# Comparing the performance of normalized database and denormalized data warehouse for simple one-dimensional and complex multi-dimensional queries

Postgraduate Diploma in Commerce

Dissertation

Under the supervision of Dr. Nigel Stanger

Jiawei Liu

University of Otago

October 12, 2018

# Abstract

Normalized databases and denormalized data warehouses are two main types of databases in modern businesses. In this research, we will compare the performance of these two kinds of databases when they are used for simple one-dimensional queries and complex multi-dimensional queries respectively in PostgreSQL. To be more precise, we will concentrate on comparing the response time and query throughput test, and using memory usage and query execution plan to understand and explain the difference of performance. The aim of this research is to investigate how normalization and denormalization affect the performance of different types of queries and the findings may instruct the future databases design in companies, governments and other institutions. Our experiment results indicate that the normalized database tends to be faster for simple queries because it can derive more data from the cache rather than the disk during the execution process. The denormalized data warehouse tends to be faster for complex queries due to its denormalized design. However, it is difficult to have a unified conclusion about which is faster. The differences between their performance become relatively smaller compared to prior researches, and the average workloads of both normalization and denormalization are reasonably similar. In general, we suggest that database designers should carefully think about their demands before choosing a normalized or denormalized schema.

Key words: Database, Data warehouse, Normalization, Denormalization, Query execution, Query throughput test

# Contents

# List of Tables

# List of Figures

# Chapter 1: Introduction

Data nowadays have become a more and more significant resource for business. With the fast-increasing amount of data, it can be seen that companies as well as governments intend to build databases to store and manage their data. “A database is a persistent partial model of reality that can be manipulated by application programs”, and it is usually used to store and manage data (Rochkind, 2013, p.83). With the development of databases, more and more data are produced through online transaction processing. A model is needed to analyse these data and support the decision-making. Bill Inmon, known as “the father of the data warehouse”, proposed the concept of data warehouse in the 1990s. He states that “a data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management’s decisions” (Inmon, 2005, p.29). In general, the main objective of database is to find the structure that can optimise the storage performance, in other words, store more data with less space. However, a data warehouse is built to support decision-making, which means that it is built with some particular schemas, such as star schema and snow flake schema, to answer analytical queries quickly and accurately. Therefore, a data warehouse developer requires a clear understanding of the requirements and subjects of the users because the ways data are organized in a data warehouse is subject-oriented and they can be different for different usages.

The conception and comparison of database and data warehouse have been investigated since 1990s. There were some research reports about the different performance between the database and data warehouse with relatively old versions of database system in 1990s and 2000s. For example, Hahnke (1996) argued that the denormalized data warehouse can improve the performance for complex queries and other business analytical processing compared to the normalized database. However, in recent years, there is little academic literature in this area which uses more recent versions of database software. Therefore, this research aims to investigate whether these previous findings are still true after 20 years of further development of the database management system. We will compare the performance of these two types of databases when they are used for simple one-dimensional queries and complex multi-dimensional queries respectively on a recent version of PostgreSQL. The performance will be measured and estimated by comparing query execution plans, the size of memory queries uses, single query response time and query throughput tests.

Our research questions are:

- Research question one: what are the differences of performance between the normalized database and denormalized data warehouse when running simple and complex queries?
- Research question two: has the difference between their performance changed compared to prior works?
- Research question three: is there any difference between normalization and denormalization with regards to query throughput?

Normalization and denormalization can influence the database performance of queries. However, there are also other factors which can affect the query response time, such as poor conceptual and physical design of the database, insufficient hardware of the working system and poor use of the database management system (Bock and Schrage, 2002). Therefore, these factors should be controlled if possible or at least measured and recorded when we conduct our experiment.

In our research, we demonstrate that: (i) the normalized database usually has better performance for simple queries while the denormalized data warehouse usually has better performance for complex queries but it is difficult to produce a generalizable conclusion; (ii) the difference between their performance has shrunk compared with previous literature; (iii) the average workloads for database and data warehouse under a realistic working are reasonably similar.

In chapter 2, we provide an overview of background studies and related works on normalization, denormalization, database, data warehouse and query performance measurement. Chapter 3 defines our experiment and performance methodology on a set of queries to compare the performances of normalized and denormalized database. Chapter 4 demonstrates and discusses the experimental results. Finally, chapter 5 concludes the paper.

# Chapter 2: Background and related works

## 2.1 Database and data warehouse

According to Date (2004, p.11) "a database is a collection of persistent data that is used by the application systems of some given enterprise". Here, the term "enterprise" is simply a convenient generic term for any reasonably self-contained commercial, scientific, technical, or other organization. It might be a single individual (with a small personal database), or a complete corporation or similar large body (with a large shared database), or anything in between. With the development of database techniques, more and more data are produced through online transaction processing. A model is needed to analyse these data and support the decision-making system. Bill Inmon, known as the father of the data warehouse concept, gave the definition of the data warehouse. "A data warehouse is a subject-oriented, integrated, non-volatile, and time-variant collection of data in support of management's decisions" (Inmon, 2005, p.29). To make it clear, the data warehouse is subject-oriented because different companies have different subject areas and when they implement their own data warehouse, it should be designed depending on the subject area. Inmon (2005) then argued that integration is the most important aspects of a data warehouse because the data in the data warehouse are fed from multiple sources. Therefore, these data need to be processed, such as re-formatted, re-sequenced and summarized before loading into the data warehouse. In terms of non-volatile, he (2005) further explained that data in the database is updated while data in the data warehouse is loaded in a snapshot, static format. When subsequent changes appear, a new snapshot record is produced and loaded into the data warehouse, and these historical records would still be kept in it. Time-variant means that "every unit of data in the data warehouse is accurate as of some moment in time" (Inmon, 2005, p.32), and it will keep sophisticated snapshots of data of various moment. The main aim of the data warehouse is to support decision-making. Therefore, a data warehouse developer requires a clear understanding of requirements and objectives of the users, and its schema can be different for different usages.

Shin (2002) argued that the data warehouse has become a central part of decision support-oriented data management. He also pointed out that in order to provide reliable and useful data to decision-making system, it is necessary to combine different sources of operational data. In addition, Chohan and Javed (2010) conclude that data for online analytical processing (OLAP) are usually stored in data warehouses and mainly used for making high level managerial decisions while data for online transaction processing (OLTP) are usually stored in "conventional"

databases and mainly used for making relatively low level operational decisions. Sen and Jacob (1998) stated that, according to their surveys and statistics, there was increasing development and use of data warehouses in the late 1990s, and the number of enterprises which already had or were building data warehouses or decision supporting systems was increasing considerably. Some large companies had more than one database and data warehouse. Shin (2002) demonstrated that 85 percent to 90 percent of the Fortune 500 companies have accepted and built data warehouses.

## 2.2 Normalization and denormalization

Because of the different usages of database and data warehouse, which is mentioned in last section, they also have different schema structures and different building strategies. Specifically, an OLTP database is usually normalized while a data warehouse is usually denormalized. According to Lightstone, Teorey and Nedeau (2007, p.337), normalization can be defined as a process which splits a table into smaller tables "to eliminate unwanted side effects of deletion of certain critical rows and to reduce the inefficiencies of updating redundant data often found in large universal tables". Codd (1990), who first raised the concept of normalization, stated that normalization can find an optimal logical structure to simplify the design of a relational database by classifying these attributes into groups for better performance in storage. In other words, normalization aims to convert a large table into several smaller tables by applying various normal forms (1NF through 6NF) to reduce redundant data, which would optimize storage performance. These smaller tables are connected by foreign keys and can be searched by join operations. Normalization can be a priority choice in storing data. However, Lightstone, Teorey and Nedeau (2007) argued that sometimes the structure of normalized databases can be complex with many extra joins and some queries become considerably inefficient because of these joins, especially when we do relatively complicated queries which contain aggregation or are designed for analytical subjects (Zaker, Phon-Amnuaisuk and Haw, 2008). For these cases, another strategy for designing database, denormalization, is proposed.

Denormalization can be defined as a process which reduces "the degree of normalization with the aim of improving query processing performance" (Sanders and Shin, 2001, p.1). They state the main objective for denormalization is to reduce the number of joining operations to decrease the number of tables that need to be accessed to obtain the desired data to answer queries. Additionally, some columns with pre-aggregated data can also be added during the denormalization process,

depending on the kinds of queries to be asked. Although it is said that denormalization can make the data warehouse have considerably better performance for doing complex queries (Zhang, Zhou, Zhang, Zhang, Su and Wang, 2016), database designers cannot use denormalization blindly without clear objectives but should carefully consider the balance of performance demand for storing data and doing queries as well as the purpose of building the database (Shin and Sanders, 2006).

Rajakumar and Raja (2015) demonstrated and compared the structures of different denormalization architecture:

- Star schema: a fact table in the middle, which is connected to a set of dimension tables. The fact table demonstrates aims of data warehouse users, and data in the fact table is highly aggregated, and it is usually the most crucial parts in the data warehouse. For example, for a retailer, the fact table usually aggregated stores sale data. Dimensions are the further extensions of the fact table from different aspects. For example, for a sale fact table, it is usually connected to time dimension, location dimension and product dimension.

- Snowflake schema: a refinement of star schema where some dimensional hierarchy is further splitting into a set of smaller dimension tables, forming a shape similar to snowflake.

- Fact constellations: multiple share dimension tables where viewed as a collection of stars.

- Concept hierarchies: grouping values for a given dimension or attribute, resulting in a set-grouping hierarchy.

Bock and Schrage (2002) pointed out that there are a variety of factors which can influence query response time, such as poor conceptual and physical design of the database, insufficient hardware of the working system and poor use of the database management system. In their article, they concentrate on how to denormalize the database, run denormalized queries and how denormalization affects the system response time. They concluded that the denormalization which they used to modify client-server systems could improve the efficiency of system response (Bock and Schrage, 2002).

Hahnke (1996) demonstrated some positive influences of denormalization on business analytical processing. In his research, he employed a more denormalized data structure to build an analytical system for business. Facts and dimensions were utilized in their data model to solve the complicated hierarchy issues which are a

significant component of multi-dimensional analysis. He also argued that the logical data model can be the most essential element of creating a business analysis application as well as the most important determinant of application performance (Hahnke, 1996).

However, Date (1998) stated that denormalization should not be done at the logical level but at the physical storage level. He further discusses that denormalization may also have detrimental influences on the clear logical system and produce undesirable consequences on it.

Cerpa (1995) proposed a conception of a pre-physical database design process which was an intermediate step between logical and physical processing. It was used to analyze the purpose of designing a database according to the requirements of the application. When highly complex database was designed, an automated tool provided requirement analysis support for database designers in terms of physical level design.

Coleman (1988) provided some potential drawbacks of denormalization. For example, denormalization usually contains trade-offs between flexibility and performance and requires a reasonably comprehensive understanding of demands before designing. He also claimed that database designers should understand how the database management system, the hardware and the operating system coordinate to optimize denormalized performance.

## 2.3 Tools and techniques for building database and data warehouse

There are many relational database management systems (DBMS) to selected, such as Oracle, PostgreSQL and MySQL. PostgreSQL is chosen as our testing DBMS because it is freely available, easy to install and configure, widely used and has considerably strong relational features. Jepson (2001) argued that PostgreSQL provides more features than MySQL, such as server-side procedural languages and more SQL functions. He also claims that because of the limited feature set of MySQL, it can be considerably faster. He concluded that PostgreSQL can handle business logic on the database server. Therefore, it is a better choice if the database is built to solve a problem with complicated business issues.

