Database Performance Analysis and Tuning:

A Comparative Study of TPC-H Benchmark on Oracle and DB2

Jing Zhou

A Major Report
in
Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements For the Degree of Master of Computer Science

Concordia University
Montreal, Quebec, Canada

March 2003

0-612-77729-4

ABSTRACT

Database Performance Analysis and Tuning:

A Comparative Study of TPC-H Benchmark on Oracle and DB2

Jing Zhou

This project concentrates on the TPC-H benchmark on Oracle9i Enterprise Edition (EE) and DB2 Universal Database Version 7.2 Enterprise Edition (EE) on Windows2000 operating system. The TPC-H benchmark is a decision-support benchmark, consisting of a set of queries and refresh functions in order to simulate a real environment. There are several size factors supported by TPC to represent the database size. In this project, we use 1GB and 10GB database size. Furthermore, the test results are used to compare the performance of Oracle and DB2 on the Windows2000 operating system. Performance tuning is still a major issue in the database applications. There are two levels of tuning: system level and application level. We just focus on the application level tuning and study different factors and their effects on the DBMS' performance.

ACKNOWLEDGEMENTS

I have prepared this report under the supervision of Dr. Grahne and Dr. Shiri. I am truly indebted to both of them for their constant encouragement and valuable guidance without which I would not have been able to complete my research successfully. I am also very grateful to the system analysts of CS Department Concordia University for providing me with the perfect environment for my report.

Last but not the least, I think all my friends and family who supported me in this endeavor of mine.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I TPC-H Benchmark

1. Overview of TPC-H

The TPC Benchmark^{TM} H is a standard benchmark, provided by TPC, a non-profit organization that was founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions [1]. We can find TPC-H information on the website at the following url: www.tpc.org/tpch/default.asp.

The TPC-H consists of 2 refresh functions for concurrent data modifications and a set of 22 queries. The queries include a rich set of operators, and are far more complex than most Online Transaction Processing (OLTP), but can answer real-world questions.

Furthermore, there are some implementation guidelines to make it better for user and some measurement guidelines to represent the performance of the system more accurately [1].

1.1 Operation Model and Database Properties

The database is available 24 hours a day, 7 days a week for ad-hoc queries from multiple users and data modifications for all tables [1]. The queries and refresh functions can be executed at any time in order to simulate the worldwide business model.

In addition, the database should meet the Atomicity, Consistency, Isolation, and Durability properties [1]. TPC-H database must be implemented by a commercial available database management system and the queries are executed via an interface using dynamic SQL.

The size of database under test is scalable. The suggested size is 1GB, 10GB, 30GB, 100GB, 300GB, 1000GB, 3000GB, and 10000GB. The minimum size is 1GB.

The TPC-H database consists of eight tables. The relationships between columns of these tables are illustrated as follows:

