \cdot \left( \sum_{i=1}^{m} (A_{i,k-1} \cdot 10^{h-1} + \cdots + A_{i,k-h}) - \sum_{j=1}^{n} (B_{j,k-1} \cdot 10^{h-1} + \cdots + B_{j,k-h}) \right) > n \cdot (10^{k-h} - 1)$$

$$\Rightarrow \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j$$

Obviously, for the worst case, we have to read all the digits of all numbers in order find out the ordering among the numbers, i.e., the case where $h = k$. The proof to Property 6 is as follows.

**Proof:** The property can be proven using mathematical induction. Let $h = 1$, from Property (4) we know that

$$10^{k-1} \cdot \left( \sum_{i=1}^{m} A_{i,k-1} - \sum_{j=1}^{n} B_{j,k-1} \right) > n \cdot (10^{k-1} - 1) \Rightarrow \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j$$

$$\sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j.$$
Let \( h = r \), and assume that the following statement is true for \( 1 \leq r < k \).

\[
10^{k-r} \cdot \left( \sum_{i=1}^{m} (A_{i,r-1} \cdot 10^{r-1} + \cdots + A_{i,k-r}) - \sum_{j=1}^{n} (B_{j,r-1} \cdot 10^{r-1} + \cdots + B_{j,k-r}) \right) > n \cdot (10^{k-r} - 1)
\]

\[\implies \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j\]

Now, consider \( h = r + 1 \), we get

\[
10^{k-r-1} \cdot \left( \sum_{i=1}^{m} (A_{i,r} \cdot 10^{r} + \cdots + A_{i,k-r}) - \sum_{j=1}^{n} (B_{j,r} \cdot 10^{r} + \cdots + B_{j,k-r}) \right) > n \cdot (10^{k-r-1} - 1)
\]

\[\implies n \cdot (10^{k-r-1} - 1) = 10^{k-r-2} \sum_{j=1}^{n} (9 + \frac{9}{10} + \cdots + \frac{9}{10^{k-r-2}})
\]

\[\geq \sum_{j=1}^{n} (B_{j,r-2} \cdot 10^{k-r-2} + \cdots + B_{j,0})
\]

\[\implies 10^{k-r-1} \cdot \left( \sum_{i=1}^{m} (A_{i,r-1} \cdot 10^{r} + \cdots + A_{i,k-r}) - \sum_{j=1}^{n} (B_{j,r-1} \cdot 10^{r} + \cdots + B_{j,k-r}) \right) > \sum_{j=1}^{n} (B_{j,r-2} \cdot 10^{k-r-2} + \cdots + B_{j,0})
\]

\[\implies \sum_{i=1}^{m} (A_{r-1} \cdot 10^{k-1} + \cdots + A_{k}) > \sum_{j=1}^{n} (B_{j,r-1} \cdot 10^{k-1} + \cdots + B_{j,0})
\]

\[\implies \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j
\]

By the principle of mathematical induction, we have proven the cases where \( 1 \leq h < k \). We show now the boundary case is also true. Let \( h = k \), we have

\[
10^{k-k} \cdot \left( \sum_{i=1}^{m} (A_{i,k-1} \cdot 10^{k-1} + \cdots + A_{i,k-k}) - \sum_{j=1}^{n} (B_{j,k-1} \cdot 10^{k-1} + \cdots + B_{j,k-k}) \right) > n \cdot (10^{k-k} - 1)
\]

\[\implies \sum_{i=1}^{m} (A_{k-1} \cdot 10^{k-1} + \cdots + A_{k}) - \sum_{j=1}^{n} (B_{j,k-1} \cdot 10^{k-1} + \cdots + B_{j,0}) > 0
\]

\[\implies \sum_{i=1}^{m} A_i > \sum_{j=1}^{n} B_j
\]

We have the proof.
Appendix B

An Example of Non-Binary Encoded Bitmap Index

Suppose that given is an attribute $A$ with domain “^[A-C][0-2][0-2]$” (expressed in regular expression), i.e., $|A| = 27$. In addition, assuming that the following frequent range searches are predefined — $A$ LIKE “A?1”, and $A$ LIKE “C2?”. Then, the encoding defined in Table B.1(a) is well-defined.

<table>
<thead>
<tr>
<th>key</th>
<th>encoded value</th>
<th>key</th>
<th>encoded value</th>
<th>key</th>
<th>encoded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleted</td>
<td>$37_{&lt;4,8&gt;}$</td>
<td>NULL</td>
<td>$36_{&lt;4,8&gt;}$</td>
<td>C00</td>
<td>$22_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A00</td>
<td>$03_{&lt;4,8&gt;}$</td>
<td>B00</td>
<td>$11_{&lt;4,8&gt;}$</td>
<td>C01</td>
<td>$23_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A01</td>
<td>$00_{&lt;4,8&gt;}$</td>
<td>B01</td>
<td>$12_{&lt;4,8&gt;}$</td>
<td>C02</td>
<td>$24_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A02</td>
<td>$04_{&lt;4,8&gt;}$</td>
<td>B02</td>
<td>$13_{&lt;4,8&gt;}$</td>
<td>C10</td>
<td>$25_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A10</td>
<td>$05_{&lt;4,8&gt;}$</td>
<td>B10</td>
<td>$14_{&lt;4,8&gt;}$</td>
<td>C11</td>
<td>$26_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A11</td>
<td>$01_{&lt;4,8&gt;}$</td>
<td>B11</td>
<td>$15_{&lt;4,8&gt;}$</td>
<td>C12</td>
<td>$27_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A12</td>
<td>$06_{&lt;4,8&gt;}$</td>
<td>B12</td>
<td>$16_{&lt;4,8&gt;}$</td>
<td>C20</td>
<td>$30_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A20</td>
<td>$07_{&lt;4,8&gt;}$</td>
<td>B20</td>
<td>$17_{&lt;4,8&gt;}$</td>
<td>C21</td>
<td>$31_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A21</td>
<td>$02_{&lt;4,8&gt;}$</td>
<td>B21</td>
<td>$20_{&lt;4,8&gt;}$</td>
<td>C22</td>
<td>$32_{&lt;4,8&gt;}$</td>
</tr>
<tr>
<td>A22</td>
<td>$10_{&lt;4,8&gt;}$</td>
<td>B22</td>
<td>$21_{&lt;4,8&gt;}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) The mapping table

