Regis University

ePublications at Regis University

Regis University Student Publications (comprehensive collection)

Regis University Student Publications

Spring 2009

A Strategy for Reducing I/O and Improving Query Processing Time in an Oracle Data Warehouse Environment

Chris Titus
Regis University

Follow this and additional works at: https://epublications.regis.edu/theses

 Part of the Computer Sciences Commons

Recommended Citation

Titus, Chris, "A Strategy for Reducing I/O and Improving Query Processing Time in an Oracle Data Warehouse Environment" (2009). *Regis University Student Publications (comprehensive collection)*. 113.
https://epublications.regis.edu/theses/113

This Thesis - Open Access is brought to you for free and open access by the Regis University Student Publications at ePublications at Regis University. It has been accepted for inclusion in Regis University Student Publications (comprehensive collection) by an authorized administrator of ePublications at Regis University. For more information, please contact epublications@regis.edu.

Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.

A Strategy for Reducing I/O and Improving Query Processing Time In an Oracle Data Warehouse Environment

By

Chris Titus
titus830@regis.edu

A Thesis/Practicum Report submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Information Technology

School of Computer and Information Sciences
Regis University
Denver, Colorado
December 12, 2008

Abstract

In the current information age as the saying goes, time is money. For the modern information worker, decisions must often be made quickly. Every extra minute spent waiting for critical data could mean the difference between financial gain and financial ruin. Despite the importance of timely data retrieval, many organizations lack even a basic strategy for improving the performance of their data warehouse based reporting systems.

This project explores the idea that a strategy making use of three database performance improvement techniques can reduce I/O (input/output operations) and improve query processing time in an information system designed for reporting. To demonstrate that these performance improvement goals can be achieved, queries were run on ordinary tables and then on tables utilizing the performance improvement techniques. The I/O statistics and processing times for the queries were compared to measure the amount of performance improvement. The measurements were also used to explain how these techniques may be more or less effective under certain circumstances, such as when a particular type of query is run.

The collected I/O and time based measurements showed a varying degree of improvement for each technique based on the query used. A need to match the types of queries commonly run on the system to the performance improvement technique being implemented was found to be an important consideration. The results indicated that in a reporting environment these performance improvement techniques have the potential to reduce I/O and improve query performance.

Table of Contents

Certification of Authorship of Thesis/Practicum Work ii
Authorization to Publish Student Work iii
Releasor Authorization to Publish Student Work on WWW iv
Regis University Faculty Approval Form vi
Abstract vii
Table of Contents viii
List of Figures ix
List of Tables xi
Executive Summary 1
Chapter 1 – Introduction 3
Chapter 2 – Review of Literature and Research 5
2.1 Databases in a Data Warehousing Environment 5
2.2 Performance Improvement and Tuning Strategies 7
2.3 Bitmap indexes 8
2.4 Table Partitioning 10
2.5 Denormalization 12
Chapter 3 – Methodology 16
3.1 Hardware and Software Testing Environment 16
3.2 Table Partitioning Test Method 16
3.3 Bitmap Index Test Method 17
3.4 Denormalization Test Method 19
3.5 Improvement Calculation 20
Chapter 4 – Partitioning 22
4.1 Partitioning Results and Analysis 22
4.2 Partitioning Summary 28
Chapter 5 – Bitmap Indexing 32
5.1 Bitmap Indexing Results and Analysis 32
5.2 Bitmap Indexing Summary 39
Chapter 6 – Denormalization 43
6.1 Denormalization Results and Analysis 43
6.2 Denormalization Summary 50
Chapter 7 – Conclusions 52
7.1 Integration 52
Chapter 8 – Lessons Learned 54
8.1 Challenges 54
8.2 Limitations 55
References 56
Annotated Bibliography 61

List of Figures

Figure 3-1: Schema for Partition testing 17
Figure 3-2: Schema for Bitmap Index testing 18
Figure 3-3: Schema for Denormalization Testing 20
Figure 4-1: Query 1u 22
Figure 4-2: Query 1p 22
Figure 4-3: Query 2u 23
Figure 4-4: Query 2p 23
Figure 4-5: Query 3u 23
Figure 4-6: Query 3p 24
Figure 4-7: Query 4u 24
Figure 4-8: Query 4p 24
Figure 4-9: Query 5u 25
Figure 4-10: Query 5p 25
Figure 4-11: Query 6u 25
Figure 4-12: Query 6p 26
Figure 4-13: Query 7u 26
Figure 4-14: Query 7p 27
Figure 4-15: Query 8u 27
Figure 4-16: Query 8p 27
Figure 4-17: Postal Code Hash Partitioning 29
Figure 4-18: Partitioning Performance 30
Figure 4-19: Partitioning Physical Reads 31
Figure 5-1: Query 1a 32
Figure 5-2: Query 1b 32
Figure 5-3: Query 2a 33
Figure 5-4: Query 2b 33
Figure 5-5: Query 3a 33
Figure 5-6: Query 3b 33
Figure 5-7: Query 4a 34
Figure 5-8: Query 4b 34
Figure 5-9: Query 5a 35
Figure 5-10: Query 5b 35
Figure 5-11: Bitmap Performance by Query Set 36
Figure 5-12: Query 6a 36
Figure 5-13: Query 6b 37
Figure 5-14: Query 7a 37
Figure 5-15: Query 7b 37
Figure 5-16: Bitmap Optimizer Cost 39
Figure 5-17: Bitmap Physical Reads 40
Figure 5-18: Execution Plan for a Count Query with No Joins 41
Figure 5-19: Execution Plan for a Count Query with Joins 41
Figure 5-20: Execution Plan for a Query Retrieving Rows 42

Figure 6-1: Query 1n 44
Figure 6-2: Query 1d 44
Figure 6-3: Query 2n 45
Figure 6-4: Query 2d 45
Figure 6-5: Query 3n 46
Figure 6-6: Query 3d 46
Figure 6-7: Query 4n 47
Figure 6-8: Query 4d 48
Figure 6-9: Denormalization Performance 48
Figure 6-10: Denormalization I/O Performance 50

List of Tables

Table 4-1: Partitioning Time Measurements 28
Table 4-2: Partitioning I/O Measurements 30
Table 5-1: Bitmap Time Measurements 38
Table 5-2: Bitmap I/O Measurements 39
Table 6-1: Denormalization Time Measurements 43
Table 6-2: Denormalization I/O Measurements 49

Executive Summary

Disk and memory access are the main obstacles on the path to reducing query processing time. Data warehouses systems, known for their immense size and long running queries, can be severely affected by these impediments. As result, a performance improvement strategy is needed which reduces the amount of costly disk and memory access, also known as I/O.

Bitmap indexing, partitioning, and denormalization are three performance improvement techniques that are available in many of the major enterprise database systems in use today. These techniques work on the premise of reducing I/O, with lower I/O leading to faster query processing times. Faster query times can mean the difference between having key data at hand when it is needed for decision making and getting it several hours after the decisions were made.

To determine the extent of practical performance these techniques could potentially provide and how they work with different types of queries, a series of queries were run in a test environment. Measurements and statistics resulting from the testing were then collected and analyzed. It was found that certain types of queries performed better when used with a particular technique and that a considerable gain in performance could be achieved when the type of query and technique were properly matched up.

Ultimately, the testing and analysis of the three performance improvement techniques indicated they are capable of considerably improving performance when implemented under the proper conditions. The techniques can co-exist in the same

system, can potentially complement one another, and are recommended for use as part of a comprehensive data warehouse performance improvement strategy.

Chapter 1 – Introduction

The reduction of query processing time in a database system is a common but sometimes elusive goal. Reducing processing time for queries in a data warehouse environment, which typically houses a large amount of data, can prove to be particularly challenging. These challenges can be overcome through the use of a three pronged approach to performance improvement. Implementing a performance improvement strategy that includes table partitioning, bitmap indexing, and de-normalization can reduce I/O in a data warehouse system, potentially leading to shorter query processing times.

Data warehouse systems allow a company to pool their data into a central repository for reporting. Reports created using this data can help provide managers with the information they need to make important business decisions. As businesses begin to realize the value of such a capability, these systems have become more popular. As Goeke and Faley (2007) state, "It is no surprise that with its potential to generate extraordinary gains in productivity and sales, the market for data warehousing software and hardware was nearly $200 billion by 2004" (p. 107).

Gray and Watson (1998) point out that, " […] a data warehouse is physically separate from operational systems; and data warehouses hold both aggregated data and transaction (atomic) data, which are separate from the databases used for On-Line Transaction Processing (OLTP)" (p. 84). Since a data warehouse system serves a different purpose than a database designed for processing transactions and therefore its

performance is affected in different ways. Improving the performance of a database configured as a data warehouse system requires a different strategy than that used for a transaction processing system.

Instead of recording financial transactions or sales orders from a web site, the data warehouse acts as a central repository for historical data. Because the data warehouse system is not constantly updated like a transaction processing system, exists primarily to respond to queries, and houses tables that often contain a large number of rows, it is an excellent candidate for de-normalized tables, indexes, and partitioned tables.

Chapter 2 – Review of Literature and Research

2.1 Databases in a Data Warehousing Environment

A database lies at the heart of every data warehouse system. According to Mallach (2000), the majority of work done on a data warehouse project is involved with the database itself. In fact, "Experts regularly mention figures of up to 80 percent" (Mallach, 2000, p. 492). The level of performance provided by this database can mean the difference between waiting for hours to generate a report on key business indicators, and having the information in the hands of decision makers within a matter of minutes. As Shasha (1996) points out, "In fields ranging from arbitrage to tactical missile defense, speed of access to data can determine success or failure" (p. 113).

Morzy and Wremble declare in their 2004 paper on multiversion data warehousing that, "A data warehouse (DW) integrates autonomous and heterogeneous external data sources (EDSs) and makes the integrated information available for analytical processing, decision making, and data mining applications" (p. 92).The data warehouse database exists separately from the production databases (Chaudhuri & Dyal, 1997), which are responsible for processing the daily transactions of the business and […] "reflect the current state of the organization" […] (Saharia & Babad, 1997, p. 43). The data warehouse database captures this data, storing it for historical reference and analysis. As Palpanas (2000) writes in his paper on data warehouse knowledge discovery, "Data warehouses tend to be fairly big in size, usually orders of magnitude larger than operational Databases" (p. 1).

Data from the production databases is moved into the warehouse through an Extract, Transform, and Load (ETL) process (March & Hevner, 2007). This can sometimes lead to problems in data warehouse implementation. In their 2007 paper on decision support systems, March and Hevner report that "Often operational systems are not designed to be integrated and data extracts must be performed manually or on a schedule determined by the operational systems" (p. 1037).

Before dedicated data warehouse systems came into being, analytic queries were run on production database systems, often adversely affecting their performance (Gray & Watson, 1998). Technical staff using these systems soon realized that a totally separate database would be required to house the historical data. Since this database would be separate from the production system, it could be designed differently and optimized for reporting.

Saharia and Babad (2000), describe a multi-level system architecture with operational production databases distinctly separated from the historical data repository. Modern data warehouses take advantage of a similar architecture with OLTP databases feeding data into a separate data warehouse database.

In their paper on data mart design, Bonifati, Cattaneo, Ceri, Fuggetta, & Paraboschi (2001) point out that many data warehouse systems are comprised of smaller known as data marts. These sub components of the data warehouse are "[…] dedicated to the study of a specific problem" (Bonifati, et al., 2001, p. 453).

Queries processed on data warehouse systems often retrieve aggregate (broad) data as well as more specific (deep) data, such as that dealing with particular products or customers (Shasha & Bonnet, 2003).

2.2 Performance Improvement and Tuning Strategies

Database performance improvement involves making changes to a database system that can potentially improve its performance. Many methods for improving performance have been suggested throughout the evolution of the relational database.

In an early paper, Shasha (1996) explores a performance improvement strategy that focuses on tuning transactions, indexes, and table structures. He suggests the use of vertical partitioning to physically segregate data by column and processing transactions so that record locking is minimized.

Agrawal, Chu, & Narrasaya (2006) declare in their paper on automated database tuning, "[…] a strategy that tunes physical design only for queries and ignores updates while tuning (thus there is no transition cost since the physical design does not need to change) can also be sub-optimal since the cost of updating physical design structures can be substantial (p. 684). The authors believe that tuning a database for one particular function may not always be practical because physical structures occasionally change. For example, indexes may be dropped before an update and re-created afterward.

Ongoing maintenance and tuning are required for any type of database system, but having a good design in the beginning can save a great deal of time. In their 2007 book on performance tuning for data warehouse systems, Stackowiak, Rayman, and

Greenwald suggest that, “Addressing performance and scalability during the development lifecycle has a significant positive impact on overall performance and reduces production performance complaints” (Production Phase section, para. 5).

2.3 Bitmap indexes

Bitmap indexes have been around in one form or another since the 1960’s (Wu, 1999) and made an appearance commercially in a system known as Model 204 (O’Neil & Quass, 1997). Bitmaps are now supported by several major database vendors including Oracle. In fact, the bitmap index has been a part of the Oracle RDBMS for several releases with Oracle users seeing the first incarnation of the bitmap index in version 7.3 (Kyte, 2005).