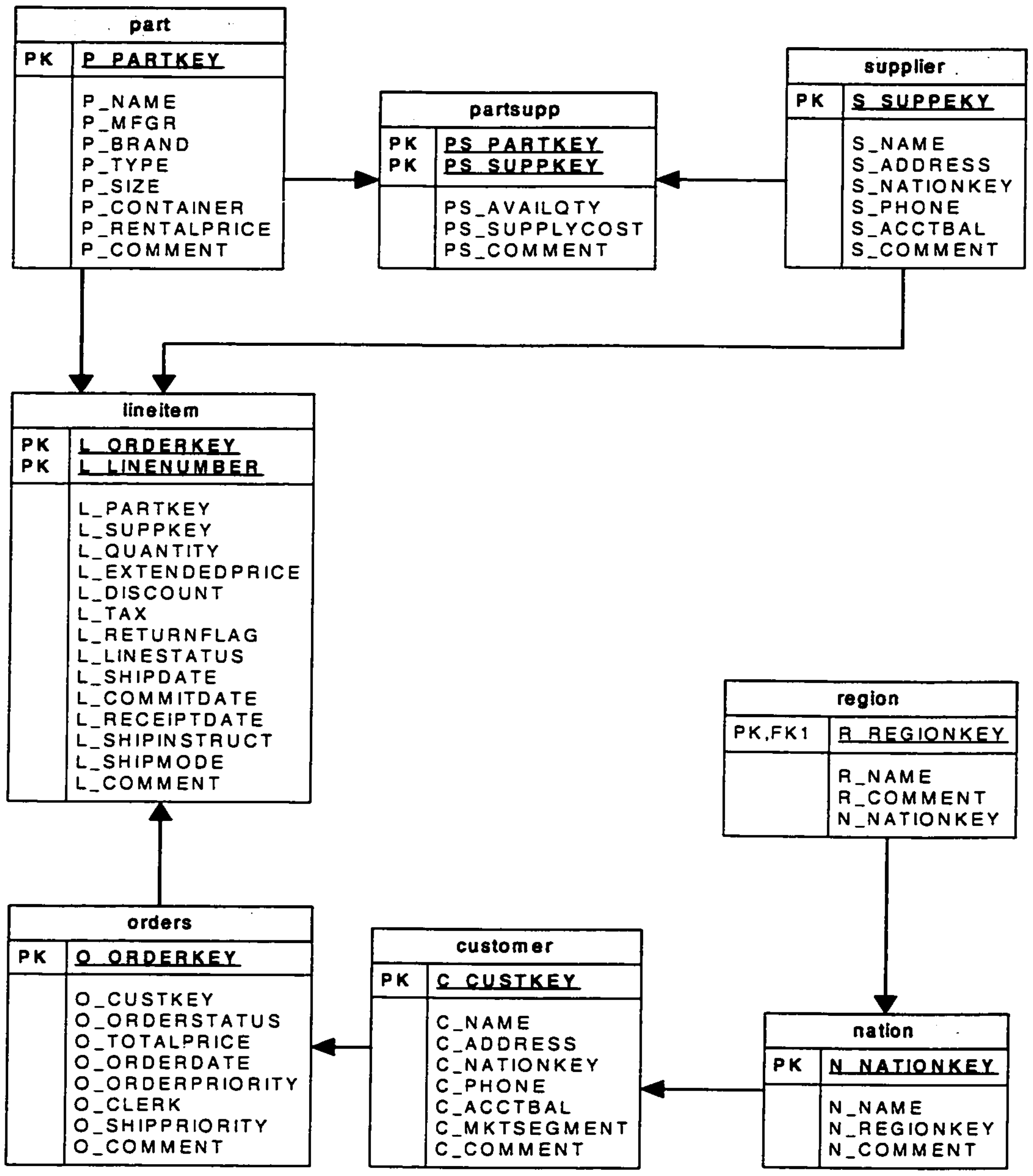


Figure 1.1 TPC-H Database Schema

1.2 Metrics

The TPC-H defines three primary metrics, which are used to measure the DBMS under test:

- The TPC-H Composite Query-per-Hour performance metric QphH@Size
- The price-performance metric is the TPC-H Price/Performance ($/QphH)

The performance metric reported by TPC-H is called TPC-H Composite Query-per-Hour Performance Metric (QphH@Size), which reflects multiple aspects of the capacities of the system to process the queries [1]. These aspects include (1) the selected database size against which the queries are executed; (2) the query processing power when queries are submitted by a single stream; (3) the query throughput when multiple concurrent users submit queries. The TPC-H Price/Performance is represented by the $/QphH@Size. Namely, the formulas of two metrics are listed respectively as follows:

$$\text{QphH@Size} = \sqrt[2]{(\text{Power@Size} * \text{Throughput@Size})}$$

$$\text{Price-per-QphH@Size} = (\$ \,/\, \text{QphH@size})$$

Where: "$", in the above formulas, represents the cost of the test bed, which consists of software and hardware components used in TPC-H test.

Furthermore, the formulae for calculating power metric and throughput metric are as follows.

$$\text{TPC-H Power@Size} = (3600 * SF) \,/\, \sqrt[24]{\left(\prod_{i=1}^{i=22} QI(i,0) * \prod_{j=1}^{j=2} RI(j,0)\right)}$$

Where:

$QI(i,0)$ is the timing interval, in seconds, of query Q_i within the single query stream of the power test

RI(j,0) is the timing interval, in seconds, of refresh function RF_j within the single query stream of the power test

Size is the database size chosen for the measurement and SF the corresponding scale factor

TPC-H Throughput@Size = $(S * 22 * 3600) * SF / T_s$

Where:

S is the number of query streams used in throughput test, and

T_s is the measurement interval defined as follows:

- It starts either when the first character of the executable query text of the first query of the first query stream is submitted to the SUT by the driver, or when the first character requesting the execution of the first refresh function is submitted to the SUT by the driver, whichever happens first.

- It ends either when the last character of output data from the last query of the last query stream is received by the driver from the SUT, or when the last transaction of the last refresh function has been completely and successfully committed at the SUT and a success message has been received by the driver from the SUT, whichever happens last.

The TPC-H performance test consists of two runs: Run1 and Run2. The reported performance metric must be for the run with the lower TPC-H Composite Query-Per-Hour Performance Metric because the TPC-H metrics reported for a given system must represent a conservative evaluation of the system's level of performance. Each of these includes a power test and a throughput test. The former measures the raw query execution power of the system when connected with a single active user, whereas the later measures the ability of the system to process most of queries in the least amount of time.