Mlodgenski (2010) proposed some other advantages of PostgreSQL. For example, PostgreSQL has the oldest, largest and fastest-growing database community. It is long established, being first built by University of California, Berkeley in 1985. It is obvious that a large community can provide more interaction opportunities for users

and developers, which can even provide users with some opportunities to design new features directly. He also states that the main strength of PostgreSQL is the safety guarantee of storing the data as well as some constraints which can be defined by users to ensure data quality when the database is modified and updated.

Both normalized databases and denormalized data warehouses are relational databases, which means that these two types of databases are organized on the relational model of data. As previously discussed, a normalized database aims at reducing redundant data and having better storage performance. In order to find the normal forms for different data sets, many researchers have been working in this area since Codd initiated this subject (Lee, 1994). Then, a series of normal forms is proposed such as three normal forms and the Boyce-Codd Normal Form. Lee (1994) also argues that higher normal forms may increase the maintenance cost or decrease the system performance. Therefore, database designers cannot blindly chase high levels of normalization but should find cost-effective normal forms depending on different data sets and actual situations.

However, denormalization processes are more complex than normalization processes because designers need to have clear understanding of demands and objectives before building denormalized data warehouses. Bock and Schrage (2002) conclude that designers must accomplish denormalization in conjunction with a detailed analysis of the tables required to support various object views of the database. Shin and Sanders (2006) also introduce various denormalization strategies to increase data warehouse performance including:

- Adding redundant attributes: data from one relation is accessed in conjunction with a few columns from another table.
- Derived attributes: frequently used aggregates are precomputed and materialized in an appropriate relation.
- Collapsing relations: combining two relations in a one-to-one relationship.

These approaches can be reasonably useful to guide designers building their data warehouses.

## 2.4 Measuring database performance

This section discusses prior work on measuring the performance of databases. There are various types of benchmarking measures which can be selected respectively

depending on different demands, and TPC (Transaction Processing Performance Council) benchmark is one of the most widely used benchmark approaches. There were various TPC benchmark methods published by TPC since late 1990s (TPC,1998). TPC-C and TPC-E are both used for evaluating OLTP. TPC-DI is designed for data integration while TPC-DS measures not only query performance but also data maintenance performance and data processing system configuration. In general, TPC-H benchmarking is the most suitable model in our research considering features and usages of different models.

TPC-H benchmarking is a decision support benchmark that consists of a series of business-oriented ad-hoc queries. This benchmark is intended for decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric, and reflects multiple aspects of the capability of the system to process queries. These aspects include the selected database size against which the queries are executed, the query processing power when queries are submitted as a single stream, and the query throughput when queries are submitted by multiple concurrent users (TPC, 2018), which is helpful for us to answer research question 3. TPC-H benchmarking is composed of various queries ranging from simple ones to complex ones. Furthermore, all these queries can be changed and modified by users to make them more suitable for estimating individual databases (Ngamsuriyaroj and Pornpattana, 2010)

TPC corporation provides a list of criteria to set queries for TPC-H benchmarking (TPC, 2017). TPC queries:

- Give answers to solve real-world business questions
- Simulate generated ad-hoc queries
- Are much more complex than most OLTP transactions
- Include a variety of operators and selectivity constraints
- Generate intensive activity on the part of the database server component of the system under test

After finalising the type of benchmarking in our experiment, performance measure variables should be discussed and decided. Darmont, Bentayed and Boussaid (2017) used memory and single query execution time to evaluate the performance of data warehouses with different schemas. Memory and single queries execution time are

two simple but important aspects, which can reflect the performance of a database clearly and directly. Rahman (2013) evaluated data warehouse SQL query performance in four aspects: CPU evaluation, I/O evaluation, spool evaluation (usage of temporary storage) and explain evaluation (query execution plan). The query execution plan is also significant because it can show how databases and data warehouses differ in terms of the way they solve queries. Finally, we decide to understand and discuss database performance from four aspects: single query response time, memory usage, query execution plan and query throughput test, which can be analysed to answer our research question 1 and 2.

# Chapter 3: Methodology

## 3.1 Building the normalized database and denormalized data warehouse

In order to compare efficiency of the normalization and denormalization processes and analyze the performance of these two approaches, we built two databases with normalized and denormalized design in PostgreSQL.

### 3.1.1 Normalized database

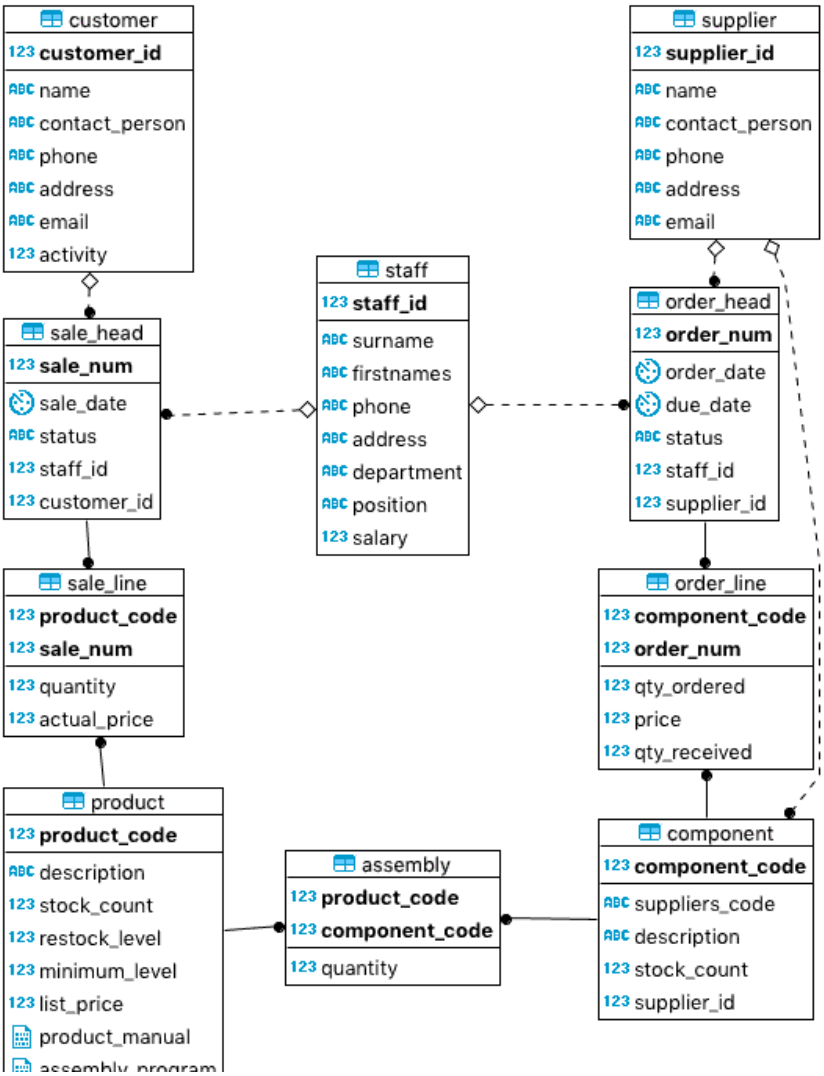



Figure 1: Schema of the normalized database

In our research, it was a database teaching scenario used for several years in the paper INFO 321, and this normalized database is used by a digital product company that produces a wide range of electronics devices which are sold in bulk to various retailers. This company also has a variety of component suppliers, and it could select the most suitable and profitable supplier according to its different target products. It can be seen from Figure 1 that this database has ten tables that can store data about various aspects of the company's business. Data in this normalized database is randomly generated with some constraints to simulate realistic data. For the purposes of this research, we used the tables related to sales, as they contained the most data. To be more precise:

- The staff table: this table contains personal information of all staff in this company, and it has a primary key of staff_id with 6000 rows, in other words there are 6000 employees in total. Staff would need to deal with both sales requests from customers and orders to suppliers.
- The customer table: this table contains customer information including contact details and activity of sales requests. This table has 75000 rows with a primary key of customer_id.
- The product table: this table stores the description, various prices, manual and current stock of 3538 different products that are currently manufactured and available in this company. In our experiment, the value for all the rows of product_manual column is 0 because it is hard to randomly generate them, and this column is never queried or joined in our research.
- The sale_head table: this table has basic information of every sale request, such as sale_date, current status, customer_id who makes the request and staff_id who deals with this request. It has a a primary key of sale_num with 700000 rows, and each sale_num means one particular sale request.
- The sale_line table: this table holds the detail of which products and how many of each is requested for each sale_num. It is worth mentioning that the sale_line is the biggest table in our experiment with more than 5200000 rows.

### 3.1.2 Denormalized data warehouse

After the normalized database was built, the data from the tables described above (including the customer table, the staff table, the product table, the sale_head table and the sale_line table) was derived to build the denormalized data warehouse. The

data that is not relevant to transaction processes has been dropped, such as phone and address of customers and staffs. The schema of denormalized data warehouse in our experiment is shown in Figure 2:

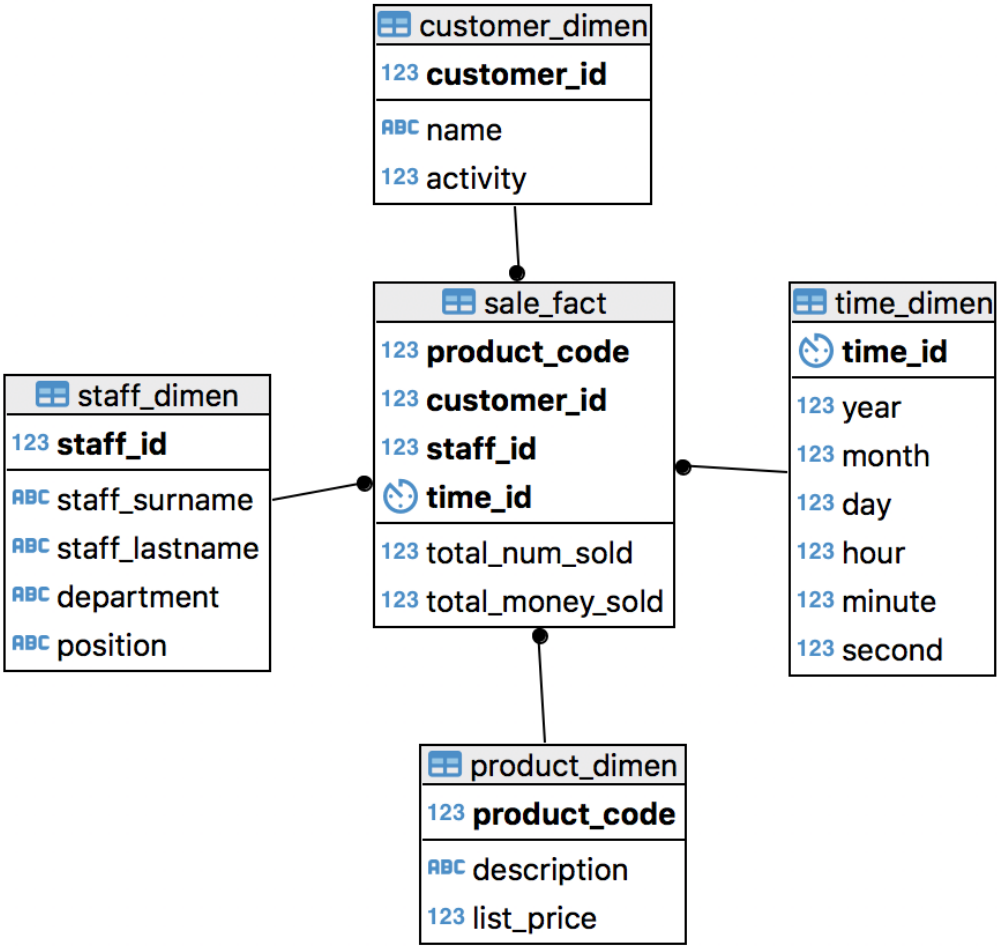



Figure 2: Schema of the denormalized data warehouse

The star schema is used in this data warehouse, and it has one fact table, sale_fact, in the middle connected to four dimension tables. It is the most basic but classical schema for data warehouse, and other complicated schemas are reasonably similar to it or even refined and developed from it directly, such as the snowflake schema (Rajakumar and Raja, 2015). The details of these tables are as follows:

- The sale_fact table: this table has one composite primary key with four columns, including product_code, customer_id, staff_id and time_id. These could show which products are in the transaction, who makes and who deals with the transaction and when the transaction happens, which might be the most significant four variables in transaction processes. The details of these four aspects are stored in the four dimension tables. It can be also found that the total

quantity and price of each product sold for every product in every transaction have been pre-calculated because these two variables might be the most common variables in queries and the pre-calculation can considerably reduce the query response time. This introduces redundancy and can be considered a form of denormalization. The sale_fact table has more than 5200000 rows.