<table>
<thead>
<tr>
<th>Component 2</th>
<th>Component 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_3 b_2 b_1$</td>
<td>$b_6 b_5 b_4 b_3 b_2 b_1 b_0$</td>
</tr>
<tr>
<td>A00</td>
<td>1 1 1</td>
</tr>
<tr>
<td>C20</td>
<td>0 0 0</td>
</tr>
<tr>
<td>B00</td>
<td>1 1 0</td>
</tr>
<tr>
<td>A01</td>
<td>1 1 1</td>
</tr>
<tr>
<td>A11</td>
<td>1 1 1</td>
</tr>
<tr>
<td>C22</td>
<td>0 0 0</td>
</tr>
<tr>
<td>C21</td>
<td>0 0 0</td>
</tr>
<tr>
<td>A21</td>
<td>1 1 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

(b) Encoded bitmaps with base $<4,8>$ and range $\leq$ bit-encoding

Table B.1: A non-binary encoded bitmap index

199
To evaluate $A$ LIKE "A?1", the selection predicate is first rewritten to

$$A \text{ IN } \{00_{<4,8>}, 01_{<4,8>}, 02_{<4,8>}\}$$

$$\implies A \leq 02_{<4,8>}$$

by substituting the attribute values with their encoded ones. Then, it will be evaluated by the following bitmap operation

$$b_0^2 b_1^1.$$  

Similarly, to evaluate $A$ LIKE "C2?", the selection predicate is first rewritten to

$$A \text{ IN } \{30_{<4,8>}, 31_{<4,8>}, 32_{<4,8>}\}$$

$$\implies 30_{<4,8>} \leq A \leq 32_{<4,8>}$$

by substituting the attribute values with their encoded ones. It can be evaluated by the following bitmap operation

$$\overline{b_0^2} b_1^1.$$
Appendix C

Algorithms and Design Considerations

C.1 Chan & Ioannidis’ RangeEval-Opt Algorithm

RangeEval-Opt

Input: A bit-sliced index with the base, $< b_n, \ldots, b_1 >$, where $n$ is the number of components and $b^i_j$ denotes the $j$-th bit vector of $i$-th component.
Selection predicate $A \ op v$, where $\ op \in \{<, >, \leq, \geq, =, \neq\}$.
$B_n$ is a bitmap for tuples with non-null values.

Output: A bitmap vector representing the set of tuples which satisfy the selection predicate, $A \ op v$.

1) Begin
2) let $B = 1$
3) if ($op \in \{<, \geq\}$) then $v = v - 1$
4) $v = v_n v_{n-1} \ldots v_1$
5) if ($op \in \{<, >, \leq, \geq\}$) then
6) if ($v_1 < b_1 - 1$) then $B = b^1_{v_1}$
7) for $i = 2$ to $n$
8) if ($v_i \neq b_i - 1$) then $B = B \cdot b^i_{v_i}$
9) if ($v_i \neq 0$) then $B = B + b^i_{v_i-1}$
10) else
11) for $i$ = 1 to $n$
12) if ($v_i = 0$) then $B = B \cdot b^i_0$
13) else if ($v_i = b_i - 1$) then $B = B \cdot b^i_{v_i-1}$
14) else $B = B \cdot (b^i_{v_i} \oplus b^i_{v_i-1})$
15) if ($op \in \{>, \geq, \neq\}$) then return $B \cdot B_n$
16) else return $B \cdot B_n$
17) End

C.2 More about Significance Threshold

In section 4.3.2 we have defined the range of the significance threshold, $\theta$, as $0 < \theta < 0.5$. Since $0 < s < t$, we have $0 < \frac{\theta}{t} < 1$. Therefore, theoretically the range of $\theta$ is $0 < \theta < 1$. The reason why we define the upper bound of $\theta$ to be 0.5 instead of 1 is as follows.

If $0 < \theta < 1$, then the scenario in Figure 4.17(c) is divided into two sub-cases, instead of three. That is, $0 < \frac{\theta}{t} < \theta$ and $\theta \leq \frac{\theta}{t} < 1$. We argue that this kind of partitioning results in rough and
poor decision. Let us look at the two cases shown in Figure C.1.

Consider the case in Figure C.1(a). Since $t \theta$, i.e., the performance degradation due to long processing time of alternative 1 is significant with respect to $\theta$, it is preferable to choose alternative 2. However, if $\theta$ is set to a too high value, the decision based only on $\theta$ will be a poor one. As Figure C.1(a) shows, the response time of alternative 1 is much shorter than that of alternative 2. However, due to improper setting of $\theta$ (setting to a too high value in this case), the significance of the difference between processing time of both alternatives is overestimated. As a result, regardless the small $X$ of alternative 1, alternative 2 is chosen.

Consider the case in Figure C.1(b). Since $s \theta$, it means that the difference between the processing time of both alternatives is insignificant with respect to $\theta$, i.e., $s \approx t$, then alternative 1 is preferable. However, if $\theta$ is set to a too low value, the decision without considering relationship between $r$ and $s$ is also a poor one. As the case in Figure C.1(b) shows, according to the setting of $\theta$, $s \approx t$. However, in reality $s << t$ and $s \approx r$. Therefore, alternative 2 is in fact a better choice than alternative 1 in this case.