1.3 Queries and Refresh Functions

There are twenty-two different queries and two refresh functions, which are chosen in the TPC-H testing. The query templates are provided in Appendix C and the properties of those queries are listed in Appendix D. These queries have a high degree of complexity and differ from each other. Furthermore, each query contains one or more substitution parameters that describe how to generate the values needed to complete the query syntax. In what follows, we provide a description of each of these queries and refresh functions.

1.3.1 General Description of the Queries

- Query Overview:

  Each query is defined by the following components: 1) the business question, which illustrates the business context in which the query can be used; 2) the functional query definition, which defines, using SQL-92 language, the function to be performed by the query; 3) the substitution parameters, which describe how to generate the values needed to complete the query syntax; 4) the query validation, which describe how to validate the query against the qualification database.

In this section, we provide a brief description of the 22 queries defined by TPC-H benchmark, which we used in our experiments. Furthermore, Appendix C provides examples for the 22 queries and insert functions, used in Oracle tests.

Query1:
Provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60- 120 days of the greatest ship date contained in the database. The query lists total for extended price, discounted extended price, and discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by return flag and line status, and listed in ascending order of return flag and line status. A count of the number of lineitems in each group is also included.

Query2:

Finds, in a given region, for each part of a certain type and size, the supplier who can supply it at minimum cost. If several suppliers in that region offer the desired part type and size at the same (minimum) cost, the query lists the parts from suppliers with the 100 highest account balances. For each supplier, the query lists the supplier's account balance, name and nation; the part's number and manufacturer; the supplier's address, phone number and comment information.

Query3:

Retrieves the shipping priority and potential revenue of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more than 10 unshipped orders exist, only the 10 orders with largest revenue are listed.

Query4:

Counts the number of orders ordered in a given quarter of a given year in which the customer received at least one lineitem later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

Query5:

Lists for each nation in a region the revenue volume that resulted from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. The query is run in order to determine whether to institute local distribution centers in a given region or not. The query considers only parts ordered in a given year. The query displays the nations and revenue volume in descending order by revenue.

Query6:

Lists all the lineitems shipped in a given year with discounts between DISCOUNT-0.01 and DISCOUNT + 0.01. The query lists the amount by which the total revenue would have increased if these discounts had been eliminated for lineitems with l_quantity less than quantity.

Query7:

Finds, for two given nations, the gross discounted revenues derived from lineitems in which parts were shipped from a supplier in either nation to a customer in the other nation during 1995 and 1996.

Query8:

The market share for a given nation within a given region is defined as the fraction of the revenue from the products of a specified type in that region to that supplied by suppliers from the given nation.

Query9:

For each nation and each year, the profit for all parts ordered in that year contains a specified sub string in their names and were filled by a supplier in that nation. The query lists the nations in ascending alphabetical order and, for each nation, the year and profit in descending order by year (most recent first).

Query10:

Finds top 20 customers, in terms of their effect on lost revenue for a given quarter, who have returned parts. The query considers only parts that were ordered in the specified quarter. The query lists the customer's name, address, nation, and so on. The customers are listed in descending order of lost revenue.

Query11:

Finds, from scanning the available stock of suppliers in a given nation, all the parts that represent a significant percentage of the total value of all available parts. The query displays the part number and the value of those parts in descending order of value.

Query12:

Counts, by ship mode, for lineitems actually received by customers in a given year, the number of lineitems belonging to orders for which the receipt date exceeds the committed date for two different specified ship modes. Only lineitems that were actually shipped before the commit date are considered. The late lineitems are partitioned into two groups, those with priority URGENT or HIGH, and those with a priority other than URGENT or HIGH.

Query13:

Determines the distribution of customers by the number of orders they have made, including customers who have no record of orders, part or present. It counts and reports how many customers have orders, how many have 1,2,3,etc. A check is made to ensure that the orders counted do not fall into one of several special categories of orders. Special categories are identified in the order comment column by looking for a particular partner.

Query14:

Determines what percentage of the revenue in a given year and month was derived from promotional parts. The query considers only parts actually shipped in that month and the percentage is given.

Query15:

Finds the supplier who contributed the most to the overall revenue for parts shipped during a given quarter of a given year. In case of a tie, the query lists all suppliers whose contribution was equal to the maximum, presented in supplier number order.

Query16:

Counts the number of suppliers who can supply parts that satisfy a particular customer's requirements. The customer is interested in parts of eight different sizes as long as they are not of a given type, not of a given brand, and not from a supplier who has had complaints registered at the Better Business Bureau. Results must be presented in descending count and ascending brand, type, and size.