- The product dimension: this dimension has a primary key of product_code and two other columns with 3538 rows. The description contains basic information of the product and the list price shows the price that the company is willing to sell the product.
- The staff dimension: this dimension contains the basic data of all staff with a primary key of staff_id and 6000 rows.
- The customer dimension: this dimension contains the basic data of all customers with a primary key of customer_id and 75000 rows.
- The time dimension: this dimension has a primary key of time_id, and all the data of year, month, week, day, hour, minute and second have been pre-extracted from sale_date in the normalized database, which might make these time-related queries reasonably faster. The size of this table is 700000 rows.

## 3.2 Query set

In order to implement the TPC-H benchmarking on the test database and data warehouse, test queries should be set. In our experiment, 12 queries have been set to evaluate the performance of different query types including aggregation, sorting, ranking and join queries. The first six are simple queries (meaning no joins) and the remaining six are complex queries (meaning one or more joins). Each one of them has a database version and data warehouse version, and it has been verified that for all these queries these two versions produce exactly the same results. All these queries would make sense in a real world situation and brief descriptions of them are shown in Table 1 (simple queries) and Table 2 (complex queries):

| Simple query No.1 | Database version | Data warehouse version |
|---|---|---|
| Query the number of sales for each product | select product_code, sum(quantity) as total_num_sold_per_product<br>from sale_line<br>group by product_code<br>order by total_num_sold_per_product; | select product_code,<br>sum(total_num_sold) as total_num_sold_per_product<br>from sale_fact<br>group by product_code<br>order by total_num_sold_per_product; |

| Simple query No.2 | Database version | Data warehouse version |
|---|---|---|
| Query the number of sales for each month | select extract(month from sale_date) as month, count(sale_num) as month_sale_num<br>from sale_head<br>group by month; | select month, count(time_id) as month_sale_num<br>from time_dimen<br>group by month; |

| Simple query No.3 | Database version | Data warehouse version |
|---|---|---|
| Query managers of every department | select staff_id, department, position<br>from staff<br>where position like 'Manager'; | select staff_id, department, position<br>from staff_dimen<br>where position like 'Manager'; |

| Simple query No.4 | Database version | Data warehouse version |
|---|---|---|
| Query all products which have a list price between 50 and 100 | select product_code, description, list_price<br>from product<br>where list_price between 50 and 100<br>order by list_price; | select product_code, description, list_price<br>from product_dimen<br>where list_price between 50 and 100<br>order by list_price; |

| Simple query No.5 | Database version | Data warehouse version |
|---|---|---|
| Query customers whose number of activities is larger than 20 | select name, activity<br>from customer<br>where activity > 20<br>order by activity; | select name, activity<br>from customer_dimes<br>where activity > 20<br>order by activity; |

| Simple query No.6 | Database version | Data warehouse version |
|---|---|---|
| Query the product which is purchased most frequently | select product_code, count(sale_num) as purchasing_time<br>from sale_line<br>group by product_code<br>order by purchasing_time desc; | select product_code, count(time_id) as purchasing_time<br>from sale_fact<br>group by product_code<br>order by purchasing_time desc; |

Table 1: Brief descriptions of simple queries

| Complex query No.1 | Database version | Data warehouse version |
|---|---|---|
| Query these products whose actual price is larger than their list price | select customer_id, product_code, list_price, actual_price, actual_price - list_price as price_diff<br>from sale_line inner join product using (product_code)<br>inner join sale_head using (sale_num)<br>where list_price < actual_price<br>order by price_diff desc; | select * from (<br>select customer_id, product_code, list_price, total_money_sold/total_num_sold as actual_price,<br>(total_money_sold/total_num_sold) - list_price as price_diff<br>from sale_fact inner join product_dimen using (product_code)<br>) sub<br>where price_diff > 0<br>order by price_diff desc; |

| Complex query No.2 | Database version | Data warehouse version |
|---|---|---|
| Query the staff who has the largest money_sold of all the sales he/she deals with. | select staff_id, surname, firstnames, sum(quantity * actual_price) as moneysold_per_staff<br>from sale_line inner join sale_head using (sale_num)<br>inner join staff using (staff_id)<br>group by staff_id, surname, firstnames<br>order by moneysold_per_staff desc; | select staff_id, staff_surname, staff_lastname, sum(total_money_sold) as moneysold_per_staff<br>from sale_fact inner join staff_dimen using (staff_id)<br>group by staff_id, staff_surname, staff_lastname<br>order by moneysold_per_staff desc; |

| Complex query No.3 | Database version | Data warehouse version |
|---|---|---|
| Query the customer who has made the largest number of sales | select customer_id, name, count(sale_date) as frequency<br>from sale_head inner join customer using (customer_id)<br>group by customer_id, name<br>order by frequency desc; | select customer_id, name, count(distinct time_id) as frequency<br>from sale_fact inner join customer_dimes using (customer_id)<br>group by customer_id, name<br>order by frequency desc; |

| Complex query No.4 | Database version | Data warehouse version |
|---|---|---|
| Query the customer who has spent the largest amount of money | select customer_id, name, sum(quantity * actual_price) as totalmoney_per_customer<br>from sale_line inner join sale_head using (sale_num)<br>inner join customer using (customer_id)<br>group by customer_id, name<br>order by totalmoney_per_customer desc; | select customer_id, name, sum(total_money_sold) as totalmoney_per_customer<br>from sale_fact inner join customer_dimen using (customer_id)<br>group by customer_id, name<br>order by totalmoney_per_customer desc; |

Table 2: Brief descriptions of complex queries (continues over next page)

| Complex query No.5 | Database version | Data warehouse version |
|---|---|---|
| Query the total number of sales for every product in every month | select product_code, description, extract(month from sale_date) as month, sum(quantity) as month_sale_amount<br>from sale_head inner join sale_line using (sale_num)<br>inner join product using (product_code)<br>group by product_code, description, month; | select product_code, description, month, sum(total_num_sold) as month_sale_amount<br>from sale_fact inner join time_dimen using (time_id)<br>inner join product_dimen using (product_code)<br>group by product_code, description, month; |

| Complex query No.6 | Database version | Data warehouse version |
|---|---|---|
| Query the staff who has dealt with the largest number of sales | select staff_id, surname, firstnames, count(distinct sale_num) as service_amount<br>from sale_line inner join sale_head using (sale_num)<br>inner join staff using (staff_id)<br>group by staff_id, surname, firstnames<br>order by service_amount desc; | select staff_id, staff_surname, staff_lastname, count(distinct time_id) as service_amount<br>from sale_fact inner join staff_dimen using (staff_id)<br>group by staff_id, staff_surname, staff_lastname<br>order by service_amount desc; |

Table 2: Brief descriptions of complex queries

## 3.3 Experimental setup

After queries were set, which data should be gathered in the experiment to demonstrate and explain the performance of queries needed to be decided. There are four aspects that are discussed in our experiment.

### 3.3.1 Query execution plan

The query execution plan can demonstrate the optimized process in which the database system executes queries. The estimates of time and memory consumption for each step of the process is also shown in the execution plan, which can help us to understand the difference of performance between database version and data warehouse version. Execution plans can be simply obtained from database system by the command “explain” in PostgreSQL.

### 3.3.2 Single query response time

Query response time can be a reasonably simple and straightforward variable to show the performance of a query, and it is commonly the main concern of users of databases and data warehouses. To improve accuracy, we wrote a script (see appendix B) to run all queries 10 consecutive times to produce minimum and maximum response time as well as an average of 10 runs for each of them. These data were gathered by the pg_stat_statements extension, which is an internal PostgreSQL extension, to improve the accuracy of results. This extension provides a tool for tracking and storing execution statistics of all SQL statements executed by the database server until it is reset. The data gathered by the pg_stat_statements can be used to understand the performance of SQL statements. For example, minimum and maximum time as well as the standard deviation can present a general distribution of single query execution time and the average time can be used for comparing the performance of the two versions of one query.

### 3.3.3 Memory

Memory consumption is reasonably important in the experiment. On the one hand, it can be one variable to evaluate the performance of queries. A high memory consuming query is usually slightly bad. On the other hand, it can also be a significant factor to explain the difference of performance between database version and data warehouse version.

There are four types of memory in our experiment, including shared memory hit, shared memory read, temporary memory read and temporary memory written. Shared memory hit and shared memory read are two main sources of data when a database system executes queries. Shared memory hit means the amount of data derived from the cache while shared memory read means the amount of data derived from the disk. Reading from the cache should normally be faster than reading from disk. Therefore, if a query can read more data from the cache and less data from the disk, it would be faster.

Temporary memory read and temporary memory written usually appear in the execution of complex queries because a complex query could have much more complicated and high memory consuming processes, such as scanning, sorting and joining, which would produce a large amount of intermediate data that leads to the temporary memory usage. In general, it can be assumed that the more temporary memory is used, the slower the query would be.

The results of these four types of memory are also gathered with the same approach as single query response time (see appendix B).

### 3.3.4 Query throughput test

In the TPC-H specification, the query throughput test can be defined as a test which calculates the number of queries which the database can execute per hour. The object for this test is to simulate a real work environment for the database and measure its performance under a continuous workload (Ngamsuriyaroj and Pornpattana, 2010). This test is commonly used for estimating the query processing power when queries are submitted by a single stream, and the query throughput when queries are submitted by multiple concurrent users (TPC, 2017). The performance metric of this test is Query-per-Hour. In our experiment, we wrote another script (see appendix B) to randomly select queries from the list and execute them one by one. This process was run for 30 minutes and both normalized database and denormalized data warehouse had five individual runs to improve accuracy and reduce the effect of random factors.