As a result, no matter $\theta$ is set to a high value or to a low value, the bipolar partitioning results in poor decision making in either case. Therefore, we propose to divide the scenarios into three partitions, i.e., $0 < t \theta$, $\theta \leq t \theta \leq 1 - t \theta$ and $1 - t \theta < t \theta < 1$. Since $\theta < 1 - \theta$, we have

$$\Rightarrow 0 < 2 \cdot \theta < 1 \Rightarrow 0 < \theta < 0.5$$
Appendix D

Relational Algebra Extended by Bitmaps

Query optimization consists of two sub-steps, logical optimization and physical optimization. Logical optimization transforms users’ queries expressed in high-level languages into the ones in the relational algebra. Physical optimization maps the optimal logical query plan to physical query operations. Physical query plans reveal the detailed execution plans, including the sequences of different operations and access paths being used.

Although logical and physical optimization are performed in two steps, they are not independent from each other. For example, the choices of access paths may affect the transformations at the step of logical optimization. To be able to express logical query plans with bitmap operations, we introduce bitmap operators to the relational algebra and the transformation rules for them. Bitmap operations can appear in the relational expression where conventionally conditions appear. That is, we will introduce bitmap operations to the selection, the join and the aggregation operators.

Let $A$ and $B$ denote two attributes, and $R$ and $S$ denote two tables. Suppose that bitmap indexes are defined on $A$ and $B$, denoted by $B^A$ and $B^B$. For each value, say $x$, of the attribute domain of $A$, the retrieval function, expressed by $f_x$, denotes the bitmap for $x$ where the 1 bits indicate those tuples with $A = x$. Now, we list the pairs of equivalent expressions in Table D.1. The column in the middle are the conventional expressions; while on the third column, the conditions are expressed by bitmap operations. A selection using bitmaps means that the bit positions of those 1 bits are the offsets of the tuples of concern in their logical table space.

<table>
<thead>
<tr>
<th>operator</th>
<th>conventional expression</th>
<th>bitmap expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma$</td>
<td>$\sigma_{A=x} R$</td>
<td>$\sigma_{f_x} R$</td>
</tr>
<tr>
<td></td>
<td>$\sigma_{A\neq x} R$</td>
<td>$\sigma_{T_x} R$</td>
</tr>
<tr>
<td></td>
<td>$\sigma_{A=x \text{ AND } B=a} R$</td>
<td>$\sigma_{f_x f_a} R$</td>
</tr>
<tr>
<td></td>
<td>$\sigma_{A \text{ IN } \langle a, y \rangle R}$</td>
<td>$\sigma_{f_x + f_y} R$</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>$R \bowtie_{A=B} S$</td>
<td>$\bigcup_{v \in A \cap B} (\sigma_{f_v} R \times \sigma_{f_v} S)$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma_{A, \text{sum}(B)} R$</td>
<td>$\bigcup_{v \in A} \pi_{A, \text{sum}(B)} (\sigma_{f_v} R)$</td>
</tr>
</tbody>
</table>

Table D.1: Bitmap operations

The following transformation rules are defined for bitmap operations.
<table>
<thead>
<tr>
<th>Equation</th>
<th>Mathematical Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_{f_a} \cdot f_b R = \sigma_{f_b} \cdot f_a R )</td>
<td>Commutative Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a + f_b} R = \sigma_{f_b + f_a} R )</td>
<td>Commutative Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a + 1} R = R )</td>
<td>Dominance Law</td>
</tr>
<tr>
<td>( \sigma_{f_a} \cdot 0 R = 0 )</td>
<td>Dominance Law</td>
</tr>
<tr>
<td>( \sigma_{f_a + f_a} R = \sigma_{f_a} \cdot f_a R = \sigma_{f_a} R )</td>
<td>Idempotent Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a \cdot 1} R = \sigma_{f_a} R )</td>
<td>Identity Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a + 0} R = \sigma_{f_a} R )</td>
<td>Identity Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a + (f_b + f_c)} R = \sigma_{f_a + f_b} + f_c R )</td>
<td>Associative Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a \cdot (f_b \cdot f_c)} R = \sigma_{f_a \cdot f_b} f_c R )</td>
<td>Associative Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a} (f_a + f_b) R = \sigma_{f_a} (f_a + f_b) R = \sigma_{f_a} R )</td>
<td>Absorption Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a} (f_a \cdot f_b) R = \sigma_{f_a \cdot f_b} R )</td>
<td>Distributive Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a} (f_b + f_c) )</td>
<td></td>
</tr>
<tr>
<td>( \sigma_{f_a \cdot (f_b + f_c)} R = \sigma_{f_a \cdot f_b} + f_c R )</td>
<td>Distribution Laws</td>
</tr>
<tr>
<td>( \sigma_{f_a + (f_b \cdot f_c)} R = \sigma_{f_a + f_b} + f_c R )</td>
<td>Distribution Laws</td>
</tr>
<tr>
<td>( \sigma_{(f_b \cdot f_c)} R = \sigma_{f_b \cdot f_c} R )</td>
<td>de Morgan’s Laws</td>
</tr>
<tr>
<td>( \sigma_{(f_a \cdot f_b)} R = \sigma_{f_a \cdot f_b} R )</td>
<td>de Morgan’s Laws</td>
</tr>
</tbody>
</table>

The above transformation rules are also applicable to Joins (\( \bowtie \)) and aggregations (\( \Sigma \)).
Appendix E

Simulation and Implementation Details

E.1 The Poisson Counting Process

A stochastic process is said to be a counting process if it represents the total number of “events” that have occurred up to a specific timepoint $t$, denoted by $N(t)$. The Poisson process with rate $\lambda (\lambda > 0)$ is a counting process, if

1. $N(0) = 0$,

2. the number of events which occur in disjoint time intervals are independent, and

3. the number of events in any time interval of length $t$ is Poisson distributed with mean $\lambda t$, and for all $s, t \geq 0$

$$P\{N(t+s) - N(s) = n\} = e^{-\lambda t} \frac{(\lambda t)^n}{n!}, \quad n = 0, 1, \ldots,$$

where $P\{N(t) = n\}$ denotes the probability of $n$ events up to timepoint $t$.