Query17:

Considers parts of a given brand and with a given container type and determines the average lineitem quantity of such parts ordered for all orders (past and pending) in the 7-year database. What would be the average yearly gross (undiscounted) loss in revenue if orders for these parts with a quantity of less than 20% of this average were no longer taken?

Query18:

Finds a list of the top 100 customers who have ever placed large quantity orders.

Query19:

Finds the gross discounted revenue for all orders for these different types of parts that were shipped by air or delivered in person. Parts are selected based on the combination of specific brands, a list of containers, and a range of sizes.

Query20:

Identifies suppliers who have an excess of a given part available; excess is defined to be more than 50% of the parts like the given part that the supplier shipped in a given year for a given nation. Only parts whose names share a certain naming convention are considered.

Query21:

Identifies suppliers, for a given nation, whose product was part of a multi-supplier order (with current status of 'F') where they were the only suppliers who failed to meet the committed delivery date.

Query22:

Counts how many customers within a specific range of country codes have not placed orders for 7 years but who have greater than average "positive" account balance. It also reflects the magnitude of that balance.

- Refresh Function Overview

The refresh functions are used to track the state of the OLTP database. Each refresh function consists of the components: 1) the business rational, which illustrates the business context in which the refresh function can be used; 2) the refresh function definition, which defines the pseudo-code the function to be performed by the refresh function; 3) the refresh data set, which defines the set of rows to be inserted or deleted by execution of the refresh function into or from the ORDERS and LINEITEM tables.

Refresh Function 1:

This refresh function inserts new sales information into the ORDERS and LINEITEMS tables, using the following scaling factor (SF) and data generation method used to populate the database.

```
LOOP (SF * 1500) TIMES
INSERT a new row into the ORDER table
LOOP RANDOME (1,7) TIMES
    INSERT a new row into the LINEITEM table
END LOOP
END LOOP
```

Refresh Function 2:

This refresh function removes old sales information from the ORDERS and LINEITEMS tables to emulate the removal of stale or obsolete information. The following scaling factor (SF) and data generation method is used to populate the database.

```
LOOP (SF * 1500) TIMES
DELETE FROM ORDER WHERE O_ORDERKEY = [value]
DELETE FROM LINETIME WHERE L_ORDERKEY = [value]
END LOOP
```

1.3.2 QGEN and DBGEN

TPC-H has given a set of prototypes of queries and refresh functions (see Appendix C) for details. According to the concrete syntax of the database system under test, those queries should be modified in order to meet the syntax requirement of the specific database. However, the modification will be limited to syntax matching only. Any other modifications that improve the system performance are not allowed.

DBGEN is a database population program (in ANSI 'C' for portability) used with the TPC-H. But the test sponsors must make some modifications to make it runable in the specific operating system environment. DBGEN will generate separate ASCII files, which contain pipe-delimited load data for one of the tables defined in the TPC-H, and data sets to be used in the refresh functions.

In addition, TPC-H provides another C program named QGEN, which also can be downloaded from the website of TPC. This program is used to generate executable queries. Like DBGEN, QGEN is controlled by a combination of command line options and environment variables. The option named -r seeds the random number generator with seed value <n>. The selection of n is done according to the following rules:

i) An initial seed (seed0) is first selected as the time stamp of the end of the database load time expressed in the format mmddhhmmss where mm is the month, dd the day, hh the hour, mm the minutes and ss the seconds. This seed is used to seed the power test of Run1.
ii) Further seeds (for the throughput test are chosen as seed0 + 1, seed0 + 2, ..., seed0 + n where n is the number of the throughput streams selected by the vendor.
iii) Sponsor decides whether Run2 should use the same seeds as the Run1, but the method of selecting seeds should be the same.

1.4 Execution Rules

The performance test follows the load test, which includes the statistics gathering activity. Any system activity that takes place between finishing of the test load and the beginning of the performance is limited and is not to improve the system's performance. For TPC-H requirements, each run includes one power test and one throughput test. Both of them should be done under the same test conditions.

1.4.1 Run Sequence

Run 1 follows the data load and Run2 follows Run1. If Run 1 is a failed run, the benchmark must be restarted with a new load test. If Run2 is a failed run, it may be restarted without a reload.

1.4.2 Power Test and Throughput Test

A power test measures the raw query execution power of the system when connected with a single user. It consists of three execution streams in order: refresh function1 stream, power test queries stream, and refresh function2 stream. The timing intervals for each query and for both refresh functions are collected and reported for the performance calculation.

A throughput test measures the ability of the system to process the most queries in the least amount of time. The throughput test must be driven by queries submitted by the driver through two or more sessions. The value of S, the minimum number of query streams for throughput test, is given in Table 1.1. In addition, another refresh stream should be parallel to those S query streams. In Table 1.1, SF is database scale factor, which represents the size of database. For example, 1 means the 1 GB size and so on.