The structure of a bitmap index is composed of ones and zeros that identify whether a certain column value exists in a particular row. Bitmaps are suited for a read-only environment such as a data warehouse system, which is usually set up for querying historical data. As Stockinger, Wu, and Shoshani (2002) observe, “They are not optimized for typical transaction operations such as insert, delete or update” (p. 73).

Ingram (2002) notes that, “Bitmap indexes are most suitable for query-intensive applications, where queries use combinations of low cardinality columns in predicates containing equality, AND, OR, and NOT operations (Bitmap Indexes section, para. 6)” Low cardinality refers to the ratio between the number of unique rows and the number of rows in a table (Lane, Schupmann, & Stuart, 2007). So a column that has 8 unique values

(such as available product colors, for example) in a table of 10,000 rows would be considered by many to have low cardinality.

Compression and encoding of bitmap indexes can increase their flexibility. Research conducted by Stockinger, Wu, and Shoshani (2002), suggests that compression can increase the flexibility of a bitmap index by making it"[…] useful for high cardinality attributes" (p. 73) as well as low cardinality attributes. Wrembel and Koncilia (2007), note that different types of encoding can make bitmaps more suitable for certain operations. For example, bitmaps can be range-encoded by storing multiple bits over a range of values, which can improve the performance of range queries (Wrembel and Koncilia, 2007).

One advantage of bitmap indexes observed by Chan and Ioannidis (1998) is their small size. Whereas a b-tree index stores rowids, the Oracle bitmap index instead stores compressed bitmaps that point to rowids (Lane et al., 2007). This results in a smaller index size and therefore a shorter index scan time.

As Wu (1999) reasons in his research on the structure of bitmap indexes, "Even for low selectivities, if the time of index processing is high, the total time spent on index processing and data retrieval may be longer than that of a table scan" (p. 227). As a result, bitmap indexes can potentially provide lower query processing times than a b-tree index when implemented on a column with low cardinality and using appropriate query operators (and, or, and not).

Bitmap indexes also provide other query processing related benefits. Hellerstein, J. & Stonebraker note that, "[…] bit map [sic] indexes can be intersected and unioned

very efficiently to deal with Boolean combinations of predicates. B-trees are much less efficient in this kind of processing" (Data Warehousing section, para. 14).

As with other indexes, bitmaps must be properly implemented and maintained in order to be effective. Powell (2004) points out that when used improperly, "[…] both bitmap and function-based indexes are largely ineffective and can be exceedingly detrimental to performance" (What and How to Index section, para. 5). Inmon, Rudin, Buss, and Sousa (1999), observe that as the cardinality of a column increases, the size of an associated bitmap index increases as well, since more bits are required to represent the value for each row.

A subtype of the bitmap index available in Oracle is known as the bitmap join index. The bitmap join index stores the rowids of two columns in different tables together as if those tables were joined (Niemiec, 2007). When a query is run in which these tables are joined on the indexed columns, the results of the join operation are already stored in the index, which can greatly reduce query processing time.

2.4 Table Partitioning

An important goal of read-optimized databases is to minimize the number of bytes read from the disk when scanning a table (Harizopoulos, Liang, Abadi, & Madden, 2006). One way this can be accomplished is by dividing a single database structure such as a table into smaller sections known as partitions.

A table can be partitioned both vertically by columns and horizontally by rows. In their paper on integrating automated database design with partitioning, Agrawal,

Narrasaya, and Yang (2004) assert that "Like indexes and materialized views, both kinds of partitioning can significantly impact the performance of the *workload* i.e., queries and updates that execute against the database system, by reducing cost of accessing and processing data" (p. 359).

Vertical partitioning can be used to separate commonly queried columns. As Graefe, (1993) mentions in his paper on the physical design of databases, "Vertical partitioning of large records Into multiple files, each with some of the attributes, allows faster retrieval of the most commonly used attributes […]" (p. 78). For example, if the values of three columns are commonly requested in queries, those three columns could be stored in a separate partition. When a query is processed requiring data from only those columns, only that partition must be accessed.

The Oracle RDBMS first supported partitioning in version 8 (Kyte, 2005). The types of partitioning supported by Oracle 10g include range partitioning, list partitioning, hash partitioning, and two hybrid partitioning methods (Powell, 2005).

Range partitioning separates data based on an array of values. For example, ten years of a company's historical sales data could be stored in a large table. In this case, the table could be range partitioned by year. If a query was run to find data from the year 2005, only the partition containing the 2005 data would have to be accessed, instead of the table containing ten years worth of data. Along this same line, Hobbs, Hillson, and Lawande (2003) note that, "If a table is partitioned, the query optimizer can determine if a certain query can be answered by reading only specific partitions. Thus, an expensive table scan can be avoided" (Benefits of Partitioning section, para. 4).

List partitioning is similar to range partitioning only it separates data based on specific values. For example, an organization's sales data could be partitioned based on geographic location, such as state, country, or region.

Hash Partitioning is a little different from list and range partitioning in that it doesn't separate data based on a value or range of values. Hash partitioning instead uses an algorithm to equally divide data into a number of partitions.

When designing a database system for a data warehouse in which very large tables will exist, partitioning can help the designer work around certain design limitations. In their text on physical database design, Lightstone, Teorey, and Nadeau (2007) note that some database systems have limits on table size that can be overcome through the use of horizontal table partitioning.

2.5 Denormalization

Denormalization involves reversing the process known as normalization in relational database systems. In their paper on denormalization guidelines, Bock and Schrage (2002) declare that a normalized table, "[…] minimizes redundant data storage and supports data manipulation language processing for row insertions, deletions, and updates without introducing errors or data inconsistencies termed anomalies"(p. 129). As a result of the importance of normalization, it has become a part of the database design process when developing transaction processing systems.

Data warehouse systems do not process the same number of update, insert, and delete operations that a transactional system does, since its primary function is usually to hold data for reporting. Because of this behavior, tables in a data warehouse system do not benefit from normalization as a transaction based system would.

Although normalization does have benefits in a transaction processing system it also suffers from a few disadvantages. Bock and Schrage (2002) believed that, "These include inadequate system response time for data retrieval and storage, and referential integrity problems due to the decomposition of natural data objects into relational tables." (p. 129).

Normalization generally results in larger tables being broken up into smaller tables through a process called decomposition (Artz, 1997). This requires more tables to be joined in complex queries, which can lead to greater query processing times (Bock and Schrage, 2002).

In his paper on data warehouse semantics and normalization, Artz (1997) argues that semantic disintegrity can occur when normalization is not performed correctly. According to Artz (1997), "Semantic disintegrity occurs when a user submits a query against a database and receives an answer, but the answer is not the answer to the question they believe that they asked" (p. 22). This can have disastrous consequences for a new data warehouse implementation.

Gorla's (2003) paper on data warehouse features offers suggestions for making data warehouse acceptance among users more likely and details his attempts to measure a system's perceived ease of use. To underline this point, Goeke and Faley (2007) find that

users who see a data warehouse system as difficult to use are not likely to adopt it, which could potentially lead to the failure of the project.

The denormalization process mandates the restructuring of database tables by reducing the number of joins required for queries. During this process, data held within several smaller tables may be combined into a single larger table.

The use of denormalization techniques to improve performance plays a big part in Ralph Kimball's dimensional modeling strategy. Kimball (2002) suggests that data should be divided into fact and dimension tables, with the fact tables holding assessed data (such as sales numbers) and dimension tables holding attributes used for filtering data in queries (such as sales region). In their 2005 paper on varying data warehousing methodologies, Sen and Sinha describe a fact table as, "[…] a specialized relation with a multi-attribute key and contains attributes whose values are generally numeric and additive" (p. 80).

Both types of tables are denormalized and many queries can be run by joining a fact table with one dimension table. This model containing a fact table connected to multiple dimension tables is known as a star schema (Kimball, 2002). A star schema can be further transformed into a snowflake schema by breaking down the dimensions into smaller tables based on certain columns (Sen & Sinha, 2005). This makes the snowflake schema a star schema/3rd normal form hybrid (Martyn, 2004).

Although many authors such as Kimball (2002) promote the use of the denormalized star schema, others such as Martyn (2004) believe in using a traditional normalized schema with the use of denormalization only when necessary to improve

performance. In Martyn's 2004 paper on multidimensional schemas, he describes the advantages/disadvantages of both normalized and denormalized schemas. Martyn (2004) argues that, "Making special case design modifications to a 3NF schema is much less radical than adopting a specialized design methodology that specifically targets a MD schema" (p. 88). Although Martyn advises that denormalized tables only be used in certain situations, he agrees that a denormalized schema can potentially improve performance. This performance gain comes with associated costs, such as a loss of meaning when the schema is compared to entities in the real world (Martyn, 2004).

Chapter 3 – Methodology

3.1 Hardware and Software Testing Environment

The hardware for the testing environment consisted of a desktop computer with a single Pentium 4, 2.4 Gigahertz processor, and 1.5 GB of memory. A single hard drive supplied 160GB of storage for the RDBMS. The testing for this project was performed using Oracle 10g Enterprise Edition software, release 10.2.0.1.0. The Oracle data warehouse sales history sample schema included with the database software was installed and modified for use in the project. Oracle's SQL Developer version 1.5.1 was used to help create and prepare the test schema. SQL*plus was used to run the test queries with timing and autotrace enabled.

3.2 Table Partitioning Test Method

Two sets of tables containing address, customer, and sales data were used to test the effect of partitioning on I/O and query processing time (see Figure 3-1). Both sets of tables contained an identical number of rows and data. The partitioned fact table was range partitioned on the time_id column by quarter, half year, and year. The partitioned dimension tables DIM_ADDRESS_PART and DIM_CUSTOMER_PART were hash partitioned on the POSTAL_CODE and CUST_EMAIL columns respectively. All the tables contained around thirty-four million rows each, except the address tables which contained around ten million rows each. Each query was run ten times on the normalized and denormalized tables to calculate the average processing time for each query.

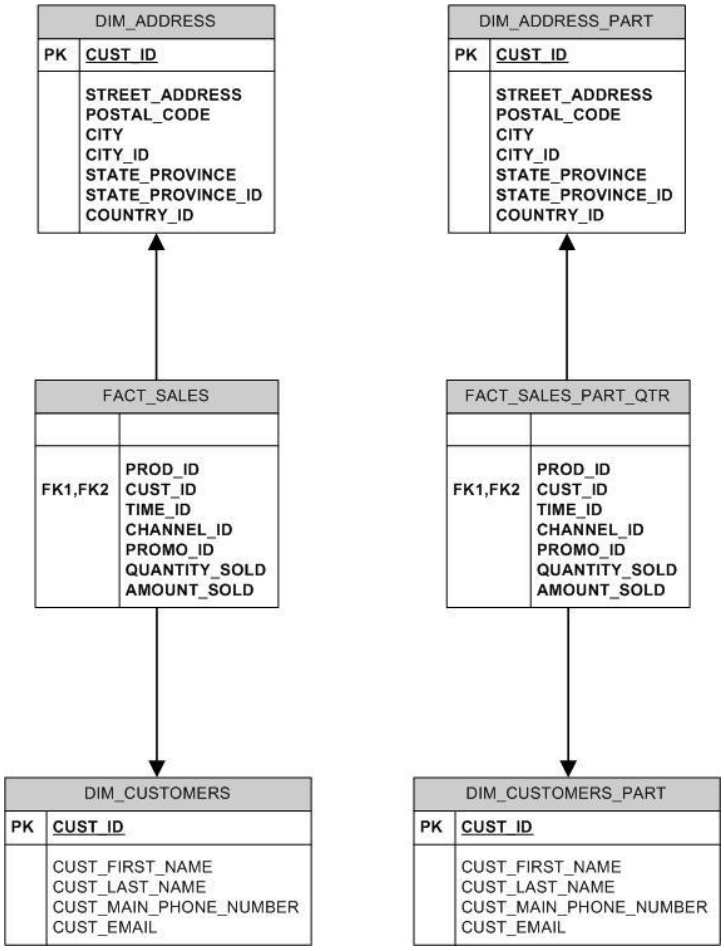


Figure 3-1: Schema for Partition testing

3.3 Bitmap Index Test Method

Tables containing address, customer, and sales data were used to test the effect of bitmap indexing on I/O and query processing time (see Figure 3-2). The dimension tables contained an identical number of rows and shared the same fact table which was not indexed. One set of tables did not have any indexed columns, while the other set of tables had bitmap indexes on the CUST_GENDER, CUST_INCOME_LEVEL, CUST_MARITAL_STATUS, and STATE_PROVINCE columns. The tables each held

around thirty-three million rows when testing was conducted, except for the fact table which held thirty four million rows. Each query was run ten times on the normalized and denormalized tables to calculate the average processing time for each query.