The Poisson process used in Section 4.3.5 is described as follows. Given is an attribute with cardinality equal to $m$, denoted by $v_1, \ldots, v_m$, and the probabilities of each value to be occurred are $p_1, \ldots, p_m$, respectively. Let $X_i (1 \leq i \leq m)$ be the number of occurrences of $v_i$ in $n$ trials. $X_i$ are independent Poisson random variables with respective means $\lambda_i (1 \leq i \leq m)$, where $\lambda_i = np_i$.

For more detailed properties of Poisson process, please confer [72].

E.2 Queries

In this section, we list the set of test queries which are run on the testing DBMSs. In Table E.1, we list the selected tables and Group By attributes of all queries.
<table>
<thead>
<tr>
<th>Query</th>
<th>parts</th>
<th>suppliers</th>
<th>lineitems</th>
<th>orders</th>
<th>customers</th>
<th>nations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table E.1: Selected tables (✓) and Group By (∨) attributes of each query

**Query 1**

```
SELECT c.mktsegment, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  o.year IN ('1995', '1996') AND
  c.nation_id IN ('01', '02')
GROUP BY c.mktsegment
ORDER BY c.mktsegment
```

**Query 2**

```
SELECT c.nation_id, SUM(l.qty)
FROM lineitems l, orders o, customers c
WHERE l.order_id = o.order_id AND
  o.cust_id = c.cust_id AND
  c.nation_id IN ('01','02','03','04','05') AND
  o.year IN ('1993', '1994', '1995')
GROUP BY c.nation_id
ORDER BY c.nation_id
```

**Query 3**

```
SELECT o.order_id, SUM(l.qty)
FROM lineitems l, orders o
WHERE l.order_id = o.order_id AND
  o.orderstatus = 'P' AND
  o.year IN ('1994', '1996') AND
  o.month IN ('01','02','03','10','11','12') AND
  o.shippriority IN ('1','W','g')
GROUP BY o.order_id
ORDER BY o.order_id
```
Query 4
```
SELECT p.brand, SUM(l.discount)
FROM parts p, lineitems l
WHERE p.part_id = l.part_id AND
  l.shipmode IN ('AIR', 'FOB') AND
  p.type IN ('PROMO', 'LARGE')
GROUP BY p.brand
ORDER BY p.brand
```

Query 5
```
SELECT p.brand, SUM(l.price)
FROM lineitems l, parts p
WHERE l.part_id = p.part_id AND
  p.brand IN ('Brand42', 'Brand34', 'Brand11', 'Brand44',
              'Brand25', 'Brand33', 'Brand23', 'Brand12') AND
  l.tax > 5
GROUP BY p.brand
ORDER BY p.brand
```

Query 6
```
SELECT c.lastname, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  c.acctbal > 9000 AND
  l.qty > 20 AND
  o.year IN ('1994', '1995', '1996') AND
  c.mktsegment IN ('BUILDING', 'AUTOMOBILE', 'MACHINERY', 'HOUSEHOLD')
GROUP BY c.lastname
ORDER BY c.lastname
```

Query 7
```
SELECT s.supp_id, s.name, s.phone, SUM(l.qty)
FROM suppliers s, parts p, lineitems l
WHERE s.supp_id = l.supp_id AND
  p.part_id = l.part_id AND
  p.brand IN ('Brand43', 'Brand54', 'Brand55', 'Brand15')
GROUP BY s.supp_id, s.name, s.phone
ORDER BY s.supp_id, s.name, s.phone
```

Query 8
```
SELECT o.year, o.month, c.nation_id, SUM(l.qty)
FROM lineitems l, orders o, customers c
WHERE l.order_id = o.order_id AND
  o.cust_id = c.cust_id AND
  c.nation_id IN ('11', '12') AND
  o.year IN ('1995', '1996') AND
  o.month IN ('01', '02') AND
  c.mktsegment IN ('BUILDING', 'AUTOMOBILE')
GROUP BY o.year, o.month, c.nation_id
ORDER BY o.year, o.month, c.nation_id
```

Query 9
```
SELECT o.year, o.month, SUM(l.qty)
FROM lineitems l, orders o
WHERE l.order_id = o.order_id AND
  o.year IN ('1995', '1996') AND
  o.month IN ('01', '02')
GROUP BY o.year, o.month
ORDER BY o.year, o.month
```
Query 10
SELECT s.nation_id, c.nation_id, o.year, SUM(l.qty)
FROM suppliers s, lineitems l, orders o, customers c
WHERE s.supp_id = l.supp_id AND
  l.order_id = o.order_id AND
  o.cust_id = c.cust_id AND
  s.nation_id IN ('02', '06') AND
  c.nation_id IN ('02', '06') AND
  o.year IN ('1995')
GROUP BY s.nation_id, c.nation_id, o.year
ORDER BY s.nation_id, c.nation_id, o.year

Query 11
SELECT c.cust_id, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  c.mktsegment IN ('BUILDING', 'AUTOMOBILE', 'MACHINERY', 'HOUSEHOLD') AND
  o.priority IN ('1-URGENT', '2-HIGH') AND
  c.nation_id IN ('01', '02')
GROUP BY c.cust_id
ORDER BY c.cust_id

Query 12
SELECT o.priority, SUM(l.qty)
FROM orders o, lineitems l
WHERE o.order_id = l.order_id AND
  o.year = '1995' AND
  o.month IN ('06', '07', '08') AND
  l.commityear = '1995' AND
  l.commitmonth IN ('06', '07', '08', '09', '10', '11', '12')
GROUP BY o.priority
ORDER BY o.priority

Query 13
SELECT s.nation_id, SUM(l.price)
FROM suppliers s, parts p, lineitems l
WHERE s.supp_id = l.supp_id AND
  p.part_id = l.part_id AND
  p.brand IN ('Brand42', 'Brand34', 'Brand11', 'Brand44',
  'Brand25', 'Brand33', 'Brand23', 'Brand12')
GROUP BY s.nation_id
ORDER BY s.nation_id