SF (Scaling Factor)	S (Stream)
1	2
10	3
30	4
100	5
300	6
1000	7
3000	8
10000	9

Table 1.1 The Minimum Required Stream Count

The throughput test must follow, one and only one, power test. No activity that improves the system performance is allowed between the power test and the throughput test. The sequence of queries used in a power test and throughput test are shown in Appendix A.

1.4.3 Measurement Interval and Timing Interval

The measurement interval, $\mathbf{T_s}$, for the throughput test is measured in seconds as in section 1.2.

Each of the TCP-H queries and the refresh functions must be executed in an atomic fashion and timed in seconds. The timing interval, QI(i, s), for the execution of query Q_i within the stream s must be measured between:

- The time when the first character of the executable query text is submitted to the SUT by the driver
  AND
- The time when the first character of the next executable query text is submitted to the SUT by the driver, except for the last query of the set for which it is the time when the last character of the query's output data is received by the driver from the SUT.

1.5 The Drivers

A driver is a logical entity, representing the workload to the SUT, that can be implemented by one or more programs, processes, or systems and perform the function defined as above.

2. Implementation

2.1 Testing Plan

2.1.1 Available Database Engines

In this project, we used the following commercial databases:

- DB2 Universal Database Enterprise Edition 7.2
- Oracle9i Enterprise Edition

Our objective is to run TPC-H Benchmark on these DBMS in Windows2000 environment and compare their performance metrics.

For these experiments, we used an IBM Desktop 6849-32U, with a CPU of P4-1.7, 256 RAM and a 80GB hard disk of about $4000.00. In order to measure the cost of the software used in the test, we checked the price of the DBMS by checking the current market and asking the vendors. The price of Oracle 9i EE is about $63,348, whereas the price of DB2 UDB EE v7.2 is about $43,686. This information will be used in the performance metrics calculation in section 1.2.

2.1.2 Database Scaling

Scale factors, 1 and 10 are chosen from the set of fixed scale factors defined in section 1.1, resulting in two database sizes of 1GB and 10GB respectively. The minimum required size for a test database is 1GB.

Thus we will get four testing combinations on the DBMS and database size:

Size(GB) / DBMS	1	10
DB2	Test1	Test2
Oracle	Test3	Test4

Table 2.1 Tests in Our Experiment

2.1.3 Testing Purpose

We will compare the results of test1 and test3, and the results of test2 with those of test4. In the end, we want to determine which DBMS is suitable for a given data size and OS. Using that information, we can choose which DBMS on which OS would be more suitable for a particular business size.

2.2 Testing Preparation

In this phase, the schema of the database should be created according to the requirement of TPC-H. This database should then be populated by generated data. Also, query sequences should be prepared for the experiment.

2.2.1 Database Definition and Creation

The logical database of TPC-H has eight tables. What we should do is to create the corresponding database schema both for Oracle and DB2 respectively because of their different syntaxes.

2.2.2 Population Data Generation

The driver, DBGEN provided by TPC, is used to generate population data. However, the original standard C program did not work in Window2000 OS, some modifications were necessary.

In order to generate 1GB data for all tables of TPC-H database, the following command line is invoked in Windows2000 OS:

```
Dbgen.exe –s 1
```

Similarly, for getting 10GB data in Windows2000 OS, we use command line:

```
Dbgen.exe –s 10
```

Where:
s denotes the scaling factor.

2.2.3 Query Generation and Validation

- Query Sequence

  The 22 queries needed in our experiment can be generated by QGEN, provided by the TPC-H. We had to modify this program so that it could run in the operating system Windows2000.

As discussed In Section 1.3, we know that a seed is used to generate the random number for the parameters substituted in the template queries. Here, we choose the seed0 as the end time of the loading 1GB data into DBMS Oracle9i first time.

The required minimum number of query streams for throughput is 2 for 1GB size and 3 for 10 GB. For 1GB data, stream1 and stream2 are used for run1 and run2, whereas for 10 GB data, stream1, stream2, and stream3 are used for run1 and run2. In our project, run1 and run2 use different seeds. All seeds used to generate query sequences are listed in Table 2.2.

The first database was created on August 7, 2002. In addition, and the end of loading data time is 14:03:30PM, so the seed for stream0 of run1 is 807140330. Others seeds are defined according to the above rules.

Seed / Run / Stream	Run1	Run2
Stream0(power)	807140330	807210122
Stream1(throughput)	807140331	807210123
Stream2(throughput)	807140332	807210124
Stream3(throughput)	807140333	807210125

Table 2.2 Seeds for Streams

- Refresh Functions Generation

A refresh function is a sequence of just Insertion functions (RF1) or Deletion function (RF2).