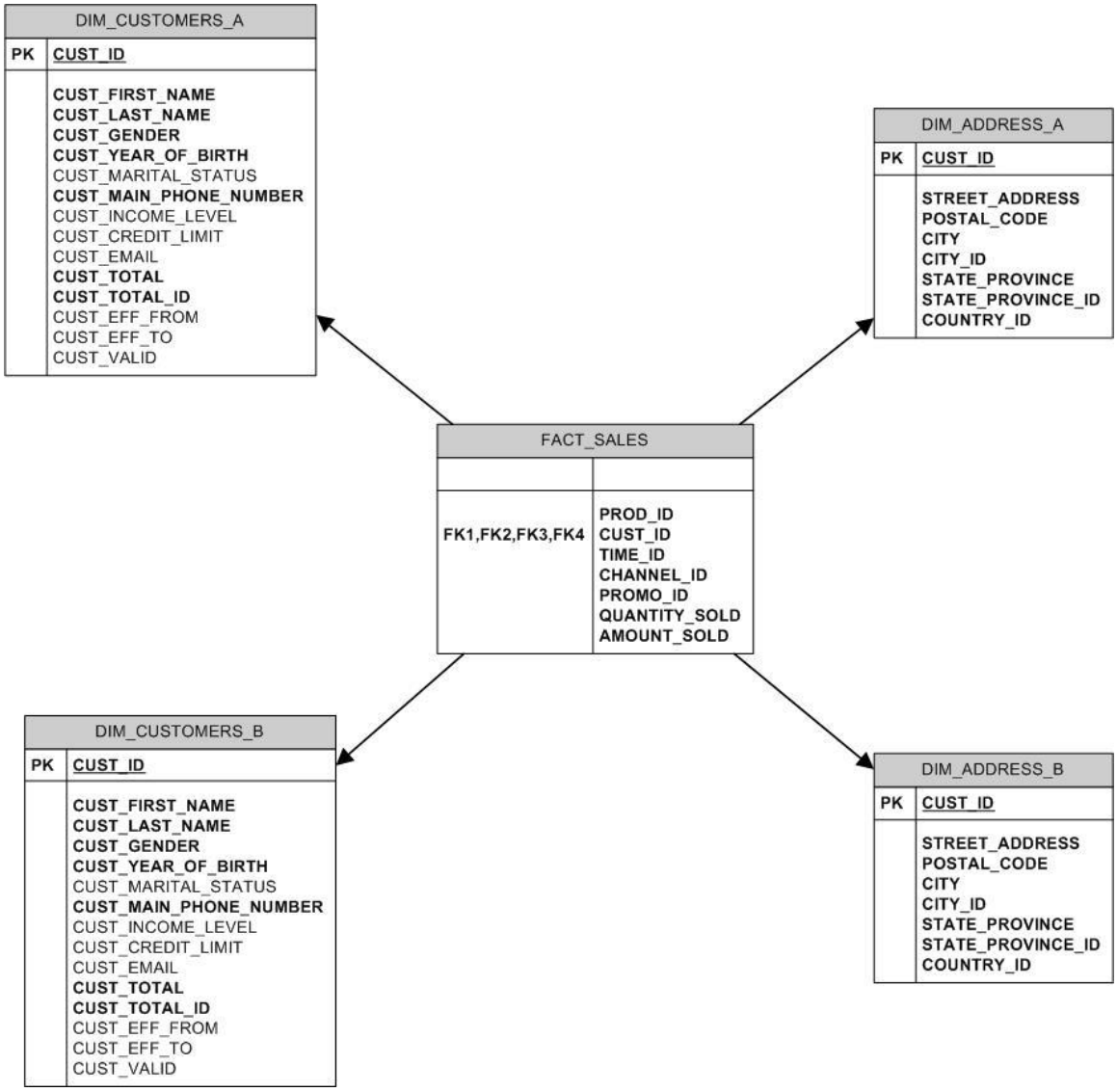


Figure 3-2: Schema for Bitmap Index testing

3.4 Denormalization Test Method

A set of normalized tables and a single denormalized table were used to test the effects of denormalization on query I/O and processing time. Both contained the same data and number of rows. Each of the tables held around eleven million rows.

It is important to note that using a single denormalized table to hold all the data in a schema is not something that is commonly done in the real world as far as I am aware. Using this single denormalized table was done to help demonstrate the performance benefits of denormalization in a data warehouse system. It is more common to see tables in a data warehouse system arranged in a star schema with joins being required between dimension and fact tables. In this arrangement users can take advantage of Oracle's star transformation feature to optimize queries, which is beyond the scope of this project.

Several queries were run on the normalized/denormalized tables and their I/O statistics collected. The normalized and denormalized tables both contained close to eleven million rows during testing. Each query was run ten times on the normalized and denormalized tables to calculate the average processing time for each query.

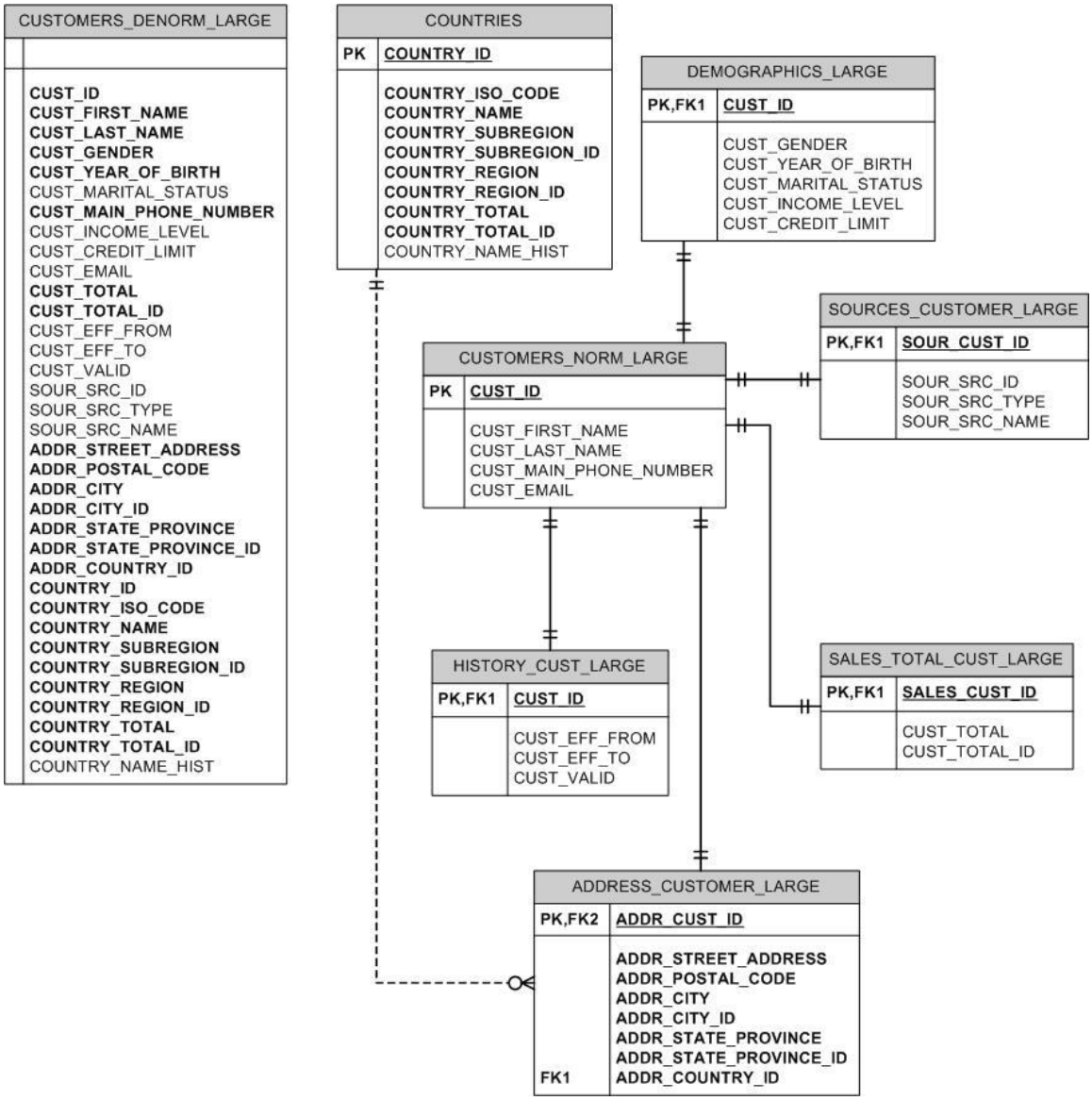


Figure 3-3: Schema for Denormalization Testing

.

3.5 Improvement Calculation

It is difficult to define substantial improvement when query processing times can vary by such a large degree and so many variables are involved. Practically speaking, an information worker would likely consider a fifty percent reduction in the processing time

of a long running query to be substantial. For example, if a query that normally takes four hours to complete finishes in two hours that would probably be considered to be a substantial improvement by many users. The improvement percentage calculation for query processing times is based on the following formula where t_1 = the first query completion time measured and t_2 = the second query completion time measured: $((t_1 - t_2) \div t_1) \times 100$

This method of calculating performance improvement as a percentage of the original query time was chosen because it provides a way to easily demonstrate the degree of improvement for queries regardless of whether the initial processing time was short or long.

Chapter 4 – Partitioning

4.1 Partitioning Results and Analysis

To test the effects of table partitioning a series of eight queries were run on partitioned (p) and unpartitioned tables (u). First, queries 1 u and 1p (see figures 4-1 and 4-2) were run.

```
select *
from fact_sales
where time_id = '03-JAN-99';
```

Figure 4-1: Query 1u

```
select *
from fact_sales_part_qtr
where time_id = '03-JAN-99';
```

Figure 4-2: Query 1p

Queries 1u and 1p (Figures 4-2 and 4-3) both accessed only one table and 1p accessed a single partition. Table 4-2 shows a big difference in the number of physical reads between these two queries. When the query was run on the unpartitioned table, the entire table of around thirty-three million rows had to be scanned. When it was run on the partitioned table, only a single partition containing a fraction of the total number of rows had to be scanned. Instead of scanning thirty-three million rows, only about two million were scanned, leading to a much lower optimizer cost for query 1p and a ninety percent improvement in performance over query 1u.

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
where time_id = '03-JAN-99'
and cust_email = 'hortense.wiley@company2.com';
```

Figure 4-3: Query 2u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
where time_id = '03-JAN-99'
and cust_email = 'hortense.wiley@company2.com';
```

Figure 4-4: Query 2p

The second set of queries (Figures 4-3 and 4-4) was similar to the first set of queries but accessed two tables. Query 2p was able to access two table partitions instead of performing a full table scan on the two tables and it outperformed query 2u by ninety-two percent. Again, Table 4-2 shows a large difference in the number of physical reads was observed as was a sizeable difference in optimizer cost.

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
        join dim_address
          using(cust_id)
where time_id = '03-JAN-99'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-5: Query 3u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
        join dim_address_part
          using(cust_id)
where time_id = '03-JAN-99'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-6: Query 3p

Queries 3u and 3p (Figures 4-6 and 4-7) continued with the format of the first and second sets of queries, but added a third table with query 3p accessing a total of three partitions. The third table added to the queries was smaller in size than the other tables and contained fewer partitions. This resulted in a very slight increase in performance that was close to that of the previous set of queries and measured less than one percent over query 2p.

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
        join dim_address
          using(cust_id)
where time_id between '01-JAN-99' and '01-JUL-99'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-7: Query 4u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
        join dim_address_part
          using(cust_id)
where time_id between '01-JAN-99' and '01-JUL-99'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-8: Query 4p

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
        join dim_address
          using(cust_id)
where time_id between '01-JAN-99' and '01-SEP-99'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-9: Query 5u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
        join dim_address_part
          using(cust_id)
where time_id between '01-JAN-99' and '01-SEP-99'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-10: Query 5p

The fourth and fifth sets of queries (Figures 4-7 through 4-10) vary from the previous queries in that they expand the range of data retrieved from the fact table. In queries 4p and 5p this results in the access of additional fact table partitions. The addition of one extra partition didn't affect performance much for query 5p with performance decreasing by less than one percent compared to query 4p. The optimizer cost and physical reads for queries 4p and 5p were also very close.

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
        join dim_address
          using(cust_id)
where time_id between '01-JAN-99' and '01-DEC-01'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-11: Query 6u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
        join dim_address_part
          using(cust_id)
where time_id between '01-JAN-99' and '01-DEC-01'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-12: Query 6p

Queries 6u and 6p (Figures 4-11 and 4-12) included an increase in the number of partitions accessed in the fact table over the previous queries. Fourteen partitions were accessed in these queries with the increase being dramatically reflected in the number of physical reads and the higher optimizer cost. Scanning nine additional fact table partitions more than tripled the number of physical reads for query 6p when compared to query 5p, and came close to doubling the optimizer cost. The increase in I/O caused by the additional partition access resulted in the percentage of improvement falling from eighty-seven percent in query 5p to sixty percent in query 6p.