### 3.3.5 Test environment

We conducted our tests on a macOS High Sierra Version 10.13.4 machine with the PostgreSQL 10.3 database system. Table 3 shows some basic information about the test machine and the disk system. In order to reduce external influences, we closed all unnecessary background processes, applications and other services on the test machine and kept the same condition during all the experiments.

| Operating system | macOS High Sierra Version 10.13.4 |
|---|---|
| CPU | Intel Core i5 (3.1 GHz) |
| Disk | Apple SSD AP0512J (512 GB) |
| Memory | 8 GB |
| Database | PostgreSQL 10.3 |

Table 3: Brief descriptions of test machine

# Chapter 4: Result analysis and discussion

## 4.1 Execution plan

The execution plans of all 24 queries, including 12 normalized database versions and 12 denormalized data warehouse versions, were gathered, and they were reasonably helpful for us to find the reason for difference performances between queries. This chapter will demonstrate two examples here, which are shown in Table 4 (simple query) and Table 5 (complex query):

| Simple query No.2 | Execution plan |
|---|---|
| Database version:<br>select extract(month<br>from sale_date) as<br>month, count(sale_num)<br>as month_sale_num<br>from sale_head<br>group by month; | GroupAggregate (cost=93820.48..107820.48 rows=700000 width=16)<br>Group Key: (date_part('month'::text, sale_date))<br>-> Sort (cost=93820.48..95570.48 rows=700000 width=16)<br>Sort Key: (date_part('month'::text, sale_date))<br>-> Seq Scan on sale_head (cost=0.00..13898.00 rows=700000 width=16) |
| Data warehouse version:<br>select month,<br>count(time_id) as<br>month_sale_num<br>from time_dimen<br>group by month; | Finalize GroupAggregate (cost=11882.07..11882.37 rows=12 width=13)<br>Group Key: month<br>-> Sort (cost=11882.07..11882.13 rows=24 width=13)<br>Sort Key: month<br>-> Gather (cost=11879.00..11881.52 rows=24 width=13)<br>Workers Planned: 2<br>-> Partial HashAggregate (cost=10879.00..10879.12 rows=12 width=13)<br>Group Key: month<br>-> Parallel Seq Scan on time_dimen (cost=0.00..9420.67 rows=291667 width=13) |

Table 4: Example of execution plan for simple query

It can be seen that query execution plans show all steps of execution process in which the database system executes queries. The estimates of cost and rows required of execution process is also shown behind each step. For the simple query, we found the execution plan for the query in the normalized database had fewer steps. However, every step in the database version needed to scan all 700000 rows (see the "rows=" entries in Table 4) while the data warehouse version scanned considerably fewer rows (at most 291667), which could make the data warehouse faster in this simple query (with total cost about one tenth that of the database query), and we will look at the actual performance in section 4.2.

Complex query No.6 (Database version and execution plan)

```
select staff_id, surname, firstnames, count(distinct sale_num) as service_amount
from sale_line inner join sale_head using (sale_num)
    inner join staff using (staff_id)
group by staff_id, surname, firstnames
order by service_amount desc;
```

```
Sort  (cost=1750207.38..1759273.29 rows=3626364 width=32)
  Sort Key: (count(DISTINCT sale_line.sale_num)) DESC
  ->  GroupAggregate  (cost=1079773.00..1181583.35 rows=3626364 width=32)
        Group Key: sale_head.staff_id, staff.surname, staff.firstnames
        ->  Sort  (cost=1079773.00..1092882.34 rows=5243737 width=32)
              Sort Key: sale_head.staff_id, staff.surname, staff.firstnames
              ->  Hash Join  (cost=34743.46..243593.22 rows=5243737 width=32)
                    Hash Cond: (sale_line.sale_num = sale_head.sale_num)
                    ->  Seq Scan on sale_line  (cost=0.00..90994.37 rows=5243737 width=8)
                    ->  Hash  (cost=21207.46..21207.46 rows=700000 width=32)
                          ->  Hash Join  (cost=223.00..21207.46 rows=700000 width=32)
                                Hash Cond: (sale_head.staff_id = staff.staff_id)
                                ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=12)
                                ->  Hash  (cost=148.00..148.00 rows=6000 width=24)
                                      ->  Seq Scan on staff  (cost=0.00..148.00 rows=6000 width=24)
```

Complex query No.6 (Data warehouse version and execution plan)

```
select staff_id, staff_surname, staff_lastname, count(distinct time_id) as service_amount
from sale_fact inner join staff_dimen using (staff_id)
group by staff_id, staff_surname, staff_lastname
order by service_amount desc;
```

```
Sort  (cost=1669128.26..1678194.17 rows=3626364 width=32)
  Sort Key: (count(DISTINCT sale_fact.time_id)) DESC
  ->  GroupAggregate  (cost=998694.83..1100504.23 rows=3626364 width=32)
        Group Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
        ->  Sort  (cost=998694.83..1011803.98 rows=5243661 width=32)
              Sort Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
              ->  Hash Join  (cost=196.00..162524.07 rows=5243661 width=32)
                    Hash Cond: (sale_fact.staff_id = staff_dimen.staff_id)
                    ->  Seq Scan on sale_fact  (cost=0.00..96134.61 rows=5243661 width=12)
                    ->  Hash  (cost=121.00..121.00 rows=6000 width=24)
                          ->  Seq Scan on staff_dimen  (cost=0.00..121.00 rows=6000 width=24)
```

Table 5: Example of execution plan for complex query

For the complex query, one extra join in the database version makes its execution plan more complicated compared with the data warehouse version. The denormalized data warehouse usually has fewer joins because it is denormalized, resulting in redundant data, pre-calculated data or aggregated data, and sometimes even merging of entire

tables. This could mean that it is faster for this complex query, and we will look at the actual performance in section 4.3 (see appendix C for all execution plans).

## 4.2 Simple query response time and memory

This section will show and discuss results of response time and memory usage for all simple queries. Summaries of these results are shown in Table 6 and Table 7, respectively.

| **Simple query** | | **Mean time (ms)** | **SD time (ms)** | **Minimum time (ms)** | **Maximum time (ms)** |
|---|---|---|---|---|---|
| 1 | DB | 766.04 | 382.23 | 625.54 | 1912.43 |
| | DW | 1037.84 | 672.80 | 810.63 | 3056.23 |
| 2 | DB | 367.62 | 46.37 | 346.06 | 505.87 |
| | DW | 150.35 | 83.23 | 121.87 | 400.02 |
| 3 | DB | 2.17 | 2.85 | 1.11 | 10.72 |
| | DW | 1.29 | 1.01 | 0.88 | 4.31 |
| 4 | DB | 1.28 | 0.35 | 1.11 | 2.33 |
| | DW | 1.23 | 0.57 | 0.96 | 2.94 |
| 5 | DB | 18.37 | 9.59 | 14.65 | 47.13 |
| | DW | 11.35 | 1.95 | 10.10 | 16.61 |
| 6 | DB | 609.54 | 7.80 | 601.19 | 628.53 |
| | DW | 625.90 | 2.50 | 622.72 | 631.34 |

Table 6: Results of response time for simple query

| **Simple query** | | **shared_mem_ hit (byte)** | **shared_mem_ read (byte)** | **the cache hit rate (=s_m_h/(s_m_h+s_m_r))** | temp_mem_read (byte) | temp_mem_written (byte) |
|---|---|---|---|---|---|---|
| 1 | DB | 41456 | 270024 | 13.3% | 0 | 0 |
| | DW | 7888 | 344592 | 2.2% | 0 | 0 |
| 2 | DB | 11232 | 29976 | 27.3% | 17848 | 17936 |
| | DW | 6280 | 48576 | 11.6% | 0 | 0 |
| 3 | DB | 644 | 61 | 91.3% | 0 | 0 |
| | DW | 439 | 49 | 90.0% | 0 | 0 |
| 4 | DB | 361 | 33 | 91.6% | 0 | 0 |
| | DW | 223 | 27 | 90.8% | 0 | 0 |
| 5 | DB | 8426 | 934 | 90% | 0 | 0 |
| | DW | 3610 | 398 | 97% | 0 | 0 |
| 6 | DB | 49136 | 262344 | 15.8% | 0 | 0 |
| | DW | 15520 | 336912 | 4.4% | 0 | 0 |

Table 7: Results of memory usage for simple query

There is no clear difference in performance of simple queries between the two versions. It can be seen from Table 6 that the results of comparing which version is

faster vary considerably, which is different from some other previous literature. In addition, the difference between their performance also becomes reasonably smaller compared with previous works (see Table 8 as an example). The ratio of response time between the database and the data warehouse for these three simple queries below is 1:2 to 2:3, while in our experiment the ratio varies and some are close to 1:1.

| Simple query | Query 1 | Query 2 | Query 3 |
|---|---|---|---|
| DB | 0.02 s | 21.2 s | 1646.98 s |
| DW | 0.031 s | 30.43 s | 2949.98 s |

Table 8: Results of simple query response time from previous literature (Zaker, Phon-Amnuaisuk and Haw, 2008)

In our experiment, we find that the shared memory hit can be a key factor. Therefore, we calculated the cache hit rate to help us understand and explain different query performance. It equals to shared memory hit divided by shared memory hit plus share memory read, in other words total shared memory usage here.

Based on the results, the six simple queries can be divided into three groups. Group one contains simple query No.1 and simple query No.6, and in this group the normalized database has better performance. Group two has simple query No.2 and simple query No.5, and the denormalized data warehouse is the winner in this group. Group three is simple query No.3 and simple query No.4 are in group three which do not have clear difference between their two versions.

- Group one: we have checked that there is no clear difference between the execution plans of database version and data warehouse version. To make it more precise, these two execution plans have nearly the same steps and each step has reasonably similar cost and derived rows. The main reason why they have different performance could be the different cache hit rate. It can be found that for these two queries, database versions have a higher rate (No.1: 13.3% vs. 2.2%; No.6: 15.8% vs. 4.4%). It means that although the execution plan demonstrates that database version and data warehouse version require a similar amount of data, the database version can make more effective use of the cache. Because of the high speed of cache, the database version is faster in these two queries.

- Group two: For simple query No.2, it has been selected as an example to show just how different the execution plan can be for a relatively simple query. The data warehouse version quickly eliminates unnecessary rows while the database version is stuck with scanning with all 700,000 rows. This situation cancels out the database version's advantage of higher cache hits (27.3% vs. 11.6%). The main reason for this might be that the star schema enables the database system to

execute the query purely from the time dimension, which is significantly smaller than Sale_Head. In addition, all the relevant date values have been pre-computed for the data warehouse version whereas for the database version these data need to be dynamically extracted. The database version also needs more temporary memory to carry out large sorts, and it is also the only simple query with temporary memory used. For simple query No.5, the data warehouse version has higher cache hit rate (90% vs. 97%), which might make it faster. Besides, the structures of execution plans for these two versions are reasonably similar. However, the data warehouse version has less cost for all steps.

- Group three: These two queries have effectively the same performance of their two versions within one standard deviation. For simple query No.3, it can be found from the execution plan that the data warehouse version again quickly eliminates unnecessary rows at the beginning, but this advantage is offset by considerably more rows than the database version (database 167 rows vs. data warehouse 291667 rows) when it scans the table. The database version also has only a slightly higher cache hit rate (91.3% vs. 90.0%). For simple query No.4, there is no clear difference between their execution plans and cache hit rate (91.6% vs. 90.8%). Therefore, there is no surprise that they have similar response times.

From the minimum and maximum response time, it can be seen that most of them have reasonably constant distribution of time. Simple query No.1 is the only exception here. The distance between the minimum and maximum is significant, and we found that the run with maximum response time usually appears at the beginning of ten runs. For this query, the sum process might be the most time and memory consuming process. Besides, this query also has reasonably small cache hit rate (13.3% and 2.2%), which means that it can only use small amount of data from the cache.

To sum up, the cache hit rate can be the key factor for simple queries and we could find that the database version usually has higher rate (win 5 times of 6), and this is the reason why the normalized database tends to have better performance for simple query. However, the structure of data warehouse sometimes helps it eliminate unnecessary rows during the execution and much fewer rows are required, which might offset its disadvantage of lower cache hit rate. Therefore, it is difficult to make a unified conclusion for this section. We may conclude that if the amount of data required is similar, the normalized database tends to be faster because it usually can use the cache more effectively. If the structure of data warehouse can help it reduce unnecessary rows for some particular queries, the winner can be different.