Query 14
SELECT c.cust_id, o.clerk, SUM(l.tax)
FROM lineitems l, orders o, customers c
WHERE l.order_id = o.order_id AND
  c.cust_id = o.cust_id AND
  c.mktsegment = 'HOUSEHOLD' AND
  o.priority = '2-HIGH'
GROUP BY c.cust_id, o.clerk
ORDER BY c.cust_id, o.clerk

Query 15
SELECT o.order_id, SUM(l.tax)
FROM lineitems l, orders o
WHERE l.order_id = o.order_id AND
  o.order_id IN ('000001', '000002', '000003', '000004',
  '000005', '000006', '000007', '000032', '000033', '000034',
  '000035', '000036', '000037', '000038', '000039', '000064')
GROUP BY o.order_id
ORDER BY o.order_id
Query 16

```
SELECT s.supp_id, SUM(l.discount)
FROM suppliers s, lineitems l
WHERE s.supp_id = l.supp_id
GROUP BY s.supp_id
ORDER BY s.supp_id
```

Query 17

```
SELECT c.cust_id, c.nation_id, SUM(l.qty)
FROM lineitems l, orders o, customers c
WHERE l.order_id = o.order_id AND
  o.cust_id = c.cust_id AND
  c.nation_id IN ('11', '12') AND
  o.year IN ('1995', '1996') AND
  o.month IN ('01', '02') AND
  c.mktsegment IN ('BUILDING', 'AUTOMOBILE')
GROUP BY c.cust_id, c.nation_id
ORDER BY c.cust_id, c.nation_id
```

Query 18

```
SELECT c.cust_id, s.supp_id, c.nation_id, s.nation_id, SUM(l.qty)
FROM suppliers s, lineitems l, orders o, customers c
WHERE s.supp_id = l.supp_id AND
  l.order_id = o.order_id AND
  o.cust_id = c.cust_id AND
  s.nation_id IN ('02', '06') AND
  c.nation_id IN ('02', '06') AND
  o.year IN ('1995')
GROUP BY c.cust_id, s.supp_id, c.nation_id, s.nation_id
ORDER BY c.cust_id, s.supp_id, c.nation_id, s.nation_id
```

Query 19

```
SELECT c.cust_id, o.clerk, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  c.nation_id IN ('10', '13', '19', '22') AND
  o.price > 300000
GROUP BY c.cust_id, o.clerk
ORDER BY c.cust_id, o.clerk
```

Query 20

```
SELECT c.cust_id, SUM(l.qty)
FROM orders o, lineitems l, customers c
WHERE o.order_id = l.order_id AND
  o.cust_id = c.cust_id AND
  o.year = '1995' AND
  o.month IN ('06', '07', '08') AND
  l.commityear = '1995' AND
  l.commitmonth IN ('06', '07', '08', '09', '10', '11', '12') AND
  o.priority IN ('1-URGENT', '2-HIGH')
GROUP BY c.cust_id
ORDER BY c.cust_id
```
Query 21
SELECT c.mktsegment, SUM(l.qty)
FROM orders o, lineitems l, customers c
WHERE o.order_id = l.order_id AND
  o.cust_id = c.cust_id AND
  o.year = '1995' AND
  o.month IN ('06', '07', '08') AND
  l.commityear = '1995' AND
  l.commitmonth IN ('06', '07', '08', '09', '10', '11', '12') AND
  o.priority IN ('1-URGENT', '2-HIGH')
GROUP BY c.mktsegment
ORDER BY c.mktsegment

Query 22
SELECT s.nation_id, SUM(l.tax)
FROM lineitems l, suppliers s
WHERE l.supp_id = s.supp_id
GROUP BY s.nation_id
ORDER BY s.nation_id

Query 23
SELECT p.brand, SUM(l.price)
FROM lineitems l, parts p
WHERE l.part_id = p.part_id AND
  l.tax > 5
GROUP BY p.brand
ORDER BY p.brand

Query 24
SELECT c.mktsegment, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  c.mktsegment IN ('BUILDING', 'AUTOMOBILE') AND
  c.acctbal > 5000 AND
  o.priority IN ('1-URGENT', '2-HIGH')
GROUP BY c.mktsegment
ORDER BY c.mktsegment

Query 25
SELECT c.cust_id, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  c.mktsegment IN ('BUILDING', 'AUTOMOBILE') AND
  c.acctbal > 5000 AND
  o.priority IN ('4-NOT SP', '5-LOW')
GROUP BY c.cust_id
ORDER BY c.cust_id

Query 26
SELECT c.cust_id, o.clerk, SUM(l.qty)
FROM customers c, orders o, lineitems l
WHERE c.cust_id = o.cust_id AND
  o.order_id = l.order_id AND
  o.price > 250000 AND
  o.year IN ('1993', '1994', '1995') AND
  c.nation_id IN ('02', '03', '18')
GROUP BY c.cust_id, o.clerk
ORDER BY c.cust_id, o.clerk
Query 27

```
SELECT s.nation_id, c.nation_id, o.year, SUM(l qty)
FROM suppliers s, lineitems l, orders o, customers c
WHERE s.supplier_id = l.supplier_id AND
  l.order_id = o.order_id AND
  o.cust_id = c.cust_id AND
  s.nation_id IN ('02', '06') AND
  c.nation_id IN ('02', '06') AND
  o.year IN ('1995', '1996')
GROUP BY s.nation_id, c.nation_id, o.year
ORDER BY s.nation_id, c.nation_id, o.year
```