The driver, DBGEN, can be used to generate raw data for RF1 and RF2 using some specific parameters. Furthermore, the parameter –S n should be given to specify the scale factor of a database, for which those refresh functions data, is used. For example, for the 1GB database, we use the following command line:

```
Dbgen.exe –S 1 –U6
```

Similarly, for the 10GB database, we use the command line:

```
Dbgen.exe –S 10 –U8
```

In both cases above, the parameter –Un is used to create a specified number (n) of data sets in flat files for the refresh function 1 and the refresh function2. The flat generated by the above command lines are just the raw data records, whose fields are separated by the pipe-delimiter '|'. In order to get the refresh functions in SQL, we developed two new drivers in C, to generate the sequence of RF1 and RF2 automatically.

The functional query definition uses the following minor modification in each DBMS, respectively:

- Oracle

  1. For date fields, we use the Oracle date function. For example, to_date(date '1998-03-21'), which converts the given string format date into '21-Mar-98', which is internal representation in Oracle.
  2. The standard Oracle date syntax is used for the date arithmetic. For example, to_date(date '1996-02-21' + interval '5' days)
  3. Queries 2, 3, 10, 13, and 21 should be modified in order to fetch the given number of the query result. The ***rownum < n*** is used in the WHERE clause of those queries.

  Where: ***n*** is an integer, representing the number of rows that are returned from the query.

- DB2

1. The standard IBM date syntax is used for the date arithmetic. For example, date ('1996-02-21')+5 days means the date '26-Jan-96' in DB2.
2. Queries 2, 3, 10, 13, and 21 should be modified in order to fetch the given number of tuples in the query results. The ***fetch last n rows*** is used in the Where clause of those queries.

Where: ***n*** is an integer, representing the number of rows that are returned from the query.

2.3 Testing

2.3.1 Load Test

After the database was created, the load test starts. By using "load" function or command, we can load the population data into a DBMS. Statistics collection activity follows the loading activity. The collected times are listed in the Table 2.3 and Table 2.4, for 1GB and 10 GB database respectively.

Database scale factor =1

Time (minutes)	Oracle	DB2
Loading time	3:57	3:32
Statistics collection time	33:20	6:20
Total	37:17	9:53

Table 2.3 Execution Time of Load Test (1GB)

Database scale factor =10

Time (minutes)	Oracle	DB2
Loading time	38:28	50:45
Statistics collection time	416:16	95:34
Total	454:44	146:19

Table 2.4 Execution Time of Load Test (10GB)

From the above Table 2.3 and Table 2.4, we can see that the statistics collection time for DB2 is just about 20% of the statistics collection time for Oracle. However, it means that Oracle will collect more information about the database.

2.3.2 Performance Test

The power test is executed first. Stream 0 contains a pair of refresh function and 22 queries in a specific sequence. The execution time of these functions and queries are recorded for the purpose of calculating TPC-H power metric.

Following the power test, we conducted the throughput test. A set of query streams is executed concurrently. This is to simulate simultaneous access of the database by several users. For 1GB size database, the number of streams in throughput test is 2, whereas for 10GB size database, this number of streams in throughput test is 3. To ensure that the streams run currently, several command windows are opened, one for each stream. An update stream is run in throughput test. The execution time of each query is recorded.

2.3.3 Test Results and Comparison

In tables of this section, T_s represents the execution time of the refresh functions.

2.3.3.1 Oracle Performance Metric

The following diagram shows the execution time of each query for 1GB database.

TPC-H Timing Intervals (in seconds):

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Stream 0								
Run 1	69.1	9.1	14.0	47.0	102.1	34.0	95.0	59.1
Run 2	61.0	10.1	11.0	47.0	89.1	31.1	90.1	44.1
Stream 1								
Run 1	569.1	101.0	185.1	848.0	892.0	766.1	971.0	748.0
Run 2	437.1	80.0	156.0	462.1	766.0	612.1	1055.1	589.1
Stream 2								
Run 1	294.0	96.1	185.0	358.0	928.0	756.1	967.1	791.1
Run 2	619.0	93.1	241.1	795.1	922.0	532.1	938.0	651.1

Query	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
Stream 0								
Run 1	276.1	52.1	9.1	44.0	201.0	37.0	76.1	15.1
Run 2	296.0	49.0	9.1	41.0	200.1	40.0	66.1	16.0
Stream 1								
Run 1	1418.1	961.1	190.0	1184.0	854.1	505.1	1151.0	68.0
Run 2	1518.0	838.1	135.0	1121.1	831.1	659.1	976.0	80.0
Stream 2								
Run 1	1369.0	893.1	196.0	860.0	766.1	662.1	1321.0	79.1
Run 2	1513.0	829.1	202.0	791.0	810.1	660.0	933.0	77.0