## 4.3 Complex query response time and memory

This section will show and discuss results of response time and memory usage for all complex queries. Summaries of these results are shown in Table 9 and Table 10.

| Complex query | | Mean time (ms) | SD time (ms) | Minimum time (ms) | Maximum time (ms) |
|---|---|---|---|---|---|
| 1 | DB | 3512.98 | 18.76 | 3484.76 | 3541.39 |
| | DW | 3729.66 | 12.27 | 3711.88 | 3756.75 |
| 2 | DB | 5143.53 | 89.87 | 5055.78 | 5372.69 |
| | DW | 3195.31 | 10.09 | 3182.17 | 3211.92 |
| 3 | DB | 792.04 | 25.37 | 771.89 | 866.49 |
| | DW | 5827.82 | 87.41 | 5751.57 | 6055.32 |
| 4 | DB | 4706.48 | 74.61 | 4643.11 | 4889.89 |
| | DW | 2936.70 | 33.85 | 2914.90 | 3035.78 |
| 5 | DB | 5795.66 | 70.22 | 5702.31 | 5916.92 |
| | DW | 7381.48 | 42.44 | 7311.05 | 7452.69 |
| 6 | DB | 8684.26 | 58.66 | 8629.23 | 8807.04 |
| | DW | 7307.01 | 42.68 | 7270.55 | 7397.56 |

Table 9: Results of response time for complex query

| Complex query | | shared_mem_ hit (byte) | shared_mem_ read (byte) | the cache hit rate (=s_m_h/(s_m_h+s_m_r)) | temp_mem_read (byte) | temp_mem_written (byte) |
|---|---|---|---|---|---|---|
| 1 | DB | 128337 | 308887 | 29.4% | 178508 | 178027 |
| | DW | 238320 | 329232 | 6.8% | 47718 | 47958 |
| 2 | DB | 164574 | 274713 | 37.5% | 760273 | 760607 |
| | DW | 32976 | 321552 | 9.3% | 401886 | 402938 |
| 3 | DB | 43232 | 7360 | 85.5% | 48368 | 48488 |
| | DW | 37520 | 316176 | 9.4% | 599472 | 600136 |
| 4 | DB | 221743 | 243425 | 47.7% | 729986 | 730281 |
| | DW | 54120 | 311312 | 14.8% | 412053 | 412716 |
| 5 | DB | 204395 | 233589 | 46.7% | 764444 | 764960 |
| | DW | 118252 | 392604 | 23.1% | 733384 | 733825 |
| 6 | DB | 121278 | 229170 | 34.6% | 778120 | 778952 |
| | DW | 51960 | 298240 | 15.3% | 606760 | 607832 |

Table 10: Results of memory usage for complex query

There is no clear difference in performance of complex queries between the two versions. It can be seen from Table 9 that the results of comparing which version is faster also vary considerably, which is different from some other previous literature. At the same time, the difference between their performance also becomes relatively smaller compared with previous works (see Table 11 as an example). We can see that for these following three complex queries the database versions need at least ten times

response time while in our experiment the distance between the database and the data warehouse is usually in several standard deviations.

| Complex query | Query 4 | Query 5 | Query 6 |
|---|---|---|---|
| DB | 830.13 s | 108000.34 s | 976.87 s |
| DW | 0.16 s | 10000.29 s | 102.32 s |

Table 11: Results of complex query response time from previous literature (Zaker, Phon-Amnuaisuk and Haw, 2008)

In our experiment, we find that the temporary memory used in simple queries is usually zero while in complex queries it can be another crucial factor. In general, the more temporary memory used, the slower the query is likely to be, due to the overhead of moving data back and forth.

Based on the results, the six complex queries can be divided into two groups. Group one contains complex query No.2, complex query No.4 and complex query No.6, and in this group the denormalized data warehouse is the winner. Group two has complex query No.1, complex query No.3 and complex query No.5, and the normalized database has better performance in this group.

- Group one: For these three queries (No.2,4,6), it can be seen from execution plans that the data warehouse version still always has fewer joins and uses less temporary memory. Although the normalized database always has higher cache hit rate (No.2 37.5% vs. 9.3%, No.4 47.7% vs. 14.8%, No.6 34.6% vs. 15.3%), this advantage seems to be cancelled out by much higher temporary memory usage. Therefore, the denormalized data warehouse has better performance in these three queries.

- Group two: For complex query No.1, the database version has significantly more complicated execution plan (see appendix C). For example, the normalized database has to join and scan the sale_head table with 700000 rows while the denormalized data warehouse does not, which considerably increase the amount of temporary memory that is required to process all steps in the normalized database execution plan. However, the database version has a reasonably higher cache hit rate (29.4% vs. 6.8%). This seems to offset the disadvantage due to the extra join of the database version. In general, the database version still works slightly faster. For complex query No.3, the data warehouse version uses much more temporary memory because in their execution plans it can be seen that the data warehouse needs to scan more than 5 million rows in the sale_fact table while the workload for the database version is about 700000 rows. The much higher temporary memory used for the data warehouse version is because the star

schema effectively merges sale_line with sale_head. Therefore, the fact table effectively includes data that are irrelevant to this query, however, all data has to be processed. Besides, the database version still has higher cache hit rate (85.5% vs. 9.4%) and the number of joins is the same in two versions. Therefore, the normalized database is faster in this query. For complex query No.5, the execution plans of both versions are reasonably similar and the number of joins is the same. In other words, the temporary memory required is close. Therefore, the higher cache hit rate (46.7% vs. 23.1%) of the database version might be the only reason to help the normalized database win in this query.

In conclusion, the temporary memory usage is another significant factor for complex queries, and the final result is usually decided by the number of joins and how complicated the execution plan is. We found that the data warehouse version usually has a simpler execution plan and fewer or at least the same number of joins as the database version, and this is the reason why the denormalized data warehouse generally has better performance for complex query. However, the normalized database always has a higher cache hit rate, which helps it sometimes offset its disadvantage of higher temporary memory. Therefore, it is also difficult to make a unified conclusion for complex queries. We may conclude that if queries on the denormalized data warehouse have fewer joins and much less temporary memory, they tend to be faster than the equivalent database query. If the number of joins and temporary memory required is similar or higher cache hit rate can help the database version offset its disadvantage, and the normalized database can be faster.

## 4.4 Distribution of query response time

In this section, we will show two queries with 100 runs as examples to demonstrate the time distribution of multiple runs. A summary of results of 100 runs is shown in Figure 3 below.

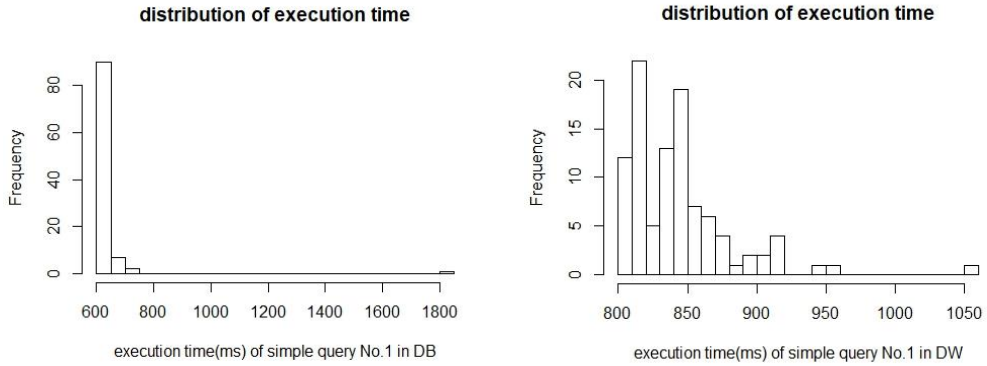
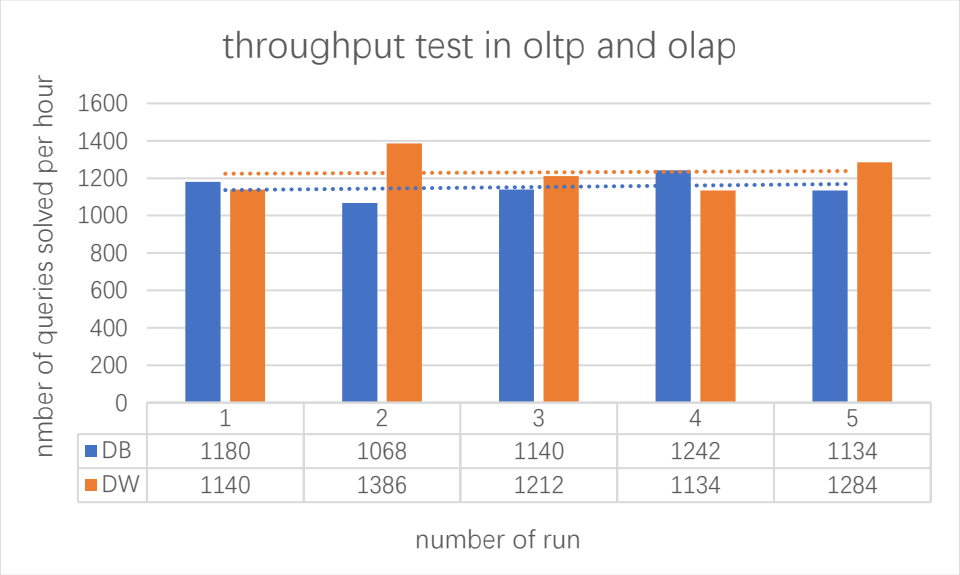



Figure 3: Examples of 100 runs for both versions for simple query No.1

It can be seen that the distributions here are more like right-skewed distributions rather than normal distributions. For the normalized database version, more than 90 runs have response time between 600 and 700 milliseconds. However, there are several runs that are considerably slow, which is consistent with our finding in section 4.2. There are several reasonably slow runs at the beginning and rest of them become faster and faster. The main reason might be that the more frequently a database system derives the same data for a same query, the more data will be stored and available in the cache. Then the following runs could be faster. Most of the values for the normalized database version are about 600 milliseconds, but the full range is very broad. For the denormalized data warehouse version, it can be seen that most of runs lie between 800 and 900 milliseconds. The distance between minimum and maximum is about 250 milliseconds, which is reasonably smaller compared to about 1200 milliseconds for the database version. In general, for simple query No.1, time distribution of the data warehouse version has a narrower range than the database version.

## 4.5 Query throughput test

In this section, we will show and discuss the performance of the query throughput test in the normalized database and the denormalized data warehouse. A summary of results of 5 individual runs and the average number is shown in Figure 4.




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DB | 1180 | 1068 | 1140 | 1242 | 1134 |
| DW | 1140 | 1386 | 1212 | 1134 | 1284 |

Figure 4: Results of query throughput test

It can be seen that there is no clear difference between the average workloads of the database and data warehouse. In addition, the variation of 5 runs in the data warehouse is slightly higher than the variation in the database. The reason might be some random factors, such as the query selection in each run. In general, it may be concluded that both normalized databases and denormalized data warehouses have similar performance when dealing with a single query stream.

# Chapter 5: Conclusion and further research

Since there is little recent literature about comparing the performance between the database and data warehouse and most of researches were finished with relatively old versions of database system in 1990s and 2000s, we want to investigate whether these previous findings are still true after about 20 years of further development of the database management system.

Our research questions are:

- Research question one: what are the differences of performance between the normalized database and denormalized data warehouse when running simple and complex queries?
- Research question two: has the difference between their performance changed compared to prior works?
- Research question three: is there any difference between normalization and denormalization with regards to query throughput?