E.3 The Syntax of Accepted SQL Statements in bitSQL

```
CREATE <FACT | DIM> TABLE <name> ( <attribute> <type> [, <attribute> <type>] );
DROP TABLE <name>;
INSERT INTO <name> ( attr_list ) VALUES ( value_list );
DELETE FROM <name> WHERE cond_lists;
LOAD FROM <name> INSERT INTO <table name>;
CREATE ENCODED BINDEX ON <name>.<attr> USING <name>.<attr>;
CREATE SIMPLE BINDEX ON <name>.<attr> USING <name>.<attr>;
CREATE GROUP BINDEX ON <name>.<attr> USING <name>.<attr>;
DROP BINDEX <name>;
CREATE FOREIGNKEY <alias.name> TO <alias.name>;
SELECT <alias.name[, alias.name]> FROM <table alias [,table alias]>;
SELECT <alias.name[, alias.name]> FROM <table alias [,table alias]> WHERE <cond>;
SELECT <alias.name[, alias.name]> FROM <table alias [,table alias]> WHERE <cond>
  GROUP BY <attr_list> ORDER BY <attr_list>;
<attr_list>:= <attribute> [, <attribute>]
<value_list>:= <value> [, <value>]
<cond>:= [alias.]name OP [alias.]name [AND <cond>]
  | [alias.]name OP values [AND <cond>]
  | [alias.]name IN { in_lists } [AND <cond>]
```
Appendix F

Symbols, Notations Used In This Work

F.1 The Relational Algebra

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ</td>
<td>selection</td>
<td>( \sigma_{A=v} R: ) select tuples from ( R ) if ( A = v )</td>
</tr>
<tr>
<td>( \pi )</td>
<td>projection</td>
<td>( \pi_A R: ) project ( R ) on attribute(s) ( A )</td>
</tr>
<tr>
<td>( \bowtie )</td>
<td>join</td>
<td>( R \bowtie_{A=B} S: ) join ( R ) and ( S ) if ( A = B )</td>
</tr>
<tr>
<td>( \cup )</td>
<td>union</td>
<td></td>
</tr>
<tr>
<td>( \cap )</td>
<td>intersection</td>
<td></td>
</tr>
<tr>
<td>( \Sigma )</td>
<td>grouping and aggregation</td>
<td>( \Sigma_{A,\sum(Q)} R: ) sum up the values of ( Q ) from table ( R ) for each distinct value of ( A )</td>
</tr>
</tbody>
</table>

Table F.1: Relational operators

Note that the operator, \( \Sigma \), is not the standard relational operator. We introduce it into this work, for the sake of clarity.

F.2 Operators and Symbols

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(- (\sim))</td>
<td>bitwise negation (used in algorithms)</td>
</tr>
<tr>
<td>(\cdot (&amp;))</td>
<td>bitwise ( AND ) (used in algorithms)</td>
</tr>
<tr>
<td>(+ (</td>
<td>))</td>
</tr>
<tr>
<td>(\mathbb{Z})</td>
<td>the set of integers</td>
</tr>
<tr>
<td>(\mathbb{Z}^+)</td>
<td>the set of positive integers (including 0)</td>
</tr>
<tr>
<td>(\mathbb{N})</td>
<td>the set of natural numbers</td>
</tr>
<tr>
<td>(\mathcal{U})</td>
<td>the universal set</td>
</tr>
<tr>
<td>(b)</td>
<td>a bitmap</td>
</tr>
<tr>
<td>(B^A)</td>
<td>a bitmap index on attribute ( A )</td>
</tr>
<tr>
<td>(A)</td>
<td>an attribute</td>
</tr>
<tr>
<td>(</td>
<td>A</td>
</tr>
</tbody>
</table>

Table F.2: Operators and general symbols
Bibliography


Index

Symbols

\( \lambda() \) .................................. 55, see binary distance
\( \sum m() \) .................................. 55, see sum of min-terms, 56, 60, 73
gt-pages ..................................... 109
bitSQL ........................................ 16, 147
architecture .................................. 148–151
compiler directives
BIT_HASHING .................................. 170
LINKED_LIST .................................. 153
ONLY_FIRST_GROUP .......................... 151
PAGEMODE_FOR_BITREAD .................. 152
REUSE_BITMAP_FOR_SELECTION ............ 156
SORTED_ARRAY ................................ 153
USING_FIBONACCI ............................ 152
restrictions .................................. 156
file descriptors .............................. 156
system internals ......................... 151–157
bitSQL\(^{\Delta} \) ................................ 170
bitSQL\(^{+} \) .................................... 153
bitSQL\(^{++} \) .................................... 162

A .............................................
absorption law ................................ 204
aggregation
on dimension table ......................... 116
using bitmaps ................................ 103
alphanumeric data ... see discrete data type
associative law ................................ 204

B ............................................
\( B^{++} \)-trees ............................... 28, see B-trees, 64
\( B^{+} \)-trees ............................... 27, see B-trees
redistributing .............................. 27
split factor .................................. 27, 64–66, 183
\( B^{+} \)-trees ............................... 27, see B-trees, 64
B-trees .................................... 15–17, 27, 27, 64
cooperativity .............................. 28, 30, 45
data structures ............................ 27
fill factor .................................. 168
height ..................................... 64
isomorphism .............................. 28, 29, 31
leaf node ................................... 27
non-leaf node .............................. 27
polymorphism .............................. 28, 30
restrictions .................................. 28–31
storage utilization ....................... 64, 183
batch-mode data loading ............. 123, 124, 157
binary coded decimals ................ 114
binary distance ............................ 54
binary distance ... see well-defined encoding
bit slices ... see bit-sliced index
bit-sliced index ......................... 17, 38, 73, 103
space-optimum ............................ 74
time-optimum .............................. 74
well-defined bases ...................... 74
bit-transposed file ......................... 82
bitmap compression ..................... 81
bitmap consumer ......................... 176
bitmap indexing ......................... 17, 32
bitmapped join-indexes ............. 44
cooperativity ......................... 34, 38, 43
deleted/non-existent tuples ........ 33, 51, 52
design space ............................ 40
bit-encoding ............................ 40
domain-encoding ....................... 41
value decomposition .................. 40
encoded bitmap index ... see encoded
bitmap index
group-set bitmap indexes ........... 42
NULL values ............................ 33
simple bitmap index ... see simple bitmap index
bitmap provider ....................... 176
bitmap-enabled query processor ........ see
bitSQL
bitmapGrouping ........................ 109, 110
bitmapOrderedGrouping ............. 112
blocking factor ..................... 18, 24, 98, 99, 122
Boolean algebra ...................... 52
bucket tuning.......................... 125, see overflow avoidance