Query	Q17	Q18	Q19	Q20	Q21	Q22	RF1	RF2
Stream 0								
Run 1	69.0	60.1	44.1	43.0	173.1	15.0	23.1	10.1
Run 2	66.0	48.0	43.1	37.1	147.0	14.0	21.1	9.0
Stream 1								
Run 1	834.1	658.0	584.1	665.0	1895.1	231.0	28.0	15.1
Run 2	1154.1	643.0	686.1	952.0	1163.0	209.0	25.1	13.0
Stream 2								
Run 1	944.1	654.0	670.0	554.0	1844.1	89.1	25.1	12.1
Run 2	1171.1	610.1	700.0	789.0	1185.1	213.0	24.1	12.0

Table 2.5 Execution Times of the Oracle Performance Tests (1GB)

The first row, in the above Table 2.5, shows the sequence of queries. The second row shows the stream number of power test and the following two rows belong to this stream. The third row shows the time taken for Run 1 on the first 8 queries. For example, execution time of Query1 took 69.1 seconds in power test of Run 1, while Query2 took 9.1 seconds. The fourth row shows the time taken for Run2 on the first 8 queries. The fifth row shows the first stream number of throughput test and the following two rows belong to this stream. The eighth row shows the second stream number of the throughput test and the following two rows belong to the second stream. Similarly, the meaning of the rest of rows is obvious. In the rest of this part, the tables have similar meanings.

Based on the time values collected in the above table, we can calculate the metrics, defined in section 1.2. Table 2.6 summarizes this important information. For instance, the queries per hour of power test of Run1 are 83.1, while the queries per hour of throughput test of Run1 are 9.7. As a result, the queries per hour of Run1 are 28.4. In addition, the execution time of the refresh functions in throughput test of Run1 is 17856.1 seconds.

Run ID	QppH@1GB	QthH@1GB	QphH@1GB	Ts
Run 1	83.1	9.7	28.4	17856.1
Run 2	87.4	10.4	30.2	16770.0

Table 2.6 Comparison of Runs in Oracle (1GB)

Since the QphH@1GB in Run 1 is lower than that in Run 2, the result of Run 1 is adopted. Finally, we divided the cost of test bed (63,348 + 4000) by adopted QphH@1GB (28.4), and we can get the Price/Performance Metric.

QphH@1GB =	**28.4**
TPC-H Price/Performance Metric	**2371**

The following diagram shows the execution time of each query on 10GB database.

TPC-H Timing Intervals (in seconds):

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Stream 0								
Run 1	729.0	182.0	951.1	1188.1	1103.0	454.0	1113.0	565.0
Run 2	759.1	257.0	889.1	1188.0	1083.1	452.0	1051.0	604.0
Stream 1								
Run 1	10588.1	2351.1	15083.1	15083.0	19939.1	10014.0	17783.1	16984.1
Run 2	11910.0	2402.0	16293.1	15777.0	14447.1	10549.1	15157.0	14189.1
Stream 2								
Run 1	11353.0	2870.1	16238.0	21236.0	8086.0	16748.1	16698.1	24237.0
Run 2	9599.1	2423.1	17338.1	18357.0	14252.0	12286.0	19134.1	14382.0
Stream 3								
Run 1	10284.0	2381.1	13434.1	16225.0	17790.1	12116.0	0.0	8545.0
Run 2	12313.1	2711.1	5577.1	15870.0	14350.0	10593.1	14717.0	12662.1

Query	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
Stream 0								
Run 1	3236.0	1059.1	92.1	851.1	2713.1	498.1	1001.0	402.1
Run 2	2923.1	1112.0	90.0	844.1	2465.1	502.1	999.0	552.1
Stream 1								
Run 1	26433.1	14888.0	3823.1	22532.1	14857.1	11815.0	25124.0	2940.0
Run 2	26397.0	18253.0	2869.0	23138.1	13783.0	13412.0	24881.0	3286.1
Stream 2								
Run 1	24237.0	16335.1	2472.1	21762.0	41452.1	13101.1	21246.0	2686.00
Run 2	26397.0	18253.0	2869.0	23138.1	13783.0	13412.0	24881.0	3286.1
Stream 3								
Run 1	26405.0	17046.0	4203.0	19646.0	15114.1	10882.0	25805.0	3053.0
Run 2	28793.1	15513.0	3573.1	21343.0	16332.0	12506.1	24492.0	3024.1

Query	Q17	Q18	Q19	Q20	Q21	Q22	RF1	RF2
Stream 0								
Run 1	965.0	565.0	579.0	910.0	2101.1	264.1	203.0	98.1