In order to answer these research questions, we have presented an experimental examination comparing the performance of a normalized database and a denormalized data warehouse for simple and complex queries. In our experiment a normalized database and a denormalized data warehouse were built first and then 12 queries with two versions were set for comparison. Four types of data, including single query response time, memory usage, execution plan and query throughput test, were gathered to measure and understand the difference of performance between normalized and denormalized design. The main findings are:

- For simple queries, the normalized database usually has better performance due to the higher cache hit rate. However, the data warehouse can sometimes process much fewer rows from the table because of its denormalized design in which some important data have been pre-calculated and some unnecessary data or rows has been pre-deleted. This might help the data warehouse have better performance in some particular queries. This finding shows the difference between the database and data warehouse when running simple queries for our research question 1.
- For complex queries, the denormalized data warehouse usually has better performance because it can have fewer joins. However, complex queries on the normalized database still tend to have a higher cache hit rate, which sometimes helps offset the disadvantage of more joins. This finding discusses the difference

between the database and data warehouse when running complex queries for our research question 1.

- Compared to previous findings, the differences of performance between the normalized database and denormalized data warehouse becomes relatively smaller when running simple and complex queries, which answers our research question 2.

- The average workloads of both normalized and denormalized designs are reasonably similar, which answers our research question 3. Besides, the difference between the performance of the database version and the data warehouse version has shrunk compared with previous works, which answers our research question three: in terms of query throughput, the average performances of two versions are similar.

- The database version usually has a larger standard deviation of response time and a wider distribution of time value than the data warehouse version because the initial run of each query in the normalized database tends to be significantly slower than following runs. The main reason might be that the more effective use of the cache is the key advantage of the database, which helps it to catch up with or exceed the performance of the data warehouse. However, there are only a few data available in the cache for the first several runs. Therefore, the normalized database tends to work slowly at the beginning, which shows one of the differences between the database and data warehouse for our research question 1.

In general, the normalized database tends to be faster for simple queries because it can derive more data from the cache rather than the disk during the execution process. The denormalized data warehouse tends to be faster for complex queries due to its denormalized design. However, the results of comparing which version is faster vary considerably in our experiment with PostgreSQL. Besides, the differences between their performance become relatively smaller compared to prior researches. The average workloads of both normalization and denormalization are reasonably similar.

There are some related works that we did not do in our research. For example, different database systems and data sets could have influence on the performance of the normalized and denormalized queries. For example, if a data set contains a lot of hierarchies and data which needs to be aggregated, the denormalized data warehouse may have a more obvious advantage during execution. These aspects are worth investigating in future research.

# References


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

Cerpa, N. (1995). Pre-physical data base design heuristics. *Information & Management, 28*(6), 351-359.

Chohan, M.A., & Javed, M.Y. (2010). OLAP and OLTP data integration for operational level decision making. *Proceedings of IEEE the 2010 International Conference on Networking and Information Technology* (pp. 493-496). Manila, Philippines: Institute of Electrical and Electronics Engineers.

Codd, E.F. (1990). *The relational model for database management: version 2.* Reading, Mass.: Addison-Wesley.

Coleman, G. (1988). Normalizing not only way. *Computerworld, 22*(52), 63-64.

Darmont, J., Bentayed, F., & Boussaid, O. (2017). *Benchmarking data warehouses*. Retrieved from https://arxiv.org/ftp/arxiv/papers/1701/1701.00399.pdf.

Date, C.J. (1998). The birth of the relational model. *Intelligent Enterprise, 1*(4), 61.

Date, C.J. (2004). *An Introduction to Database Systems (eighth edition)*. Boston: Pearson/Addison Wesley.

Hahnke, J. (1996). Data model design for business analysis. *Unix Review, 14*(10), 35-40.

Inmon, W.H. (2005). *Building the data warehouse*. Hoboken: Wiley 2005.

Jepson, B. (2001). Postgre SQL vs My SQL: Building better databases. *Web Techniques, 6*(9), 32.

Lee, H. (1994). Justifying database normalization: A cost/benefit model. *Information Processing & Management, 31*(1), 59-67.

Lightstone, S., Teorey, T.J., & Nadeau, T. (2007). *Physical database design the database professional's guide to exploiting indexes, views, storage, and more*. Amsterdam; Boston: Morgan Kaufmann/Elsevier.

Mlodgenski, J. (2010). PostgreSQL vs MySQL How to Select the Right Open Source Database. *eWeek,* 21 Oct 2010. Retrieved from

http://go.galegroup.com.ezproxy.otago.ac.nz/ps/i.do?&id=GALE|A240077993&v=2.1& u=otago&it=r&p=AONE&sw=w&authCount=1.

Ngamsuriyaroj, S., & Pornpattana, R. (2010). Performance evaluation of TPC-H queries on MySQL cluster. *Proceedings of IEEE the 2010 24 th International Conference on Advanced Information Networking and Applications Workshops* (pp. 1035-1040). Perth, Australia: Institute of Electrical and Electronics Engineers.

Rahman, N. (2013). *Measuring performance for data warehouses-a balanced scorecard approach.* Retrieved from http://www.ijcit.org/ijcit_papers/vol4no1/IJCIT-130701.pdf.

Rajakumar, E., & Raja, R. (2015). An overview of data warehousing and OLAP Technology. *Advances in Natural and Applied Sciences, 9*(6), 288-296.

Rochkind, M. (2013). *Expert PHP and MySQL application design and development*. Berkeley, CA: Imprint: Apress.

Sanders, G.L., & Shin, S.K. (2001). Denormalization effects on performance of RDBMS. *Proceedings of IEEE the 34th Hawaii International Conference on System Sciences*. Maui, Hawaii: Institute of Electrical and Electronics Engineers.

Sen, A., & Jacob, V. (1998). Industrial-strength data warehousing. *Communications of the ACM, 41*(9), 28-31.

Shin, B. (2002). A case of data warehousing project management. *Information and Management, 39*(7), 581-592.

Shin, S.K., & Sanders, G.L. (2006). Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems, 42*(1), 267-282.

TPC. (1998). *History and overview of the TPC*. Retrieved from TPC.org: http://www.tpc.org/information/about/history.asp.

TPC. (2017). *TPC $BENCHMARK^{TM}$ H (Decision Support) Standard Specification Revision 2.17.3.* Retrieved from TPC.org: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.

TPC. (2018). *Active TPC Benchmarks*. Retrieved from TPC.org: http://www.tpc.org/information/benchmarks.asp.

Zaker, M., Phon-Amnuaisuk, S., & Haw, S.C. (2008). Optimizing the data warehouse design by hierarchical denormalizing. *Proceedings of the 8th WSEAS*

*International Conference on Applied Computer Science (pp. 131-138).* Venice, Italy: Institute of World Scientific and Engineering Academy and Society.

Zhang, Y., Zhou, X., Zhang, Y., Zhang, Y., Su, M., & Wang, S. (2016). Virtual denormalization via array index reference for main memory OLAP. *IEEE Transactions on Knowledge and Data Engineering, 28*(4), 1061-1074.

# Appendix A: Building the denormalized data warehouse

```
//fact

create table sale_fact
(product_code integer,
 customer_id integer,
 staff_id integer,
 time_id timestamp,
 total_num_sold numeric,
 total_money_sold numeric,

 primary key (time_id,customer_id,staff_id,product_code),

 foreign key (customer_id) references customer_dimen(customer_id),
 foreign key (time_id) references time_dimen(time_id),
 foreign key (staff_id) references staff_dimen(staff_id),
 foreign key (product_code) references product_dimen(product_code)
 );


//dimension

create table staff_dimen
(staff_id integer,
 staff_surname varchar,
 staff_lastname varchar,
 department varchar,
 position varchar,

 primary key (staff_id)
);

create table customer_dimen
(customer_id integer,
 name text,
 activity numeric,

 primary key (customer_id)
);

create table product_dimen
(product_code integer,
 description text,
```

```
 list_price numeric,

 primary key (product_code)
);

create table TIME_DIMEN
(time_id timestamp,
 year numeric,
 month numeric,
 day numeric,
 hour numeric,
 minute numeric,
 second numeric,

 primary key (time_id)
);


//insert data

insert into sale_fact
select product_code, customer_id, staff_id,
    sale_date as time_id,
    sum(quantity) as total_num_sold,
    sum(quantity * actual_price) as total_money_sold
from product inner join sale_line using (product_code)
    inner join sale_head using (sale_num)
    inner join customer using (customer_id)
    inner join staff using(staff_id)
group by product_code, customer_id, staff_id, sale_date
order by customer_id, sale_date, product_code;

insert into time_dimen
select distinct sale_date,
    extract(year from sale_date) as year,
    extract(month from sale_date) as month,
    extract(day from sale_date) as day,
    extract(hour from sale_date) as hour,
    extract(minute from sale_date) as minute,
    extract(second from sale_date) as second
from sale_head;

insert into product_dimen
select product_code, description, list_price
```

```
from product;

insert into customer_dimen
select customer_id, name, activity
from customer;

insert into staff_dimen
select staff_id, surname, firstnames, department, position
from staff;

drop table assembly, component, customer, order_head, order_line, product,
     sale_head, sale_line, staff, supplier;

vacuum full;
analyze verbose;
```

# Appendix B: Shell scripts to gather data of single query response time, memory usage and throughput test

(1) Script to obtain single query response time and memory usage

```
#!/bin/bash

# Number of simple or complex queries.
num_queries=6

# Number of times to run each query.
num_runs=10

# Each query has been stored in an appropriately named file.
# For example, "simple_oltp_1.sql" will contain the first simple query
# for the transaction database, while "complex_olap_4.sql" will contain
# the fourth complex query for the data warehouse.
#
# Results are written into pg_stat_statements

# oltp = transaction database, olap = data warehouse
# info490 is the name of test normalized database
# info4900 is the name of test denormalized data warehouse
for type in oltp olap
do
    if [ ${type} == "oltp" ]
    then
        dbname="info490"
    else
        dbname="info4900"
    fi
    psql --quiet --dbname=${dbname} --file=reset_stats.sql > /dev/null
    for complexity in simple complex
    do
        # Loop through the queries.
        for query in $(seq 1 ${num_queries})
        do
            echo
            echo "${complexity} ${type} query ${query} (${num_runs} runs)"
            echo
            cat ${complexity}_${type}_${query}.sql

            # Run each query the specified number of times.
            for run in $(seq 1 ${num_runs})
```

```
            do
               psql --quiet --dbname=${dbname} --
file=${complexity}_${type}_${query}.sql > /dev/null
            done
        done
    done
done
```

(2) reset_stats.sql in previous script

```
select pg_stat_statements_reset();
```

(3) Script to do throughput test for normalized database and denormalized data warehouse

```
# normalized database (script called 1.sh)

#!/bin/bash
count=0
while(( $num_queries<=12 ))
do
num_queries=$(( $RANDOM % 12 + 1));
psql --quiet --dbname=info490 --file=oltp_${num_queries}.sql > /dev/null ;
#let "num_queries++"
        count=`expr ${count} + 1`
        trap "echo total = ${count}; exit 1" SIGINT
done
# timeout 1800 ./1.sh



# denormalized data warehouse (script called 2.sh)

#!/bin/bash
count=0
while(( $num_queries<=12 ))
do
num_queries=$(( $RANDOM % 12 + 1));
psql --quiet --dbname=info4900 --file=olap_${num_queries}.sql > /dev/null;
#let "num_queries++"
        count=`expr ${count} + 1`
        trap "echo total = ${count}; exit 1" SIGINT
done
# timeout 1800 ./2.sh
```