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
        join dim_address
          using(cust_id)
where time_id between '01-JAN-99' and '01-DEC-02'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-13: Query 7u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
        join dim_address_part
          using(cust_id)
where time_id between '01-JAN-99' and '01-DEC-02'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-14: Query 7p

The seventh and eighth sets of queries (Figures 4-13 through 4-16) continued to increase the range of data in the fact table, for a total of eighteen in query 7p and twenty-two in query 8p. The trend of decreasing performance continued as well, with the percentage of improvement dropping to fifty-one percent for query 8p. Physical reads increased as did the optimizer cost, indicating a relationship between the number of partitions accessed and the amount of I/O generated by a query.

```
select *
from fact_sales
    join dim_customers
      using(cust_id)
        join dim_address
          using(cust_id)
where time_id between '01-JAN-98' and '01-DEC-03'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-15: Query 8u

```
select *
from fact_sales_part_qtr
    join dim_customers_part
      using(cust_id)
        join dim_address_part
          using(cust_id)
where time_id between '01-JAN-98' and '01-DEC-03'
and cust_email = 'hortense.wiley@company2.com'
and postal_code = '66798';
```

Figure 4-16: Query 8p

4.2 Partitioning Summary

By looking at the test results in Tables 4-1 and 4-2 it can be seen that table partitioning has the potential to reduce I/O and improve performance, with the amount of improvement determined by several factors. The level of performance provided by a table partitioning strategy will increase when queries access a small number of partitions in relation to the total number in the table. As the number of partitions accessed by a query increases, the level of performance will decrease.

Table 4-1: Partitioning Time Measurements

Queries	Number of Table Partitions Accessed	Avg. Query Time (seconds) Query u	Avg. Query Time (seconds) Query p	Avg. Time Difference (seconds)	Percentage of Improvement
1u, 1p	1	31	3	28	90%
2u, 2p	2	67	5	62	92%
3u, 3p	3	85	7	78	92%
4u, 4p	4	89	12	77	87%
5u, 5p	5	95	12	83	87%
6u, 6p	14	98	39	59	60%
7u, 7p	18	99	41	58	57%
8u, 8p	22	99	49	50	51%

Partitioning performance is also affected by the number of rows in a partition and whether the rows in a partitioned table are evenly distributed. Partitions that do not contain approximately the same number of rows will likely still show performance improvement, but may not provide predictable results. A greater number of partitions in

a table can potentially provide a greater amount of performance improvement, especially on large tables with many rows.

Figure 4-17: Postal Code Hash Partitioning

```
DW_USER@orcldw> select count(*) from dim_address_part partition(sys_p111) where postal_code = '66798';

  COUNT(*)
----------
         0

Elapsed: 00:00:02.74
DW_USER@orcldw> select count(*) from dim_address_part partition(sys_p112) where postal_code = '66798';

  COUNT(*)
----------
     17399

Elapsed: 00:00:02.15
DW_USER@orcldw> select count(*) from dim_address_part partition(sys_p113) where postal_code = '66798';

  COUNT(*)
----------
         0

Elapsed: 00:00:02.48
```

Figure 4-17 reveals that the 17399 rows in this table containing postal code 66798 reside in a single partition. If a query's where clause did not filter based on the postal code column, but instead filtered by state, the data requested could reside in more than one partition. That would require accessing multiple partitions and could possibly affect query performance. To achieve optimal performance with partitioning, it is best to match the where clause column of queries to the partition key column of a partitioned table.

Table 4-2: Partitioning I/O Measurements

Queries	Physical Reads Query u	Physical Reads Query p	Optimizer Cost Query u	Optimizer Cost Query p
1u, 1p	161362	10349	34737	1136
2u, 2p	419296	18849	86396	2845
3u, 3p	532214	25369	108000	4291
4u, 4p	532160	44891	117000	44243
5u, 5p	532175	44901	120000	47220
6u, 6p	531872	134781	159000	86287
7u, 7p	532082	135147	161000	87644
8u, 8p	531967	163605	178000	105000

The sudden drop in performance shown in Figure 4-18 can be attributed to the increase in I/O caused by going from five partitions scanned to fourteen partitions scanned. Table 4-2 shows that performance decreased from eighty-seven percent to sixty percent between queries six and seven as the additional partitions were accessed.

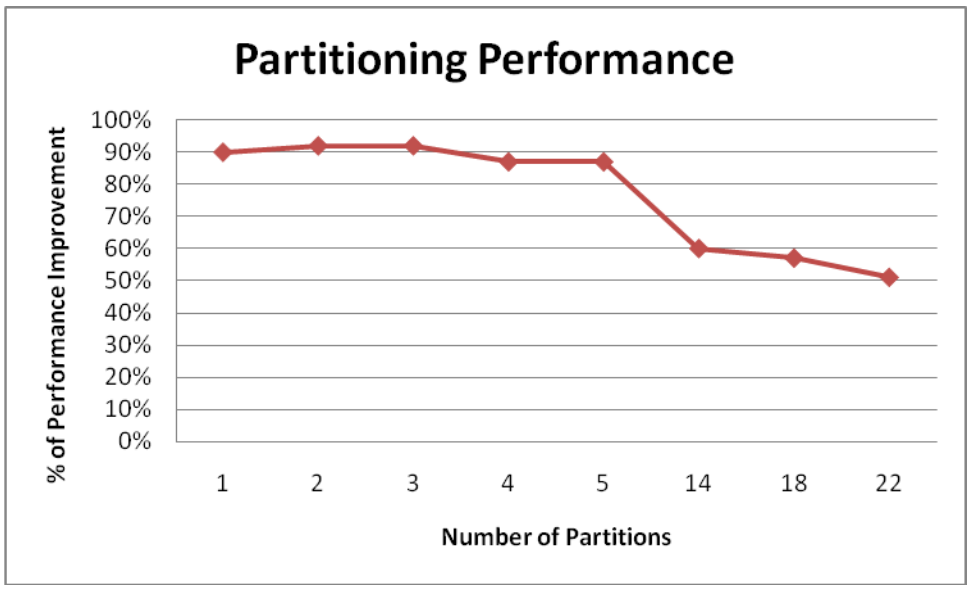


Figure 4-18: Partitioning Performance

Figure 4-19 graphically illustrates the rise in physical reads between queries 5p and 6p that corresponds with the drop in performance observed in Figure 4-18. While the I/O on the unpartitioned tables stayed the same after the third set of queries, the I/O on the partitioned tables slowly increased as additional partitions were accessed (see Figure 4-19). Since the amount of I/O increases with the number of partitions scanned, running queries that require access to a small number of partitions will provide the best performance.

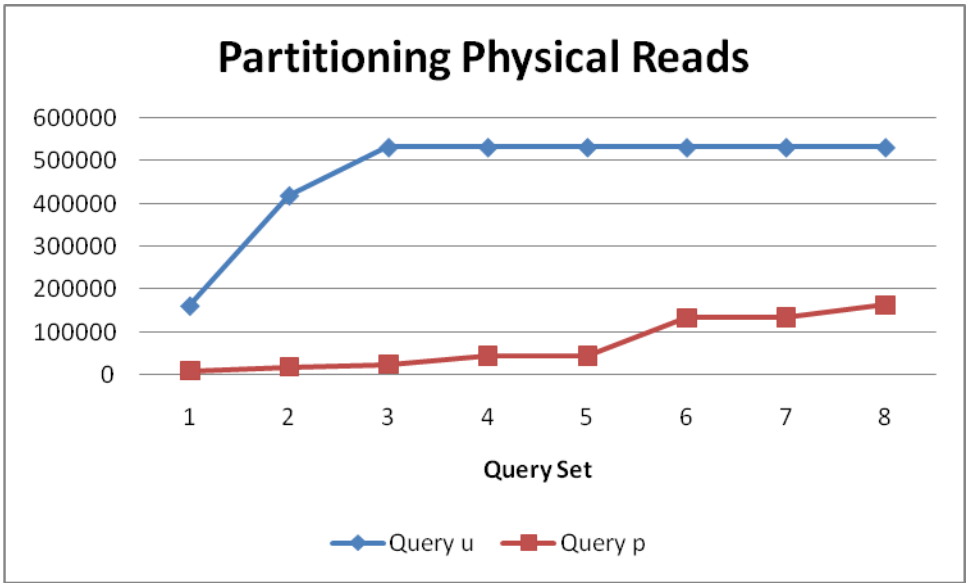


Figure 4-19: Partitioning Physical Reads

Chapter 5 – Bitmap Indexing

5.1 Bitmap Indexing Results and Analysis

The bitmap index testing consisted of running a set of seven queries on tables with (b) and without (a) bitmap indexes. Queries 1a through 7a (Figures 5-1 through 5-17) were run on tables without bitmap indexes, while queries 1b through 7b were run on tables with bitmap indexes. I/O statistics and time measurements were collected in an attempt to determine how query performance might be affected by the implementation of bitmap indexes.

```
select count(*)
from dim_customers_a
where cust_gender = 'F';
```

Figure 5-1: Query 1a

```
select count(*)
from dim_customers_b
where cust_gender = 'F';
```

Figure 5-2: Query 1b

The first set of queries that were run (see Figures 5-1 and 5-2) returned a count from two nearly identical indexed and non-indexed tables. Query 1a performed a full table scan while query 1b read the bitmap index. Since a count operation was performed, the bits in the index designating the female gender were counted without touching the table. Because only the index was accessed for query 1b, physical I/O is practically

nonexistent as shown in Table 5-2. As a result, a major performance gain was achieved by running a count query that referenced a bitmap indexed column.

```
select count(*)
from dim_customers_a
where cust_gender = 'F'
  and cust_marital_status = 'divorced';
```

Figure 5-3: Query 2a

```
select count(*)
from dim_customers_b
where cust_gender = 'F'
  and cust_marital_status = 'divorced';
```

Figure 5-4: Query 2b

The second and third sets of queries (Figures 5-3 through 5-6) added additional WHERE clauses to filter on marital status and income level. Again, the queries on the indexed table produced minimal I/O and consequently finished in a fraction of the time required by the queries on the non-indexed table.

```
select count(*)
from dim_customers_a
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above';
```

Figure 5-5: Query 3a

```
select count(*)
from dim_customers_b
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above';
```

Figure 5-6: Query 3b

Queries 4a and 4b (Figures 5-7 and 5-8) were run next, which returned rows based on criteria in the previous queries instead of counting them. When this query was run the bits in the bitmap indexes were counted as before, but this time the rows had to be returned instead of only being counted. Because the rows had to be returned, an additional bitmap conversion step was required in which the affected rowids were identified through the bitmap index. The table then had to be accessed to retrieve the rows.

```
select *
from dim_customers_a
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above';
```

Figure 5-7: Query 4a

```
select *
from dim_customers_b
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above';
```

Figure 5-8: Query 4b

When running the previous count queries only the index had to be accessed. The bitmap to rowid conversion operation required additional I/O and caused a small drop in performance. As shown in table 5-2, the amount of physical disk access for query 4b was zero since a very small number of rows were retrieved and already available in the cache.

```
select count(*)
from dim_customers_a
  join fact_sales
    using(cust_id)
      join dim_address_a
        using(cust_id)
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above';
```

Figure 5-9: Query 5a

```
select count(*)
from dim_customers_b
  join fact_sales
    using(cust_id)
      join dim_address_b
        using(cust_id)
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above';
```

Figure 5-10: Query 5b

The 5th set of queries (see Figures 5-9 and 5-10) brought about a steep drop in performance as physical and logical I/O increased dramatically. A graphical illustration of the performance decrease can be seen in Figure 5-11and matching I/O statistics in Table 5-2. This was caused by joining to a fact table and another dimension table. A full table scan was performed on the fact table while a fast full index scan was performed on the primary key index of other dimension table.

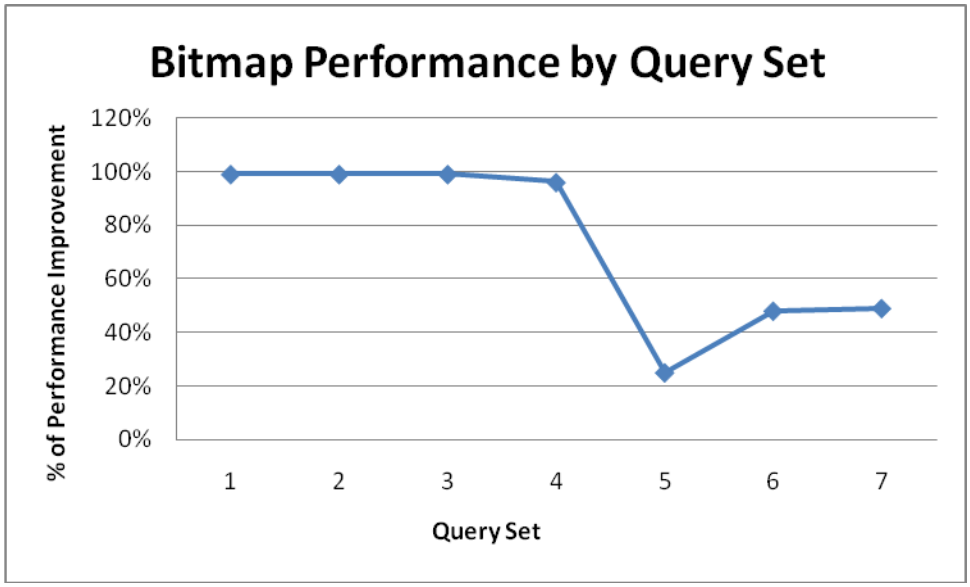


Figure 5-11: Bitmap Performance by Query Set

Since this was a count query only the primary key index needed to be accessed in order to determine the number of rows in the table and thus a full table scan on the other dimension table was avoided. The full table scan and hash joins operations caused by joining tables together were very costly in terms of I/O, bringing the performance improvement level down to twenty-five percent.