C .............................................. 145
  cache management...................... 64, 101
  chain .................................... 54, 184–186
  chain.......................... see well-defined encoding
  clustering factor ...................... 122, 124, 125
  coloring problem ...................... 18, 100
  combination with repetitions ......... 193
  commutative law ....................... 204
  compressed bitmaps........... see bitmap compression
  continuous data type .............. 39, 75, 177
  continuous range query ......... 76, see continuous range search
  continuous range search .... 68, 74, 75, 76, 79
  continuous range selection .... see continuous range search
  cooperativity ...................... see McDips
  counting process .................... 125, 205
  CPU-bound ............................ 136, 152
  cross-relation partitions .......... 121

D .............................................
  data fetching phase ............... 96, 98, 102
  Data Warehouse ....................... 14
    definition ............................. 14
    logical architecture .............. 14
    data store component .......... 14
    OLAP engine ......................... 14
  DataIndex ................................ 82
  de Morgan’s law ...................... 204
  decision support systems ......... see DSS
  dimension table 17, 31, 36, 42, 44, 58, 90, 91, 117, 126, see Star Schema
  discrete data type ............... 39, 60, 73, 75, 177
  discrete range query ....... 76, see discrete range search
  discrete range search ............ 68, 77, 79
  discrete range selection .... see discrete range search
  disjointness constraint ........ 91, 105
  disjunctive normal form .......... 37
  distributive law .................... 204
  dominance law ..................... 96, 204
  drill-down ............................ 58
  drill-up .................................. 58
  DSS ........................................ 13
  duality .................................. see McDips
  dynamic bitmap ....................... 81
  dynamic destaging .................. 125, see overflow avoidance
  dynamic optimization ............. 95
  dynamic range-based bitmap ...... 81, see range-based encoded bitmap

E .............................................
  EBI ...................................... see encoded bitmap index
  effective cardinality of range selections 64, see cardinality of range selection
  effective inclusion .................. 102
  EIS ...................................... 13
  encoded bitmap index ......... 17, 35–38, 47–58
    definition ............................ 47
    encoding ............................ 53
    hierarchy ............................ 58
    isomorphism ....................... 48, 61
    optimal ............................. 54
    range-based ......................... 61
    total-order preserving .......... 60
    well-defined ...................... see well-defined encoding
    examples ............................. 58
    maintaining ....................... 48–52
    domain expansion ............... 49
    performance ........................ 62
      compare with B-trees ............ 64
      compare with simple bitmaps ... 70
    processing costs .................. 63
    space requirements .............. 62
    entropy ............................ 180, see Huffman coding
    equality-encoding ............... 73
  evalRestAgg ......................... 116
  executive information systems .... see EIS
  experimental setups ............... 157
    commercial DBMSs .............. 159
    data population ................. 157
    power-test ........................ 157
    schema .............................. 157