# Appendix C: Query execution plans

## (1) Simple query No.1

```
select  product_code, sum(quantity) as total_num_sold_per_product
from sale_line
group by product_code
order by total_num_sold_per_product;
```

```
Sort  (cost=73822.79..73831.64 rows=3538 width=12)
  Sort Key: (sum(quantity))
  ->  Finalize GroupAggregate  (cost=73525.80..73614.25 rows=3538 width=12)
        Group Key: product_code
        ->  Sort  (cost=73525.80..73543.49 rows=7076 width=12)
              Sort Key: product_code
              ->  Gather  (cost=72330.35..73073.33 rows=7076 width=12)
                    Workers Planned: 2
                    ->  Partial HashAggregate  (cost=71330.35..71365.73 rows=3538 width=12)
                          Group Key: product_code
                          ->  Parallel Seq Scan on sale_line  (cost=0.00..60405.90 rows=2184890 width=8)
```

```
select product_code, sum(total_num_sold) as total_num_sold_per_product
from sale_fact
group by product_code
order by total_num_sold_per_product;
```

```
Sort  (cost=78998.70..79007.54 rows=3538 width=36)
  Sort Key: (sum(total_num_sold))
  ->  Finalize GroupAggregate  (cost=78675.17..78790.16 rows=3538 width=36)
        Group Key: product_code
        ->  Sort  (cost=78675.17..78692.86 rows=7076 width=36)
              Sort Key: product_code
              ->  Gather  (cost=77470.88..78222.71 rows=7076 width=36)
                    Workers Planned: 2
                    ->  Partial HashAggregate  (cost=76470.88..76515.11 rows=3538 width=36)
                          Group Key: product_code
                          ->  Parallel Seq Scan on sale_fact  (cost=0.00..65546.59 rows=2184859 width=9)
```

## (2) Simple query No.2

```
select extract(month from sale_date) as month, count(sale_num) as month_sale_num
from sale_head
group by month;
```

```
GroupAggregate  (cost=93820.48..107820.48 rows=700000 width=16)
  Group Key: (date_part('month'::text, sale_date))
  ->  Sort  (cost=93820.48..95570.48 rows=700000 width=16)
        Sort Key: (date_part('month'::text, sale_date))
        ->  Seq Scan on sale_head  (cost=0.00..13898.00 rows=700000 width=16)
```

```
select month, count(time_id) as month_sale_num
from time_dimen
group by month;
```

```
Finalize GroupAggregate  (cost=11882.07..11882.37 rows=12 width=13)
  Group Key: month
  ->  Sort  (cost=11882.07..11882.13 rows=24 width=13)
        Sort Key: month
        ->  Gather  (cost=11879.00..11881.52 rows=24 width=13)
              Workers Planned: 2
              ->  Partial HashAggregate  (cost=10879.00..10879.12 rows=12 width=13)
                    Group Key: month
                    ->  Parallel Seq Scan on time_dimen  (cost=0.00..9420.67 rows=291667 width=13)
```

## (3) Simple query No.3

```
select staff_id, department, position
from staff
where position like 'Manager';
```

```
Seq Scan on staff  (cost=0.00..163.00 rows=167 width=28)
  Filter: (("position")::text ~~ 'Manager'::text)
```

```
select staff_id, department, position
from staff_dimen
where position like 'Manager';
```

```
Finalize GroupAggregate  (cost=11882.07..11882.37 rows=12 width=13)
  Group Key: month
  ->  Sort  (cost=11882.07..11882.13 rows=24 width=13)
        Sort Key: month
        ->  Gather  (cost=11879.00..11881.52 rows=24 width=13)
              Workers Planned: 2
              ->  Partial HashAggregate  (cost=10879.00..10879.12 rows=12 width=13)
                    Group Key: month
                    ->  Parallel Seq Scan on time_dimen  (cost=0.00..9420.67 rows=291667 width=13)
```

## (4) Simple query No.4

```
select product_code, description, list_price

from product

where list_price between 50 and 100

order by list_price;


Sort  (cost=117.00..118.42 rows=567 width=32)

  Sort Key: list_price

  ->  Seq Scan on product  (cost=0.00..91.07 rows=567 width=32)

        Filter: ((list_price >= '50'::numeric) AND (list_price <= '100'::numeric))



select product_code, description, list_price

from product_dimen

where list_price between 50 and 100

order by list_price;


Sort  (cost=107.00..108.42 rows=567 width=32)

  Sort Key: list_price

  ->  Seq Scan on product_dimen  (cost=0.00..81.07 rows=567 width=32)

        Filter: ((list_price >= '50'::numeric) AND (list_price <= '100'::numeric))
```

## (5) Simple query No.5

```
select name, activity

from customer

where activity > 20

order by activity;


Sort  (cost=2107.22..2107.44 rows=85 width=17)

  Sort Key: activity

  ->  Seq Scan on customer  (cost=0.00..2104.50 rows=85 width=17)

        Filter: (activity > '20'::numeric)



select name, activity

from customer_dimen

where activity > 20

order by activity;


Sort  (cost=1438.26..1438.48 rows=86 width=17)

  Sort Key: activity

  ->  Seq Scan on customer_dimen  (cost=0.00..1435.50 rows=86 width=17)

        Filter: (activity > '20'::numeric)
```

## (6) Simple query No.6

```
select product_code, count(sale_num) as purchasing_time
from sale_line
group by product_code
order by purchasing_time desc;
```

```
Sort  (cost=73822.79..73831.64 rows=3538 width=12)
  Sort Key: (count(sale_num)) DESC
  ->  Finalize GroupAggregate  (cost=73525.80..73614.25 rows=3538 width=12)
        Group Key: product_code
        ->  Sort  (cost=73525.80..73543.49 rows=7076 width=12)
              Sort Key: product_code
              ->  Gather  (cost=72330.35..73073.33 rows=7076 width=12)
                    Workers Planned: 2
                    ->  Partial HashAggregate  (cost=71330.35..71365.73 rows=3538 width=12)
                          Group Key: product_code
                          ->  Parallel Seq Scan on sale_line  (cost=0.00..60405.90 rows=2184890 width=12)
```

```
select product_code, count(time_id) as purchasing_time
from sale_fact
group by product_code
order by purchasing_time desc;
```

```
Sort  (cost=78963.32..78972.16 rows=3538 width=12)
  Sort Key: (count(time_id)) DESC
  ->  Finalize GroupAggregate  (cost=78666.33..78754.78 rows=3538 width=12)
        Group Key: product_code
        ->  Sort  (cost=78666.33..78684.02 rows=7076 width=12)
              Sort Key: product_code
              ->  Gather  (cost=77470.88..78213.86 rows=7076 width=12)
                    Workers Planned: 2
                    ->  Partial HashAggregate  (cost=76470.88..76506.26 rows=3538 width=12)
                          Group Key: product_code
                          ->  Parallel Seq Scan on sale_fact  (cost=0.00..65546.59 rows=2184859 width=12)
```

## (7) Complex query No.1

```
select customer_id, product_code, list_price, actual_price, actual_price - list_price as price_diff
from sale_line inner join product using (product_code)
    inner join sale_head using (sale_num)
where list_price < actual_price
order by price_diff desc;
```

```
Gather Merge  (cost=237663.52..407611.35 rows=1456594 width=52)
  Workers Planned: 2
  ->  Sort  (cost=236663.50..238484.24 rows=728297 width=52)
        Sort Key: ((sale_line.actual_price - product.list_price)) DESC
        ->  Hash Join  (cost=24433.60..140853.11 rows=728297 width=52)
              Hash Cond: (sale_line.sale_num = sale_head.sale_num)
              ->  Hash Join  (cost=117.60..93567.63 rows=728297 width=24)
                    Hash Cond: (sale_line.product_code = product.product_code)
                    Join Filter: (product.list_price < sale_line.actual_price)
                    ->  Parallel Seq Scan on sale_line  (cost=0.00..60405.90 rows=2184890 width=18)
                    ->  Hash  (cost=73.38..73.38 rows=3538 width=10)
                          ->  Seq Scan on product  (cost=0.00..73.38 rows=3538 width=10)
              ->  Hash  (cost=12148.00..12148.00 rows=700000 width=12)
                    ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=12)
```

```
select * from (
      select customer_id, product_code, list_price, total_money_sold/total_num_sold as actual_price,
(total_money_sold/total_num_sold) - list_price as price_diff
      from sale_fact inner join product_dimen using (product_code)
) sub
where price_diff > 0
order by price_diff desc;
```

```
Gather Merge  (cost=251719.01..421664.28 rows=1456572 width=78)
  Workers Planned: 2
  ->  Sort  (cost=250718.99..252539.70 rows=728286 width=78)
        Sort Key: (((sale_fact.total_money_sold / sale_fact.total_num_sold) - product_dimen.list_price)) DESC
        ->  Hash Join  (cost=107.61..115083.25 rows=728286 width=78)
              Hash Cond: (sale_fact.product_code = product_dimen.product_code)
              Join Filter: (((sale_fact.total_money_sold / sale_fact.total_num_sold) - product_dimen.list_price) >
'0'::numeric)
              ->  Parallel Seq Scan on sale_fact  (cost=0.00..65546.59 rows=2184859 width=21)
              ->  Hash  (cost=63.38..63.38 rows=3538 width=10)
                    ->  Seq Scan on product_dimen  (cost=0.00..63.38 rows=3538 width=10)
```

## (8) Complex query No.2

```
select staff_id, surname, firstnames, sum(quantity * actual_price) as moneysold_per_staff
from sale_line inner join sale_head using (sale_num)
    inner join staff using (staff_id)
group by staff_id, surname, firstnames
order by moneysold_per_staff desc;
```

```
Sort  (cost=1819004.60..1828070.51 rows=3626364 width=56)
  Sort Key: (sum(((sale_line.quantity)::numeric * sale_line.actual_price))) DESC
  ->  Finalize GroupAggregate  (cost=506133.03..1176012.57 rows=3626364 width=56)
        Group Key: sale_head.staff_id, staff.surname, staff.firstnames
        ->  Gather Merge  (cost=506133.03..1076060.77 rows=4369780 width=56)
              Workers Planned: 2
              ->  Partial GroupAggregate  (cost=505133.01..570679.71 rows=2184890 width=56)
                    Group Key: sale_head.staff_id, staff.surname, staff.firstnames
                    ->  Sort  (cost=505133.01..510595.23 rows=2184890 width=34)
                          Sort Key: sale_head.staff_id, staff.surname, staff.firstnames
                          ->  Hash Join  (cost=34743.46..155583.60 rows=2184890 width=34)
                                Hash Cond: (sale_line.sale_num = sale_head.sale_num)
                                ->  Parallel Seq Scan on sale_line  (cost=0.00..60405.90 rows=2184890 width=18)
                                ->  Hash  (cost=21207.46..21207.46 rows=700000 width=32)
                                      ->  Hash Join  (cost=223.00..21207.46 rows=700000 width=32)
                                            Hash Cond: (sale_head.staff_id = staff.staff_id)
                                            ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=12)
                                            ->  Hash  (cost=148.00..148.00 rows=6000 width=24)
                                                  ->  Seq Scan on staff  (cost=0.00..148.00 rows=6000 width=24)
```

```
select staff_id, staff_surname, staff_lastname, sum(total_money_sold) as moneysold_per_staff
from sale_fact inner join staff_dimen using (staff_id)
group by staff_id, staff_surname, staff_lastname
order by moneysold_per_staff desc;
```

```
Sort  (cost=1730869.55..1739935.46 rows=3626364 width=56)
  Sort Key: (sum(sale_fact.total_money_sold)) DESC
  ->  Finalize GroupAggregate  (cost=428931.14..1087877.52 rows=3626364 width=56)
        Group Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
        ->  Gather Merge  (cost=428931.14..987926.50 rows=4369718 width=56)
              Workers Planned: 2
              ->  Partial GroupAggregate  (cost=427931.12..482552.59 rows=2184859 width=56)
                    Group Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
                    ->  Sort  (cost=427931.12..433393.27 rows=2184859 width=32)
                          Sort Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
                          ->  Hash Join  (cost=196.00..93323.20 rows=2184859 width=32)
```

```
                    Hash Cond: (sale_fact.staff_id = staff_dimen.staff_id)
                    ->  Parallel Seq Scan on sale_fact  (cost=0.00..65546.59 rows=2184859 width=12)
                    ->  Hash  (cost=121.00..121.00 rows=6000 width=24)
                          ->  Seq Scan on staff_dimen  (cost=0.00..121.00 rows=6000 width=24)
```