```
select count(*)
from dim_customers_a
  join fact_sales
    using(cust_id)
      join dim_address_a
        using(cust_id)
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above'
 and state_province = 'OH';
```

Figure 5-12: Query 6a

```
select count(*)
from dim_customers_b
  join fact_sales
    using(cust_id)
      join dim_address_b
        using(cust_id)
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above'
  and state_province = 'OH';
```

Figure 5-13: Query 6b

Figure 5-11 graphically illustrates the rapid decrease in performance that was measured when queries 5a and 5b were run. A quick look at Figure 5-16 and Figure 5- 17 shows that the cost of both queries increased substantially over their predecessors, while the number of physical reads for query 5a was over twice the number recorded for query 5b. Not having to perform full table scans on DIM_CUSTOMERS_B and DIM_ADDRESS_B considerably lowered the amount of physical disk access produced by query 5b.

```
select *
from dim_customers_a
  join fact_sales
    using(cust_id)
      join dim_address_a
        using(cust_id)
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above'
 and state_province = 'OH';
```

Figure 5-14: Query 7a

```
select *
from dim_customers_b
  join fact_sales
    using(cust_id)
      join dim_address_b
        using(cust_id)
where cust_gender = 'F'
  and cust_marital_status = 'divorced'
  and cust_income_level = 'L: 300,000 and above'
  and state_province = 'OH';
```

Figure 5-15: Query 7b

Queries 6a and 6b (see Figures 5-12 and 5-13) added an additional WHERE clause to the previous set of queries. Query 6b was able to make use of a bitmap index on the DIM_ADDRESS_B table to improve performance. While query 6a was performing full table scans on three tables, query 6b was only performing a full table scan on one table, leading to a forty-eight percent difference in performance.

Table 5-1: Bitmap Time Measurements

Queries	Bitmap Indexes Accessed	Number of Joins	Avg. Query Time (seconds) Query a	Avg. Query Time (seconds) Query b	Percentage of Improvement
1a, 1b	1	0	83	1	99%
2a, 2b	2	0	79	1	99%
3a, 3b	3	0	79	1	99%
4a, 4b	3	0	81	3	96%
5a, 5b	3	2	288	217	25%
6a, 6b	4	2	165	85	48%
7a, 7b	4	2	167	85	49%

The seventh and final set of queries (see Figures 5-14 and 5-15) changed the previous set of count queries so that rows are returned instead of counted. Had the query not already included join operations, this would have generated additional I/O for bitmap to rowid conversion. However, these bitmap conversions were already taking place because of the joins so query performance was not adversely affected. A look at Table 5-1 indicates that the average processing times for queries 6b and 7b were the same.

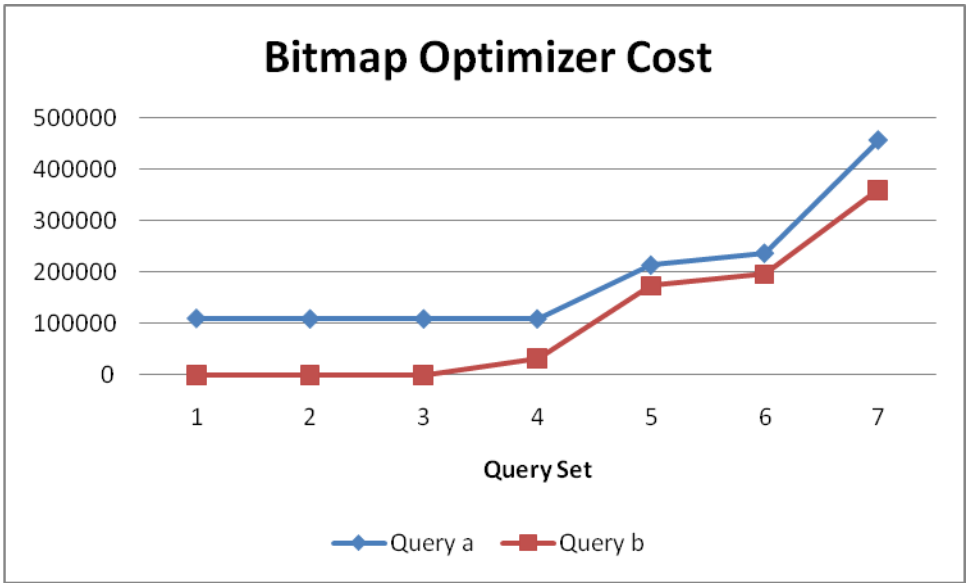


Figure 5-16: Bitmap Optimizer Cost

5.2 Bitmap Indexing Summary

In the previous tests bitmap indexes exhibited excellent performance improvement on count queries with no table joins. When a count query's WHERE clauses reference columns that have been indexed, many times only the indexes must be accessed. This greatly improves the performance of these types of queries when compared to that of tables not associated with bitmap indexes.

Table 5-2: Bitmap I/O Measurements

Queries	Physical Reads Query a	Physical Reads Query b	Optimizer Cost Query a	Optimizer Cost Query b
1a, 1b	558551	0	111000	584
2a, 2b	558396	0	110000	282
3a, 3b	558790	0	110000	424
4a, 4b	558787	0	110000	32763
5a, 5b	889232	331719	214000	174000
6a, 6b	1049830	274031	237000	197000
7a, 7b	1049847	273095	456000	361000

Bitmap indexes can significantly reduce the amount of physical I/O in queries, especially count queries. Figure 5-17 displays the progression of the physical reads for the seven test queries. The number of physical reads for queries 1b through 7b (on the indexed tables) is consistently far lower than that of queries 1a through 7a (non-indexed tables). Even the two join operations in query 5b didn't cause the same sharp increase in physical I/O that was observed in query 5a.

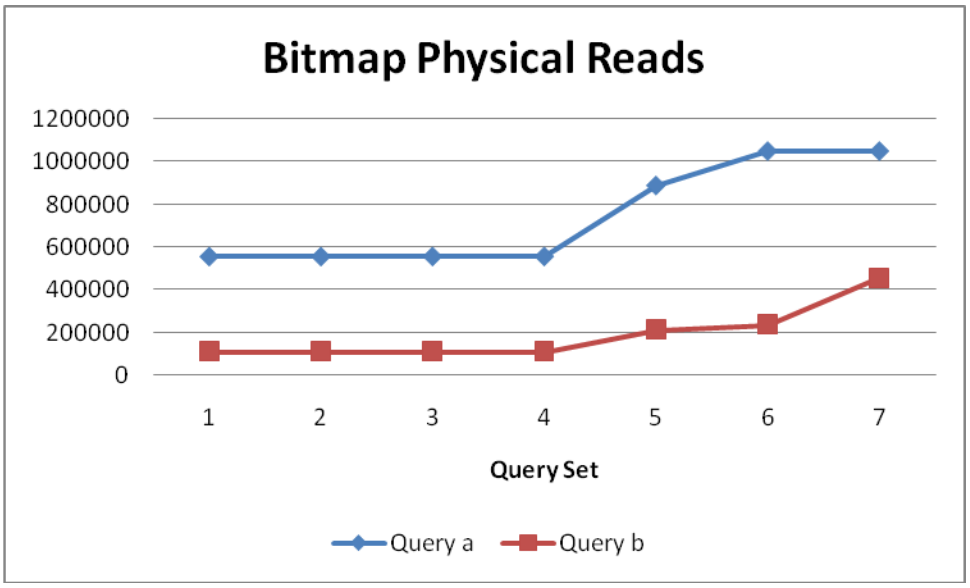


Figure 5-17: Bitmap Physical Reads

The level of performance improvement provided by the bitmap indexes quickly deteriorated when joins were required or when rows needed to be returned instead of counted. The bitmap to rowid conversion process generates additional I/O which can be detrimental to performance.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 4148990278

-------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                             |     1 |    29 |   424   (1)| 00:00:06 |
|   1 |  SORT AGGREGATE               |                             |     1 |    29 |            |          |
|   2 |   BITMAP CONVERSION COUNT     |                             | 84734 |  2399K|   424   (1)| 00:00:06 |
|   3 |    BITMAP AND                 |                             |       |       |            |          |
|*  4 |     BITMAP INDEX SINGLE VALUE| DCUSTB_MARITAL_STATUS_BIX   |       |       |            |          |
|*  5 |     BITMAP INDEX SINGLE VALUE| DCUSTB_INCOME_LEVEL_BIX     |       |       |            |          |
|*  6 |     BITMAP INDEX SINGLE VALUE| DCUSTB_GENDER_BIX           |       |       |            |          |
-------------------------------------------------------------------------------------------------------------
```

Figure 5-18: Execution Plan for a Count Query with No Joins

Figure 5-18 shows the execution plan for a count query that didn't require any joins. Three bitmap index single value scans took place followed by a bitmap AND operation. Note how no table access or bitmap to rowid conversion operations were required as this query was only returning a row count. The index was used to determine how many rows satisfied the criteria in the query's where clause. The results were then counted and returned to the user.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3036180575

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                             |     1 |    46 |       |   174K  (4)| 00:34:50 |
|   1 |  SORT AGGREGATE               |                             |     1 |    46 |       |            |          |
|*  2 |   HASH JOIN                   |                             |    33M|  1491M|  3896K|   174K  (4)| 00:34:50 |
|   3 |    TABLE ACCESS BY INDEX ROWID| DIM_CUSTOMERS_B             | 84734 |  2896K|       | 32760   (1)| 00:06:34 |
|   4 |     BITMAP CONVERSION TO ROWIDS|                            |       |       |       |            |          |
|   5 |      BITMAP AND               |                             |       |       |       |            |          |
|*  6 |       BITMAP INDEX SINGLE VALUE| DCUSTB_MARITAL_STATUS_BIX  |       |       |       |            |          |
|*  7 |       BITMAP INDEX SINGLE VALUE| DCUSTB_INCOME_LEVEL_BIX    |       |       |       |            |          |
|*  8 |       BITMAP INDEX SINGLE VALUE| DCUSTB_GENDER_BIX          |       |       |       |            |          |
|*  9 |    HASH JOIN                  |                             |    33M|   356M|   538M|   103K  (4)| 00:20:37 |
|  10 |     INDEX FAST FULL SCAN      | DIM_ADDRESS_B_PK            |    31M|   179M|       | 14179   (6)| 00:02:51 |
|  11 |     TABLE ACCESS FULL         | FACT_SALES                  |    33M|   162M|       | 32768   (4)| 00:06:34 |
-----------------------------------------------------------------------------------------------------------------------
```

Figure 5-19: Execution Plan for a Count Query with Joins

Figure 5-19 details the execution plan for a count query with two joins. Three bitmap index single value scans took place followed by a bitmap AND operation. A bitmap conversion to rowids then took place and the table was accessed using the rowids.

Because a join was required the rows identified as meeting query criteria in the bitmap indexes had to be accessed so that they could be joined to those in another table. The bitmap conversion to rowids step maps the needed bitmap values in the index to their respective rowids so that they can be accessed during the join operation.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 792722436

-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                            | 84734 |  9681K| 32763   (1)| 00:06:34 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | DIM_CUSTOMERS_B            | 84734 |  9681K| 32763   (1)| 00:06:34 |
|   2 |   BITMAP CONVERSION TO ROWIDS|                            |       |       |            |          |
|   3 |    BITMAP AND                 |                            |       |       |            |          |
|*  4 |     BITMAP INDEX SINGLE VALUE| DCUSTB_MARITAL_STATUS_BIX  |       |       |            |          |
|*  5 |     BITMAP INDEX SINGLE VALUE| DCUSTB_INCOME_LEVEL_BIX    |       |       |            |          |
|*  6 |     BITMAP INDEX SINGLE VALUE| DCUSTB_GENDER_BIX          |       |       |            |          |
-------------------------------------------------------------------------------------------------------------------
```

Figure 5-20: Execution Plan for a Query Retrieving Rows

Figure 5-20 reveals the execution plan for a query that retrieves rows from a bitmap indexed table. As in the two previous examples, three bitmap index single value scans took place followed by a bitmap AND operation. Since rows must be returned in this query, the rows identified as meeting query criteria must be converted to rowids for retrieval from the table. This results in the bitmap conversion to rowid operation and subsequent table access seen in the query execution plan.

Chapter 6 – Denormalization

6.1 Denormalization Results and Analysis

It has been stated in data warehouse design literature that schema denormalization can improve performance, but under what circumstances this performance gain can be realized and its extent are often not mentioned. In an attempt to test the effects of denormalization on a database schema, a set of queries were run on a set of normalized tables designated by the letter n and a single denormalized table designated by the letter d.