F .............................................
  fact table 17, 31, 36, 42, 44, 58, 90, 91, 94, 117, 121, 126, see Star Schema
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false drops</td>
<td>43, 121</td>
</tr>
<tr>
<td>Fibonacci numbers</td>
<td>152</td>
</tr>
<tr>
<td>fill factor</td>
<td>43, 91, 105, 149, 170, 180, 189</td>
</tr>
<tr>
<td>fundamental conjunction</td>
<td>37, 47, 52, 54–56</td>
</tr>
<tr>
<td>generalized hash teams</td>
<td>43, 121</td>
</tr>
<tr>
<td>global time optimum</td>
<td>79</td>
</tr>
<tr>
<td>space constraint</td>
<td>80</td>
</tr>
<tr>
<td>Group By evaluation</td>
<td>104</td>
</tr>
<tr>
<td>using bitmaps</td>
<td>103</td>
</tr>
<tr>
<td>group-set bitmap index</td>
<td>42, 43, 90, 92, 94, 106, 107, 116, 129, 162, 178</td>
</tr>
<tr>
<td>hierarchy encoding</td>
<td>58, 59, 175</td>
</tr>
<tr>
<td>Huffman coding</td>
<td>180</td>
</tr>
<tr>
<td>Huffman coding tree</td>
<td>44, 123, 125</td>
</tr>
<tr>
<td>hybrid hashing</td>
<td>44, 123, 125</td>
</tr>
<tr>
<td>I/O-bound</td>
<td>136</td>
</tr>
<tr>
<td>idempotent law</td>
<td>204</td>
</tr>
<tr>
<td>identity law</td>
<td>54, 56, 96, 204</td>
</tr>
<tr>
<td>IN-lists</td>
<td>96, 97, 99, 101, 102</td>
</tr>
<tr>
<td>index scanning phase</td>
<td>96, 97, 99, 101, 102</td>
</tr>
<tr>
<td>index tuning</td>
<td>170</td>
</tr>
<tr>
<td>merging of trailing groups</td>
<td>174</td>
</tr>
<tr>
<td>indexes</td>
<td>17–45, 47, 54–56</td>
</tr>
<tr>
<td>B-trees, bitmap indexing, encoded</td>
<td>174</td>
</tr>
<tr>
<td>bitmap index, bit-sliced index, simple</td>
<td>174</td>
</tr>
<tr>
<td>bitmap index</td>
<td>174</td>
</tr>
<tr>
<td>isomorphism</td>
<td>25</td>
</tr>
<tr>
<td>Join DataIndex</td>
<td>82</td>
</tr>
<tr>
<td>Karnaugh map</td>
<td>55, 179</td>
</tr>
<tr>
<td>circuit</td>
<td>55</td>
</tr>
<tr>
<td>path</td>
<td>55</td>
</tr>
<tr>
<td>least significant bit</td>
<td>47</td>
</tr>
<tr>
<td>loading factor</td>
<td>65</td>
</tr>
<tr>
<td>logical optimization</td>
<td>88</td>
</tr>
<tr>
<td>LSB</td>
<td>see least significant bit</td>
</tr>
<tr>
<td>McDips</td>
<td>25</td>
</tr>
<tr>
<td>min-term</td>
<td>47, 55, 56, 60, 102</td>
</tr>
<tr>
<td>minimal loading factor</td>
<td>47</td>
</tr>
<tr>
<td>most significant bit</td>
<td>47</td>
</tr>
<tr>
<td>MSB</td>
<td>see most significant bit</td>
</tr>
<tr>
<td>multi-value Group By</td>
<td>91, 92, 120, 122</td>
</tr>
<tr>
<td>multidimensional indexes</td>
<td>see UB-tree</td>
</tr>
<tr>
<td>multidimensionality</td>
<td>see McDips</td>
</tr>
<tr>
<td>multidimensionality</td>
<td>137, 145</td>
</tr>
<tr>
<td>nested query</td>
<td>180</td>
</tr>
<tr>
<td>nested-loop join</td>
<td>29</td>
</tr>
<tr>
<td>non-binary encoded bitmap index</td>
<td>199</td>
</tr>
<tr>
<td>numeric data</td>
<td>see continuous data type</td>
</tr>
<tr>
<td>OLAP</td>
<td>13, 25</td>
</tr>
<tr>
<td>multidimensional OLAP</td>
<td>14</td>
</tr>
<tr>
<td>OLAP queries</td>
<td>15, 31, 90, 94, 158</td>
</tr>
<tr>
<td>relational OLAP</td>
<td>14</td>
</tr>
<tr>
<td>OLTP</td>
<td>13</td>
</tr>
<tr>
<td>on-line analytical processing</td>
<td>see OLAP</td>
</tr>
<tr>
<td>on-line transaction processing</td>
<td>see OLTP</td>
</tr>
<tr>
<td>online aggregation</td>
<td>121</td>
</tr>
<tr>
<td>Order By</td>
<td>111</td>
</tr>
<tr>
<td>optimization</td>
<td>111</td>
</tr>
<tr>
<td>overflow avoidance</td>
<td>125</td>
</tr>
<tr>
<td>page hits</td>
<td>see coloring problem, 100, 122, 124, 125</td>
</tr>
<tr>
<td>page-level bitmap</td>
<td>19, 25, 94, 98</td>
</tr>
<tr>
<td>paradox of index- and table-scans</td>
<td>18</td>
</tr>
<tr>
<td>partial aggregates</td>
<td>113</td>
</tr>
<tr>
<td>physical optimization</td>
<td>88</td>
</tr>
<tr>
<td>pipelining</td>
<td>94</td>
</tr>
<tr>
<td>Poisson counting process</td>
<td>94</td>
</tr>
<tr>
<td>process</td>
<td></td>
</tr>
</tbody>
</table>
Poisson process .................................. 125, 205
polymorphism ...................................... see McDips
prefix B-trees ..................................... 25
prime chain ........................................ see well-defined encoding, 54, 186
principle of inclusion and exclusion .......... 96
cost-efficiency .................................... 99
projection expansion .......................... 151
projection index ................................. 38, 82
query execution .................................. 86
   compilation ................................. 86
   execution .................................... 86
   optimization ................................. 86
   parsing ....................................... 86
query execution flows .......................... 87
query processing ............................... 85
data fetch ................................. see data fetching phase
   index scans .......................... see index scanning phase
overview ....................................... 86
query rewrite ................................. see query rewrite
   selection evaluation ...................... 95
dynamic optimization ........................ 96
   static optimization ....................... 95
tree-reduction .................................. 96
query processing time ........................ 85
query response time .......................... 85
query rewrite 88, 91, 132–134, 149–151, 159,
   162, 164, 170, 178, 189
   proof ......................................... 189
range-based encoded bitmap ................ 61, see encoded bitmap index
range-encoding .................................. 73
reducible Boolean function .................. 53, 55, 57
Reductio ad Absurdum ......................... 54, 56
referential integrity ........................... 91, 134, 180, 189
relational algebra ............................. 16, 29, 104, 203
   extension .................................. 90
   transformation rules ....................... 88
report generator ............................. 13
retrieval Boolean function 36, 47, 49, 50, 52,
   57, 60, 61, 63, 75, 97, 100, 102, 106
   retrieval function .......................... see retrieval Boolean function
rotation latency .................................. 68
seek time ........................................ 68
selectivity ...................................... 18, 21, 48, 68, 100, 103, 122, 128, 132, 156, 168
significance threshold ....................... 115, 116, 201
simple bitmap index .......................... 32
   example ....................................... 32
Snow-flake Schema 36, see Star Schema, 43,
   91
snow-flaked dimension 150, see Snow-flake
   Schema, 157, 159, 162, 164, 168, 175, 178
space efficiency ............................... see McDips
   Star Schema .................................. 58
Star-Join ......................................... 31, 44, 88, 90, 116, 117, 126, 129, 132–134, 149, 150,
   157, 162, 168, 178, 189
static optimization .......................... 95
stochastic process ............................ 125, 205
sum of min-terms ............................... 55, 60, 73
tautology ........................................ 52, 54, 56, 63, 186
test queries ..................................... 205
total specification ............................. 91
TPC-D Benchmark .............................. 140
TPC-H Benchmark 31, 35, 42, 127, 140, 157,
   158
transfer time ................................... 68
transposed file .................................. 82
tree-reduction ................................. 96
tuple-level bitmap 19, see page-level bitmap,
   25, 94, 98
Type I query see continuous range search
Type II query see discrete range search
well-defined encoding ........................ 56, 184
working set ..................................... 164
zig-zag join ..................................... 29