## (9) Complex query No.3

```
select customer_id, name, count(sale_date) as frequency
from sale_head inner join customer using (customer_id)
group by customer_id, name
order by frequency desc;
```

```
Sort  (cost=214976.89..216726.89 rows=700000 width=26)
  Sort Key: (count(sale_head.sale_date)) DESC
  ->  GroupAggregate  (cost=116266.40..130266.40 rows=700000 width=26)
        Group Key: sale_head.customer_id, customer.name
        ->  Sort  (cost=116266.40..118016.40 rows=700000 width=26)
              Sort Key: sale_head.customer_id, customer.name
              ->  Hash Join  (cost=3294.50..31555.92 rows=700000 width=26)
                    Hash Cond: (sale_head.customer_id = customer.customer_id)
                    ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=12)
                    ->  Hash  (cost=1917.00..1917.00 rows=75000 width=18)
                          ->  Seq Scan on customer  (cost=0.00..1917.00 rows=75000 width=18)
```

```
select customer_id, name, count(distinct time_id) as frequency
from sale_fact inner join customer_dimen using (customer_id)
group by customer_id, name
order by frequency desc;
```

```
Sort  (cost=1993823.44..2006932.59 rows=5243661 width=26)
  Sort Key: (count(DISTINCT sale_fact.time_id)) DESC
  ->  GroupAggregate  (cost=1052779.46..1157652.68 rows=5243661 width=26)
        Group Key: sale_fact.customer_id, customer_dimen.name
        ->  Sort  (cost=1052779.46..1065888.61 rows=5243661 width=26)
              Sort Key: sale_fact.customer_id, customer_dimen.name
              ->  Hash Join  (cost=2625.50..216608.71 rows=5243661 width=26)
                    Hash Cond: (sale_fact.customer_id = customer_dimen.customer_id)
                    ->  Seq Scan on sale_fact  (cost=0.00..96134.61 rows=5243661 width=12)
                    ->  Hash  (cost=1248.00..1248.00 rows=75000 width=18)
                          ->  Seq Scan on customer_dimen  (cost=0.00..1248.00 rows=75000 width=18)
```

## (10) Complex query No.4

```
select customer_id, name, sum(quantity * actual_price) as totalmoney_per_customer
from sale_line inner join sale_head using (sale_num)
    inner join customer using (customer_id)
group by customer_id, name
order by totalmoney_per_customer desc;
```

```
Sort  (cost=2118974.30..2132083.64 rows=5243737 width=50)
  Sort Key: (sum(((sale_line.quantity)::numeric * sale_line.actual_price))) DESC
  ->  Finalize GroupAggregate  (cost=501543.49..1175253.51 rows=5243737 width=50)
        Group Key: sale_head.customer_id, customer.name
        ->  Gather Merge  (cost=501543.49..1066009.00 rows=4369780 width=50)
              Workers Planned: 2
              ->  Partial GroupAggregate  (cost=500543.47..560627.94 rows=2184890 width=50)
                    Group Key: sale_head.customer_id, customer.name
                    ->  Sort  (cost=500543.47..506005.69 rows=2184890 width=28)
                          Sort Key: sale_head.customer_id, customer.name
                          ->  Hash Join  (cost=45091.92..165932.06 rows=2184890 width=28)
                                Hash Cond: (sale_line.sale_num = sale_head.sale_num)
                                ->  Parallel Seq Scan on sale_line  (cost=0.00..60405.90 rows=2184890 width=18)
                                ->  Hash  (cost=31555.92..31555.92 rows=700000 width=26)
                                      ->  Hash Join  (cost=3294.50..31555.92 rows=700000 width=26)
                                            Hash Cond: (sale_head.customer_id = customer.customer_id)
                                            ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=12)
                                            ->  Hash  (cost=1917.00..1917.00 rows=75000 width=18)
                                                  ->  Seq Scan on customer  (cost=0.00..1917.00 rows=75000 width=18)
```

```
select customer_id, name, sum(total_money_sold) as totalmoney_per_customer
from sale_fact inner join customer_dimen using (customer_id)
group by customer_id, name
order by totalmoney_per_customer desc;
```

```
Sort  (cost=2059622.52..2072731.67 rows=5243661 width=50)
  Sort Key: (sum(sale_fact.total_money_sold)) DESC
  ->  Finalize GroupAggregate  (cost=453141.62..1115917.77 rows=5243661 width=50)
        Group Key: sale_fact.customer_id, customer_dimen.name
        ->  Gather Merge  (cost=453141.62..1006674.82 rows=4369718 width=50)
              Workers Planned: 2
              ->  Partial GroupAggregate  (cost=452141.59..501300.92 rows=2184859 width=50)
                    Group Key: sale_fact.customer_id, customer_dimen.name
                    ->  Sort  (cost=452141.59..457603.74 rows=2184859 width=26)
                          Sort Key: sale_fact.customer_id, customer_dimen.name
                          ->  Hash Join  (cost=2625.50..117533.68 rows=2184859 width=26)
```

```
                        Hash Cond: (sale_fact.customer_id = customer_dimen.customer_id)
                        ->  Parallel Seq Scan on sale_fact  (cost=0.00..65546.59 rows=2184859 width=12)
                        ->  Hash  (cost=1248.00..1248.00 rows=75000 width=18)
                              ->  Seq Scan on customer_dimen  (cost=0.00..1248.00 rows=75000 width=18)
```

## (11)Complex query No.5

```
select product_code, description,
    extract(month from sale_date) as month, sum(quantity) as month_sale_amount
from sale_head inner join sale_line using (sale_num)
    inner join product using (product_code)
group by product_code, description, month;
```

```
Finalize GroupAggregate  (cost=520770.12..1189017.92 rows=5243737 width=42)
  Group Key: sale_line.product_code, product.description, (date_part('month'::text, sale_head.sale_date))
  ->  Gather Merge  (cost=520770.12..1079773.41 rows=4369780 width=42)
        Workers Planned: 2
        ->  Partial GroupAggregate  (cost=519770.10..574392.35 rows=2184890 width=42)
              Group Key: sale_line.product_code, product.description, (date_part('month'::text, sale_head.sale_date))
              ->  Sort  (cost=519770.10..525232.32 rows=2184890 width=38)
                    Sort Key: sale_line.product_code, product.description, (date_part('month'::text, sale_head.sale_date))
                    ->  Hash Join  (cost=24433.60..170220.70 rows=2184890 width=38)
                          Hash Cond: (sale_line.product_code = product.product_code)
                          ->  Hash Join  (cost=24316.00..137062.11 rows=2184890 width=16)
                                Hash Cond: (sale_line.sale_num = sale_head.sale_num)
                                ->  Parallel Seq Scan on sale_line  (cost=0.00..60405.90 rows=2184890 width=16)
                                ->  Hash  (cost=12148.00..12148.00 rows=700000 width=16)
                                      ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=16)
                          ->  Hash  (cost=73.38..73.38 rows=3538 width=26)
                                ->  Seq Scan on product  (cost=0.00..73.38 rows=3538 width=26)
```

```
select  product_code, description, month, sum(total_num_sold) as month_sale_amount
from sale_fact inner join time_dimen using (time_id)
    inner join product_dimen using (product_code)
group by product_code, description, month;
```

```
Finalize GroupAggregate  (cost=526056.31..1205218.90 rows=5243661 width=63)
  Group Key: sale_fact.product_code, product_dimen.description, time_dimen.month
  ->  Gather Merge  (cost=526056.31..1085051.67 rows=4369718 width=63)
        Workers Planned: 2
        ->  Partial GroupAggregate  (cost=525056.29..579677.76 rows=2184859 width=63)
              Group Key: sale_fact.product_code, product_dimen.description, time_dimen.month
              ->  Sort  (cost=525056.29..530518.43 rows=2184859 width=36)
```

```
Sort Key: sale_fact.product_code, product_dimen.description, time_dimen.month
->  Hash Join  (cost=25779.60..175510.37 rows=2184859 width=36)
      Hash Cond: (sale_fact.product_code = product_dimen.product_code)
      ->  Hash Join  (cost=25672.00..147824.41 rows=2184859 width=14)
            Hash Cond: (sale_fact.time_id = time_dimen.time_id)
            ->  Parallel Seq Scan on sale_fact  (cost=0.00..65546.59 rows=2184859 width=17)
            ->  Hash  (cost=13504.00..13504.00 rows=700000 width=13)
                  ->  Seq Scan on time_dimen  (cost=0.00..13504.00 rows=700000 width=13)
      ->  Hash  (cost=63.38..63.38 rows=3538 width=26)
            ->  Seq Scan on product_dimen  (cost=0.00..63.38 rows=3538 width=26)
```

## (12)Complex query No.6

```
select staff_id, surname, firstnames, count(distinct sale_num) as service_amount
from sale_line inner join sale_head using (sale_num)
    inner join staff using (staff_id)
group by staff_id, surname, firstnames
order by service_amount desc;
```

```
Sort  (cost=1750207.38..1759273.29 rows=3626364 width=32)
  Sort Key: (count(DISTINCT sale_line.sale_num)) DESC
  ->  GroupAggregate  (cost=1079773.00..1181583.35 rows=3626364 width=32)
        Group Key: sale_head.staff_id, staff.surname, staff.firstnames
        ->  Sort  (cost=1079773.00..1092882.34 rows=5243737 width=32)
              Sort Key: sale_head.staff_id, staff.surname, staff.firstnames
              ->  Hash Join  (cost=34743.46..243593.22 rows=5243737 width=32)
                    Hash Cond: (sale_line.sale_num = sale_head.sale_num)
                    ->  Seq Scan on sale_line  (cost=0.00..90994.37 rows=5243737 width=8)
                    ->  Hash  (cost=21207.46..21207.46 rows=700000 width=32)
                          ->  Hash Join  (cost=223.00..21207.46 rows=700000 width=32)
                                Hash Cond: (sale_head.staff_id = staff.staff_id)
                                ->  Seq Scan on sale_head  (cost=0.00..12148.00 rows=700000 width=12)
                                ->  Hash  (cost=148.00..148.00 rows=6000 width=24)
                                      ->  Seq Scan on staff  (cost=0.00..148.00 rows=6000 width=24)
```

```
select staff_id, staff_surname, staff_lastname, count(distinct time_id) as service_amount
from sale_fact inner join staff_dimen using (staff_id)
group by staff_id, staff_surname, staff_lastname
order by service_amount desc;
```

```
Sort  (cost=1669128.26..1678194.17 rows=3626364 width=32)
  Sort Key: (count(DISTINCT sale_fact.time_id)) DESC
  ->  GroupAggregate  (cost=998694.83..1100504.23 rows=3626364 width=32)
```

```
Group Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
->  Sort  (cost=998694.83..1011803.98 rows=5243661 width=32)
      Sort Key: sale_fact.staff_id, staff_dimen.staff_surname, staff_dimen.staff_lastname
      ->  Hash Join  (cost=196.00..162524.07 rows=5243661 width=32)
            Hash Cond: (sale_fact.staff_id = staff_dimen.staff_id)
            ->  Seq Scan on sale_fact  (cost=0.00..96134.61 rows=5243661 width=12)
            ->  Hash  (cost=121.00..121.00 rows=6000 width=24)
                  ->  Seq Scan on staff_dimen  (cost=0.00..121.00 rows=6000 width=24)
```