Table 6-1: Denormalization Time Measurements

Queries	Number of Joins	Avg. Query Time for Normalized Schema n (seconds)	Avg. Query Time for Denormalized Schema d (seconds)	Time Difference (seconds)	Percentage of Improvement
1n, 1d	6	275	266	9	3%
2n, 2d	12	382	131	251	66%
3n, 3d	18	667	196	471	71%
4n, 4d	18	354	195	159	45%

To start the testing, two queries (Figure 6-1 and 6-2) returning the same columns were run on the normalized and denormalized tables. The queries returned around 655,000 rows from each table with the denormalized table barely outperforming the normalized table by an average of nine seconds. In this case, the denormalized table

demonstrated only a small three percent increase in performance over the normalized tables, and the average processing times were very close.

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from demographics_large
  join sales_total_cust_large
    on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
      join sources_customer_large
        on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
          join customers_norm_large
            on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
              join address_customer_large
                on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                  join countries
                    on address_customer_large.addr_country_id = countries.country_id
                      join history_cust_large
                        on address_customer_large.addr_cust_id = history_cust_large.cust_id
 where addr_state_province = 'CA';
```

Figure 6-1: Query 1n

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from customers_denorm_large
where addr_state_province = 'CA';
```

Figure 6-2: Query 1d

Next, a second set of queries (Figures 6-3 and 6-4) with a more restrictive where clause, were run on the normalized and denormalized tables. Query 2n included twelve joins -twice the number of joins in query 1n, and both second set queries brought back far fewer rows (196) than the first set. Query 2d finished 251 seconds faster than query 2n indicating the denormalized table provided a sixty-six percent reduction in processing time over the normalized tables.

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from demographics_large
  join sales_total_cust_large
    on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
      join sources_customer_large
        on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
          join customers_norm_large
            on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
              join address_customer_large
                on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                  join countries
                    on address_customer_large.addr_country_id = countries.country_id
                      join history_cust_large on address_customer_large.addr_cust_id = history_cust_large.cust_id
 where cust_gender = 'M'
  and cust_year_of_birth = (select min (cust_year_of_birth) from demographics_large
                            join sales_total_cust_large
                              on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
                               join sources_customer_large
                                on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
                                  join customers_norm_large
                                    on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
                                      join address_customer_large
                                        on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                                          join countries
                                            on address_customer_large.addr_country_id = countries.country_id
                                              join history_cust_large
                                                on address_customer_large.addr_cust_id = history_cust_large.cust_id);
```

Figure 6-3: Query 2n

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from customers_denorm_large
  where cust_gender = 'M'
    and cust_year_of_birth = (select min (cust_year_of_birth) from customers_denorm_large);
```

Figure 6-4: Query 2d

The third set of queries (see Figures 6-5 and 6-6) expanded the number of joins from twelve to eighteen over the second set of queries, and returned the same number of rows. Query 3d finished 471 seconds before query 3n resulting in a performance increase of seventy-one percent for the denormalized table.

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from demographics_large
  join sales_total_cust_large
    on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
      join sources_customer_large
        on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
          join customers_norm_large
            on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
              join address_customer_large
                on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                  join countries
                    on address_customer_large.addr_country_id = countries.country_id
                      join history_cust_large
                        on address_customer_large.addr_cust_id = history_cust_large.cust_id
 where cust_gender = 'M'
  and cust_year_of_birth = (select min (cust_year_of_birth) from demographics_large
                              join sales_total_cust_large
                                on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
                                 join sources_customer_large
                                  on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
                                    join customers_norm_large
                                      on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
                                        join address_customer_large
                                          on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                                            join countries
                                              on address_customer_large.addr_country_id = countries.country_id
                                                join history_cust_large
                                                  on address_customer_large.addr_cust_id = history_cust_large.cust_id)
  and cust_credit_limit = (select max(cust_credit_limit)
                              from demographics_large
                                join sales_total_cust_large
                                  on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
                                    join sources_customer_large
                                      on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
                                        join customers_norm_large
                                          on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
                                            join address_customer_large
                                              on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                                                join countries
                                                  on address_customer_large.addr_country_id = countries.country_id
                                                    join history_cust_large
                                                      on address_customer_large.addr_cust_id = history_cust_large.cust_id);
```

Figure 6-5: Query 3n

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from customers_denorm_large
  where cust_gender = 'M'
    and cust_year_of_birth = (select min (cust_year_of_birth)
                               from customers_denorm_large)
    and cust_credit_limit = (select max(cust_credit_limit)
                              from customers_denorm_large);
```

Figure 6-6: Query 3d

The fourth and final set of queries (Figures 6-7 and 6-8) were identical to those used in the third set, only with more restrictive where clauses. Query 4d finished 152 seconds faster than query 4n giving the denormalized table a forty-five percent increase in performance over the normalized tables. The difference in processing times for this set of queries is lower because query 4n completed in nearly half the time as query 3n.

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from demographics_large
  join sales_total_cust_large
    on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
      join sources_customer_large
        on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
          join customers_norm_large
            on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
              join address_customer_large
                on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                  join countries
                    on address_customer_large.addr_country_id = countries.country_id
                      join history_cust_large
                        on address_customer_large.addr_cust_id = history_cust_large.cust_id
 where cust_gender = 'M'
  and cust_year_of_birth = (select min (cust_year_of_birth) from demographics_large
                            join sales_total_cust_large
                              on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
                               join sources_customer_large
                                 on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
                                   join customers_norm_large
                                     on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
                                       join address_customer_large
                                         on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                                           join countries
                                             on address_customer_large.addr_country_id = countries.country_id
                                               join history_cust_large
                                                 on address_customer_large.addr_cust_id = history_cust_large.cust_id
                           where sour_src_type = 'internet'
                             and country_id = '52790'
                             and cust_credit_limit > '3000')
  and cust_credit_limit = (select max(cust_credit_limit)
                           from demographics_large
                             join sales_total_cust_large
                               on demographics_large.cust_id = sales_total_cust_large.sales_cust_id
                                 join sources_customer_large
                                   on sales_total_cust_large.sales_cust_id = sources_customer_large.sour_cust_id
                                     join customers_norm_large
                                       on sources_customer_large.sour_cust_id = customers_norm_large.cust_id
                                         join address_customer_large
                                           on customers_norm_large.cust_id = address_customer_large.addr_cust_id
                                             join countries
                                               on address_customer_large.addr_country_id = countries.country_id
                                                 join history_cust_large
                                                   on address_customer_large.addr_cust_id = history_cust_large.cust_id
                          where addr_state_province = 'CA'
                             and cust_valid = 'I'
                             and cust_marital_status = 'single');
```

Figure 6-7: Query 4n

```
select cust_id, cust_income_level, cust_gender, addr_state_province, country_name, sour_src_type, cust_total_id, cust_valid
from customers_denorm_large
  where cust_gender = 'M'
    and cust_year_of_birth = (select min (cust_year_of_birth)
                              from customers_denorm_large
                              where sour_src_type = 'internet'
                                and country_id = '52790'
                                and cust_credit_limit > '3000')
    and cust_credit_limit = (select max(cust_credit_limit)
                             from customers_denorm_large
                             where addr_state_province = 'CA'
                                and cust_valid = 'I'
                                and cust_marital_status = 'single');
```

Figure 6-8: Query 4d

A look at Table 1 tells us that on average the queries run on the denormalized table finished consistently faster than the queries run on the normalized tables. Figure 6-9 shows that the number of joins in the normalized query increased the query processing times also increased, with the greatest percentage of improvement being recorded after 18 joins were added.

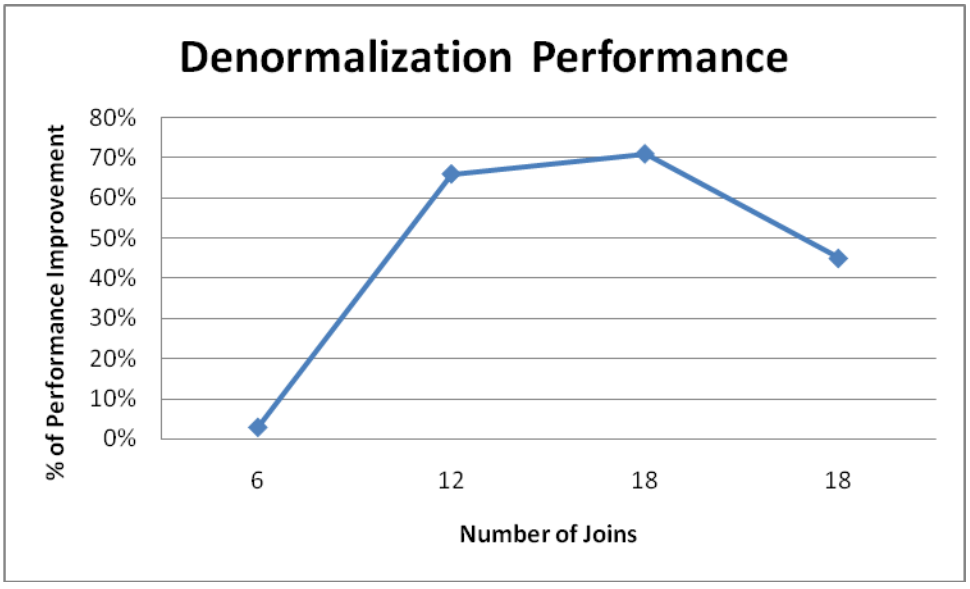


Figure 6-9: Denormalization Performance

A comparison of Table 5-1 with Table 5-2 points to a relationship between the number of joins, number of physical reads, query cost, and the number of sorts. This is because joins generate I/O (both physical and logical) which in turn increases disk access, optimizer cost, the number of sorts that take place, and ultimately query processing time. Queries on a denormalized schema do not require as many joins and as a result can provide a lower I/O cost than a similar normalized schema. Although the same columns/criteria were used in queries on the normalized/ denormalized schemas and the tables contained the same data/number of rows, the I/O overhead of multiple joins meant that the queries on the denormalized schema finished more quickly.

Table 6-2: Denormalization I/O Measurements

Queries	Physical Reads for Norm Schema n	Physical Reads for Denorm Schema d	Cost for Norm Schema n	Cost for Denorm Schema d	Memory Sorts for Norm Schema n	Memory Sorts for Denorm Schema d
1n, 1d	505190	442125	174000	87130	28	4
2n, 2d	1216413	884239	398000	173000	28	4
3n, 3d	1941600	1325925	568000	259000	28	4
4n, 4d	1558974	1326019	394000	261000	29	4

One point of interest is that the optimizer cost and processing time of Query 4n were reduced by nearly half as a result of additional WHERE clauses that were added. These WHERE clauses help to filter the data, causing the rowsets that must be joined together to be much smaller, therefore making the joins less costly. This indicates that the performance of queries on normalized tables can be affected to a great degree by the

types of queries run on them and may play a role in the decision of whether or not to implement a denormalized schema. Queries that are joining smaller rowsets on a normalized schema will not produce as great a difference in performance when compared to similar queries run on a denormalized schema. Figure 6-11 shows the sharp decrease in physical reads between queries 3n and 4n caused by smaller rowsets.

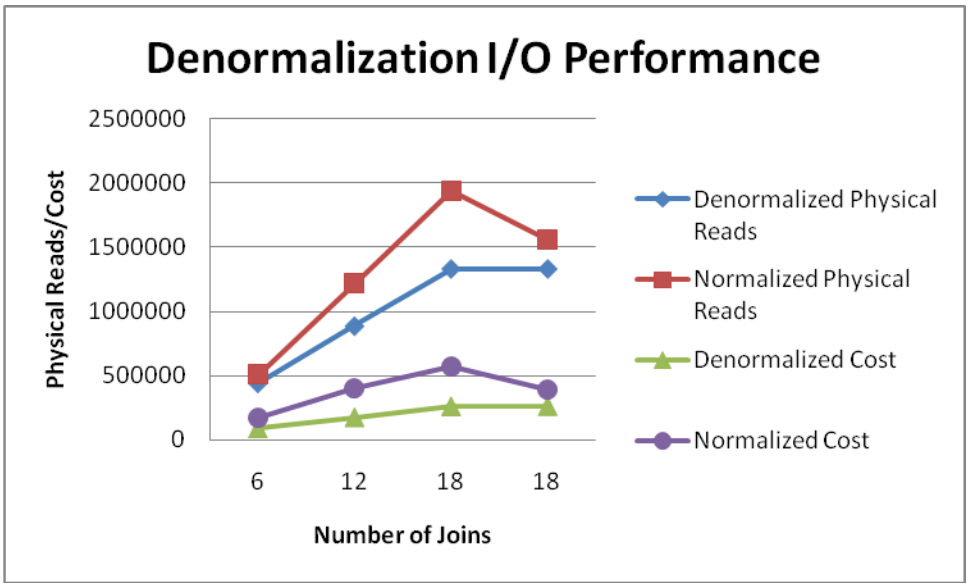


Figure 6-10: Denormalization I/O Performance

6.2 Denormalization Summary

In the previous tests denormalization improved performance by reducing I/O, but might only be practical to implement under certain conditions. Denormalization is most effective at improving performance when queries on a similar normalized schema would require many joins. During the testing conducted for this project minimal gains were seen

at six joins with more substantial improvement being observed between six and twelve joins.

Denormalization is also very effective in improving performance when queries bring back large amounts of unfiltered data. When queries containing multiple WHERE clauses are used, the percentage of performance over a normalized schema is reduced. This is because WHERE clauses filter data so that smaller rowsets are created, which results in joins that are less costly in terms of I/O.

Chapter 7 – Conclusions

The performance improvement techniques discussed in previous chapters have been shown to increase performance by varying degrees based on the types of queries being processed. Certain key query types have the ability to unlock the performance potential of these techniques and provide performance gains of over ninety percent in some cases.

It is important to note that in order to achieve the maximum increase in performance afforded by these techniques they must be implemented in an environment where the key queries for a particular technique are commonly run. For example, at an automotive sales headquarters where the counts of vehicle colors and models that have been sold across the country are regularly retrieved, bitmap indexing could provide a sizeable performance gain.

7.1 Integration

Partitioning, bitmap indexing, and denormalization are compatible with one another and can be used in the same system. This makes them ideally suited to form the foundation of a data warehouse based performance improvement strategy. In fact, they are not only compatible, but can complement one another.

Testing in previous chapters has shown that bitmap indexes must perform a bitmap to rowid conversion to return rows or when a count query contains joins, resulting in additional I/O. Using bitmap indexes in a denormalized schema in which few joins are

required will eliminate this additional I/O and allow count queries to perform at their best.

A denormalized schema will likely still require a few joins to made, but fortunately partitioning and bitmap indexing both lower the size of the resulting rowsets that must be joined together. This means that table joins are not as costly as they would be if full table scans were being performed instead of index/partition scans. The larger the amount of data that must be sorted in memory or on disk means a longer wait for a query to produce results.

Some very large tables could benefit from a combination of all three techniques. Bitmap indexes could provide fast count queries, while queries that return rows may only have to access a single partition. The denormalized structure would eliminate the need for a large number of costly joins, saving unnecessary I/O and therefore time, while the impact of any necessary joins would be lessened by smaller rowsets.

Chapter 8 – Lessons Learned

8.1 Challenges

Setting up the test environment proved to be the most time consuming part of the project. During the process of modifying the Oracle sample schema for use in testing, several technical problems were encountered that slowed things down. Oracle's SQL Developer tool was used to help with preparing the testing environment and it often crashed or stopped responding.

The tool caused problems when inserting large number of rows because after crashing in middle of an insert, the insert would have to be rolled back. The single disk drive would then become very active and other transactions would take much longer to complete. If the insert was tried again before the previous insert had been completely rolled back, massive I/O contention would result, making the system unusable for a considerable amount of time.

After setting up the testing environment and running a few test queries, it became apparent that a larger number of rows would be required to see any difference in performance. The largest of the tables contained only around one million rows and the tables were small enough to be read into memory, making disk access unnecessary. Indexing, denormalization, and partitioning improve performance by reducing disk access, so on such small tables they were not effective. More rows were needed to demonstrate an increase in performance - especially in the case of table partitioning.

8.2 Limitations

The testing environment hardware was limited by the number of processors, amount of disk space, and memory in the test system. The test system supported one Pentium 4 processor and 2GB of memory which was upgraded from 500MB to 1.5 GB. Disk space became an issue in the testing process early on, so the internal hard disk was upgraded to provide 160 GB of storage for the database installation software, documentation, and databases. An external 160 GB drive was also added to store database backups.

The database was initially set up as a data warehouse with common data warehouse parameter settings, including the STAR_TRANSFORMATION_ENABLED parameter set to true. This parameter allows the Oracle optimizer to more efficiently perform joins between fact and dimension tables when certain criteria are met (Lane, Schupmann, & Stuart, 2003). Unfortunately, thoroughly exploring and testing the effect of this parameter on performance was beyond the scope of this project.

References

Artz, J. (1997). How good is that data in the warehouse?. *ACM SIGMIS Database, 28*(3), 21 – 31.

Agrawal, S., Narrasaya, V., & Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (pp. 359 – 370). New York: Association for Computing Machinery.

Agrawal, S., Chu, E. & Narrasaya, V. (2006). Automatic physical design tuning. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (pp. 683 – 694). New York: Association for Computing Machinery.

Bonifati, A., Cattaneo, F., Ceri, S., Fuggetta, A., & Paraboschi, S. (2001). Designing data marts for data warehouses. *ACM Transactions on Software Engineering and Methodology*, *10*(4), 452 – 483.

Bock, D. & Schrage, J. (2002). Denormalization guidelines for base and transaction tables. *ACM SIGCSE Bulletin, 34*(4), 129 – 133.

Chan, C. & Ioannidis, Y. (1998) Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (pp. 355 – 366). New York: Association for Computing Machinery.

Chaudhuri, S. & Dayal, U. (1997). An overview of data warehousing and OLAP technology. *ACM SIGMOD Record, 26*(1), 65 – 74

Goeke, R. & Faley, R. (2007). Leveraging the flexibility of your data warehouse.

Communications of the ACM, 50(10), 107 – 111.

Gorla, N. (2003). Features to consider in a data warehousing system. *Communications of the ACM, 46*(11), 111 – 115.

Graefe, G. (1993). Options in physical database design. *ACM SIGMOD Record, 22*(3), 76 – 83.

Gray, P. & Watson, H. (1998). Present and future directions in data warehousing. *ACM SIGMIS Database, 29*(3), 83 – 90.

Harizopoulos, S., Liang, V., Abadi, D., & Madden, S. (2006). Performance tradeoffs in read optimized databases. In *Proceedings of the 32nd international conference on very large databases* (pp. 487 – 498). New York: Association for Computing Machinery.

Hellerstein, J. & Stonebraker, M. (2005). *Readings in database systems* (4th ed.). Cambridge, MA: Massachusetts Institute of Technology.

Hobbs, L., Hillson, S., & Lawande, S. (2003). *Oracle 9iR2 data warehousing*. Burlington, MA: Digital Press.

Ingram, G. (2002). *High performance oracle: Proven methods for achieving optimum performance and availability*. New York: Wiley.

Inmon, W., Rudin, K., Buss, C., & Sousa, R. (1999). *Data Warehouse Performance*. New York: Wiley

Kimball, R. & Ross, M. (2002). *The data warehouse toolkit: The complete guide to dimensional modeling* (2nd ed.). New York: Wiley.

Kyte, T. (2005). *Expert oracle database architecture: 9i and 10g programming techniques and solutions*. Berkely, CA: Apress.

Lane, P, Schupmann, V., & Stuart, I. (2003). *Oracle 10g database data warehouse guide*. Retrieved December 27, 2007 from http://download.oracle.com/docs/cd/B14117_01/server.101/b10736.pdf

Lightstone, S., Teorey, T., Nadeau, T. (2007). *Physical database design: The database professional's guide to exploiting indexes, views, storage, and more*. San Francisco: Morgan Kaufmann.

March, S. & Hevner, A. (2007) Integrated decision support systems: A data warehousing perspective. *Decision Support Systems 43*(3), 1031 - 1043.

Mallach, E. (2000). *Decision Support and Data Warehouse Systems*. Boston: McGraw-Hill

Martyn, T. (2004). Reconsidering multidimensional schemas. *ACM SIGMOD Record, 33*(1), 83 -88.

Morzy, T. & Wrembel, R. (2004). On querying versions of a multiversion data warehouse. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP* (pp. 92 - 101). New York: Association for Computing Machinery.

Niemiec, R. (2007). *Oracle database 10g performance tuning tips and techniques*. New York: McGraw-Hill

O'Neil, P. & Quass, D. (1997). Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on*

Management of Data (pp. 38 - 49). New York: Association for Computing Machinery.

Palpanas, T. (2000). Knowledge discovery in data warehouses. *ACM SIGMOD Record, 29*(3), 98 - 109.

Powell, G. (2004). *High Performance Tuning for Oracle 9i and 10g*. Burlington, MA: Digital Press.

Powell, G. (2005). *Oracle data warehouse tuning for 10g*. Burlington, MA: Digital Press

Saharia, A. & Babad, Y. (2000). Enhancing data warehouse performance through query caching. *ACM SIGMIS Database, 31*(2), 43 - 63.

Sen, A. & Sinha, A. (2005). A comparison of data warehousing methodologies. *Communications of the ACM, 48*(3), 79 - 84.

Shasha, D. (1996). Tuning databases for high performance. *ACM Computing Surveys, 28*(1), 113 - 115.

Shasha, D. & Bonnet, P. (2003). *Database tuning: Principles, experiments, and troubleshooting techniques*. San Francisco: Morgan Kaufmann.

Stackowiak, R., Rayman, J., & Greenwald, R. (2007). *Oracle data warehousing and business intelligence solutions*. Indianapolis, IN: Wiley.

Stockinger, K., Wu, K., Shoshani, A. (2002). Strategies for processing ad hoc queries on large data warehouses. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP* (pp. 72 – 79). New York: Association for Computing Machinery.

Wrembel, R. & Koncilia, C. (2007). *Data warehouses and OLAP: Concepts, architectures, and solutions*. Hershey, PA: IRM Press.

Wu, M. (1999). Query Optimization for Selections Using Bitmaps. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (pp. 227 – 238). New York: Association for Computing Machinery.

Annotated Bibliography

Artz, J. (1997). How good is that data in the warehouse?. *ACM SIGMIS Database, 28*(3), 21 – 31.

This paper focuses on the use of normalization in data warehouse systems. The author points out several flaws in the normalization process that can contribute to a loss of meaning in warehoused data. Problems with decomposed normalization and synthetic normalization are discussed. The results of an experiment are detailed in which subjects attempt to determine the relationships of database objects.

Agrawal, S., Narrasaya, V., & Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (pp. 359 – 370). New York: Association for Computing Machinery.

In this paper, the authors describe the importance of horizontal and vertical partitioning. They discuss the benefits of these partitioning techniques and propose a method for implementing them in the automatic physical design of a database. The ways that partitioning and other elements of physical design such as indexes interact are also explained.

Agrawal, S., Chu, E. & Narrasaya, V. (2006). Automatic physical design tuning. In *Proceedings of the 2006 ACM SIGMOD International Conference on*

Management of Data (pp. 683 – 694). New York: Association for Computing Machinery.

This paper discusses the use of tools native to a database platform to tune its physical design. More specifically, the paper proposes that the order of queries used when tuning the database can play an important role in getting an accurate simulation of real world database performance. An example is explained in which a data warehouse system is optimized using this technique.

Bonifati, A., Cattaneo, F., Ceri, S., Fuggetta, A., & Paraboschi, S. (2001). Designing data marts for data warehouses. *ACM Transactions on Software Engineering and Methodology*, *10*(4), 452 – 483.

The authors of this paper describe the importance of data warehouse systems and propose a three step method for creating data marts. This method is explained in detail and sample schema diagrams are provided. An introduction to how data warehouses generally function explains how schemas can be designed and the use of star/snowflake schemas.

Bock, D. & Schrage, J. (2002). Denormalization guidelines for base and transaction tables. *ACM SIGCSE Bulletin, 34*(4), 129 – 133.

This paper presents guidelines on how and when to denormalize data. It discusses factors that can cause poor performance and explains how denormalization can be an effective way to substantially improve performance. The paper goes on to

provide advice on how to denormalize data that is in the most common normal forms and also includes a section on denormalizing from the higher normal forms.

Chan, C. & Ioannidis, Y. (1998) Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (pp. 355 – 366). New York: Association for Computing Machinery.

The main focus of this paper is on the different architectures of the bitmap index and how these index architectures can be optimized. The paper presents analytical and experimental results showing the performance of different bitmap architectures. An algorithm improving bitmap performance on certain queries is introduced and bitmap compression is discussed.

Chaudhuri, S. & Dayal, U. (1997). An overview of data warehousing and OLAP technology. *ACM SIGMOD Record, 26*(1), 65 – 74

In this paper the authors provide a broad description of data warehouse systems and techniques. They explain the role that the data warehouse system plays in business intelligence and how such a system can be used effectively by decision makers. Typical data warehouse architecture is discussed as well as the use of denormalized tables, indexes, and materialized views.

Goeke, R. & Faley, R. (2007). Leveraging the flexibility of your data warehouse. *Communications of the ACM, 50*(10), 107 – 111.

This journal article discusses the importance of flexibility in a data warehouse system. The authors suggest increasing system flexibility by expanding the capability of users to perform ad hoc queries and offering additional ways that reports can be displayed. The results from an industry survey are presented to illustrate how several different data warehouse models scored on flexibility and ease of use.

Gorla, N. (2003). Features to consider in a data warehousing system. *Communications of the ACM, 46*(11), 111 – 115.

The author of this journal article presents his ideas on which features are the most essential to have in a data warehouse system. He compares Multidimensional Online Analytical Processing (MOLAP) with Relational Online Analytical Processing (ROLAP) and provides guidelines for using both methods. A section containing points to consider when creating a data warehouse system is also included.

Graefe, G. (1993). Options in physical database design. *ACM SIGMOD Record, 22*(3), 76 – 83.

This paper focuses on different physical design factors affecting database performance. Several physical design techniques including horizontal partitioning, vertical partitioning, compression, and index selection are touched on. A brief discussion of the how physical design factors are interrelated is

included as well as the importance of disk I/O (input/output) in system performance.

Gray, P. & Watson, H. (1998). Present and future directions in data warehousing. *ACM SIGMIS Database*, *29*(3), 83 – 90.

In this journal article, the authors provide a variety of information on data warehouse database systems. They provide a brief description of the purpose behind a data warehouse then proceed to describe its functionality and architecture. The cost limitations of implementing a data warehouse system are examined and predicted trends in data warehousing are presented.

Harizopoulos, S., Liang, V., Abadi, D., & Madden, S. (2006). Performance tradeoffs in read optimized databases. In *Proceedings of the 32nd international conference on very large databases* (pp. 487 – 498). New York: Association for Computing Machinery.

The primary focus of this paper is the design of databases that have been optimized for reading data. The authors propose that column oriented database systems can provide better performance than row oriented systems. They then compare the advantages and disadvantages of both column orientations. Later in the paper, data decomposition is mentioned for use in read-only optimization.

Hellerstein, J. & Stonebraker, M. (2005). *Readings in database systems* (4th ed.). Cambridge, MA: Massachusetts Institute of Technology. Retrieved March 13, 2008 from: http://library.books24x7.com.dml.regis.edu/book/id_10757/viewer.asp?bookid=10757&chunkid=879254663

This book contains a collection of database research written by a variety of authors. It includes sections on query processing, data warehousing, data mining, and database architecture. The data warehousing section contains research on topics such as data cubes, view maintenance, and improving query performance in a data warehouse environment. A section on transaction management presents research on locking with b-tree indexes, as well as how locking affects performance.

Hobbs, L., Hillson, S., & Lawande, S. (2003). *Oracle 9iR2 data warehousing*. Burlington, MA: Digital Press. Retrieved March 13, 2008 from: http://library.books24x7.com.dml.regis.edu/book/id_5958/viewer.asp?bookid=5958&chunkid=0000000001

The authors of this edited book provide advice on how to design and tune a data warehouse system utilizing the Oracle RDBMS (Relational database Management System). They cover partitioning, indexing, compression, tuning, disaster recovery, and many other topics. The section on query optimization is particularly interesting and an explanation of indexes illustrates the concept of the bitmap

index architecture very well. Several partitioning techniques available in Oracle 9i are explained and sample code is provided for creating the partitions.

Ingram, G. (2002). *High performance oracle: Proven methods for achieving optimum performance and availability*. New York: Wiley.

Describing proven methods to achieve optimum performance on an Oracle database system is the purpose of this book. The author starts by explaining the fundamentals of an Oracle system including software installation and database setup. He moves into a series of chapters on performance tuning that covers topics such as partitioning, indexes, managing growth, stress testing, and the use of various performance management tools. The chapter on managing indexes is very thorough and a section on database tuning fundamentals provides a great deal of practical information.

Inmon, W., Rudin, K., Buss, C., & Sousa, R. (1999). *Data Warehouse Performance*. New York: Wiley

This book describes how to design a high performance data warehouse system. The book does not describe how to build a data warehouse system on a particular database platform and doesn't mention technical details. Instead, it provides general advice on designing a data warehouse architecture and explains how the various components of the system work together. Data marts, data cleansing, and

monitoring are covered, as well as the hardware required to support a high performance data warehouse system.

Kimball, R. & Ross, M. (2002). *The data warehouse toolkit: The complete guide to dimensional modeling* (2nd ed.). New York: Wiley.

Dimensional modeling is the focal point of this edited book. The authors introduce the concept of dimensional modeling and how to use it to design an effective data warehouse system. The book doesn't mention specific database platforms or get into the technical details of implementing the dimensional model. It mainly discusses the high level design of a data warehouse system using dimensional structures which can contain denormalized data.

Kyte, T. (2005). *Expert oracle database architecture: 9i and 10g programming techniques and solutions*. Berkely, CA: Apress.

This edited book authored by Oracle expert Tom Kyte provides a summary of practical information about the Oracle RDBMS. The sections on bitmap indexes and partitioning are of particular interest. Kyte illustrates how partitioning can improve system performance and how the different partitioning methods work. He provides sample PL/SQL scripts to set up partitioned tables and tips for using partitioning effectively in different situations.

Lane, P, Schupmann, V., & Stuart, I. (2003). *Oracle 10g database data warehouse guide*. Retrieved December 27, 2007 from http://download.oracle.com/docs/cd/B14117_01/server.101/b10736.pdf

This guide to data warehousing with the Oracle Relational Database Management System (RDBMS) covers a wide variety of topics. It includes sections on logical design, physical design, disk I/O, partitioning, indexes, system maintenance, and system performance. The sections providing information on the use of bitmap indexes and partitioning to improve system performance are especially informative.

Lightstone, S., Teorey, T., Nadeau, T. (2007). *Physical database design: The database professional's guide to exploiting indexes, views, storage, and more*. San Francisco: Morgan Kaufmann.

This book provides a glimpse into the complex world of physical database design. It explores how the physical design of a database can affect its performance. Of particular interest is a chapter detailing physical design considerations for data warehousing and decision support systems. The book covers server resources and configuration, as well as network based storage systems. A chapter on denormalization presents a strategy for denormalizing data and gives an example of a denormalized schema.

March, S. & Hevner, A. (2007) Integrated decision support systems: A data warehousing perspective. *Decision Support Systems 43*(3), 1031 - 1043.

How data warehouse systems can be used in decision making is the main focus of this paper. It describes what information should be provided by the warehouse from the viewpoint of decision makers. Data warehouse architecture and integrating a data warehouse into existing information systems are also discussed.

Mallach, E. (2000). *Decision Support and Data Warehouse Systems*. Boston: McGraw-Hill

This is an entry level decision support/data warehousing book that does a good job of introducing the concepts and vocabulary used in these types of systems. Each chapter contains case studies meant to help to illustrate the ideas presented. The book explains the need for decision support/data warehouse systems in business and covers system modeling, architecture, and design. A chapter on data quality stresses the importance of having high quality data.

Martyn, T. (2004). Reconsidering multidimensional schemas. *ACM SIGMOD Record, 33*(1), 83 -88.

The main idea behind this paper is that the technique of denormalizing database tables in a data warehouse database to increase performance should be done with care and only when necessary. The author points out a few problems with using denormalized tables such as a loss of meaning, and problems with updates in some systems. Guidelines to help database designers decide whether or not to denormalize are also presented.

Morzy, T. & Wrembel, R. (2004). On querying versions of a multiversion data warehouse. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP* (pp. 92 - 101). New York: Association for Computing Machinery.

The authors of this paper describe a data warehouse system that is able to change along with production systems. They propose a system that maintains different versions of a data warehouse structure and data. In order to access these multiple versions within the system, a special extended query language is presented. The authors also describe the prototype system they have built to test their ideas.

Niemiec, R. (2007). *Oracle database 10g performance tuning tips and techniques.* New York: McGraw-Hill

The objective of this book is to provide effective methods for tuning the Oracle 10g RDBMS. The book explains query tuning in great detail and has a thorough section on index tuning. A chapter on disk configuration and the tuning of database structures does a good job of illustrating why these key tuning areas greatly affect system performance. The author also discusses the V$ views, X$ tables, the statspack utility, and how to tune the system using certain initialization parameters.

O'Neil, P. & Quass, D. (1997). Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (pp. 38 - 49). New York: Association for Computing Machinery.

Using variant indexes to increase performance is the focus of this paper. The paper explores several types of indexes and how they function in the typically read-only environment of the data warehouse. The performance enhancing characteristics of the different indexes are described as well as the tasks for which each is the most suited.

Palpanas, T. (2000). Knowledge discovery in data warehouses. *ACM SIGMOD Record, 29*(3), 98 - 109.

This paper explains how key information can be effectively mined when working with very large amounts of data, such as those found in data warehouse systems. The author describes the need for new tools and techniques that are specifically designed for data mining using data warehouse systems. Sections on mining with data cubes and association rules are also presented.

Powell, G. (2004). *High Performance Tuning for Oracle 9i and 10g*. Burlington, MA: Digital Press.

Database tuning for performance improvement is the focus of this edited book. The author walks the reader through various tuning tools available in the Oracle RDBMS including tkprof, SQL trace, statspack, dynamic performance views, and the explain plan command. He also gives advice on how to read execution plans and decipher the statistics generated by the statspack utility. The book contains a section on hardware tuning and provides suggestions for tuning the low level physical structures in the database for maximum performance.

Powell, G. (2005). *Oracle data warehouse tuning for 10g*. Burlington, MA: Digital Press

Tuning an Oracle database in a data warehouse environment is the idea behind this edited book. The chapters in this book are well organized and cover a variety of tuning techniques. The author explains the architecture of data warehouse systems and how this architecture can affect tuning. SQL query tuning, partitioning, and parallel processing are discussed, as well as how materialized views can be utilized to reduce processing time for aggregate data. A section on improving the performance of materialized views is also included.

Saharia, A. & Babad, Y. (2000). Enhancing data warehouse performance through query caching. *ACM SIGMIS Database*, *31*(2), 43 - 63.

Improving the performance of a data warehouse system through the use of query caching is the primary concept behind this paper. The author proposes that an efficient query cache could improve query response times for ad hoc queries. An

explanation of how different types of queries would be positively affected by the caching system is provided. Algorithms for detecting subsumption and query modification are also described.

Sen, A. & Sinha, A. (2005). A comparison of data warehousing methodologies. *Communications of the ACM, 48*(3), 79 - 84.

This journal article provides a comparison of various data warehousing methodologies. The authors describe keys tasks involved in designing a data warehouse system and how they fit into the stages of the development process. The strengths/weaknesses of each methodology are discussed as well as how they are categorized. Each methodology is rated based on certain attributes that are commonly seen in a data warehouse environment.

Shasha, D. (1996). Tuning databases for high performance. *ACM Computing Surveys, 28*(1), 113 - 115.

This paper is about tuning database systems to achieve the best performance possible. Transaction tuning techniques are presented and methods for tuning concurrency control are discussed. The purpose of concurrency control algorithms is explained and techniques are detailed for avoiding problems with deadlock situations. Partitioning and sparse clustering indexed are also discussed.

Shasha, D. & Bonnet, P. (2003). *Database tuning: Principles, experiments, and troubleshooting techniques*. San Francisco: Morgan Kaufmann.

The authors of this edited book introduce database tuning techniques and provide troubleshooting guidelines for many common database performance problems. Two chapters apply to data warehouse environments. The first chapter focuses on how data warehouses are used by companies to make decisions, while the second chapter provides advice on tuning these systems. The book includes a section that presents database tuning case studies from the author's experiences in a financial environment. The case studies discuss how various real world database tuning problems were solved and contain a good deal of practical tuning advice.

Stackowiak, R., Rayman, J., & Greenwald, R. (2007). *Oracle data warehousing and business intelligence solutions*. Indianapolis, IN: Wiley.

This book explains how data warehousing works in an Oracle environment. The text provides an introduction to the data warehousing tools and features available in the Oracle RDBMS. A section on tuning and monitoring discusses strategies for improving performance using the available Oracle tools. Although the book specifically focuses on data warehousing in an Oracle environment, it does not go into a great amount of technical detail. Instead, the authors provide more of a high level overview of what data warehousing on an Oracle system entails.

Stockinger, K., Wu, K., Shoshani, A. (2002). Strategies for processing ad hoc queries on large data warehouses. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP* (pp. 72 – 79). New York: Association for Computing Machinery.

In this paper the authors describe techniques for running ad hoc queries on large data warehouse systems. The use of bitmap indexes to improve performance is discussed and the effect of compression on bitmaps. Several types of bitmap indexes are introduced and their effect on performance in various scenarios explained. Performance gains using vertical partitioning are detailed as well as the advantages gained when use vertical partitioning over b-tree indexes.

Wrembel, R. & Koncilia, C. (2007). *Data warehouses and OLAP: Concepts, architectures, and solutions*. Hershey, PA: IRM Press.

This text covers a variety of data warehousing topics, from database design to system tuning. Star schema queries are discussed and a brief section on star query optimization is presented. A chapter on tuning bitmap indexes is particularly interesting and goes into depth on subjects such as bitmap compression and binning. The bitmap index offerings of major database vendors are also summarized toward the end of the chapter.

Wu, M. (1999). Query Optimization for Selections Using Bitmaps. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (pp. 227 – 238). New York: Association for Computing Machinery.

This paper focuses on strategies for using bitmap indexes to improve the speed of database queries. The author describes static and dynamic query optimization and describes the attributes of each classification. He goes on to propose a tree reduction technique he developed to improve the performance of an algorithm used in bit sliced indexes. Analytical and probabilistic cost models are also proposed which the author claims will lead to improved query execution plans.