Run 2	958.0	550.1	575.1	989.1	2096.1	257.0	200.1	98.0
Stream 1								
Run 1	21463.1	12701.1	11816.0	16576.1	8162.1	3962.0	249.0	204.1
Run 2	20494.0	12096.0	14513.1	16483.1	38866.1	4021.1	315.1	200.1
Stream 2								
Run 1	23175.0	11902.1	13340.1	12816.0	8401.1	3921.0	323.0	212.0
Run 2	24541.1	11543.0	10428.0	16007.0	38335.0	5052.1	308.1	211.0
Stream 3								
Run 1	25414.0	12306.0	11272.0	13183.1	40712.1	4048.0	321.0	225.1
Run 2	20347.1	24594.0	13761.0	17645.1	39338.0	4028.0	333.0	203.1

Table 2.7 Execution Time of Oracle Performance Test (10GB)

Run ID	QppH@10GB	QthH@10GB	QphH@10GB	Ts
Run 1	56.3	6.9	19.7	364576.8
Run 2	55.3	7.1	19.8	355577.0

Table 2.8 Comparison of Runs in Oracle (10GB)

Since the value QphH@10GB in Run 1 is lower than Run 2, the result of Run1 is adopted:

QphH@10GB =	**19.7**
TPC-H Price/Performance Metric	**3419**

Observations: Although, the size of the database whose scaling factor is 10, is 10 times that of the database whose scaling factor is 1, the queries per hour of the 10GB database is just 69.4% to that of the 1GB database.

2.3.3.2 DB2 Performance Metric

The following diagram shows the execution time of each query for 1GB database.

TPC-H Timing Intervals (in seconds):

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Stream 0								
Run 1	65.9	11.0	128.2	81.0	106.5	31.0	299.1	147.5
Run 2	66.9	11.2	127.6	79.4	116.4	25.5	255.3	143.6
Stream 1								
Run 1	11927.9	102.7	6133.7	819.5	242.4	1323.7	710.7	324.3
Run 2	159.2	34.0	352.1	177.8	5524.5	216.8	529.5	9413.0
Stream 2								
Run 1	98.8	49.4	133.1	503.5	267.7	925.7	506.4	1084.2
Run 2	65.2	83.8	278.6	189.2	117.3	143.3	7614.7	252.8

Query	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
Stream 0								
Run 1	1638.3	105.2	118.2	45.4	129.6	37.1	28.3	22.2
Run 2	1619.7	94.3	123.6	57.0	153.8	41.0	28.3	12.0
Stream 1								
Run 1	1802.2	588.8	260.5	225.8	293.1	152.4	322.6	72.8
Run 2	1703.6	650.6	297.3	468.2	235.4	109.5	98.3	38.9
Stream 2								
Run 1	2146.7	136.6	285.3	9111.2	243.2	1763.0	58.2	38.9
Run 2	2044.1	1679.3	372.0	6669.9	244.9	107.9	178.1	72.6

Query	Q17	Q18	Q19	Q20	Q21	Q22	RF1	RF2
Stream 0								
Run 1	6396.7	60.8	47.3	10241.8	581.4	109.9	51. 3	125.0
Run 2	5714.0	64.2	54.6	8465.6	580.7	118.9	49.5	126.2
Stream 1								
Run 1	6484.6	124.1	90.1	8356.2	473.6	264.9	236.1	362.3
Run 2	6311.9	135.7	173.9	13765.7	782.0	299.4	216.6	286.1
Stream 2								
Run 1	12002.0	143.5	115.7	7514.6	1099.4	360.6	265.3	321.9
Run 2	6918.7	403.6	118.5	9308.5	1020.2	455.3	242.5	299.5

Table 2.9 Execution Time of DB2 Performance Test (1GB)

Run ID	QppH@1GB	QthH@1GB	QphH@1GB	Ts
Run 1	28.7	3.8	10.4	62085.8
Run 2	29.3	4.1	11.0	56714.4

Table 2.10 Comparison of Runs in DB2 (1GB)

Since the QphH@1GB in run 1 is lower than that in run 2, so the result of run 1 is adopted:

QphH@1GB = **10.4**

TPC-H Price/Performance Metric ($) **4585**

The following diagram shows the execution time of each query for 10GB database.

TPC-H Timing Intervals (in seconds):

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Stream 0								
Run 1	849.3	355.4	2113.5	1345.4	1705.3	579.5	2779.1	1866.2
Run 2	850.5	358.1	2129.3	1348.6	1671.2	713.5	2837.3	1991.5
Stream 1								
Run 1	3581.2	2020.1	6248.4	2208.5	21465.3	2341.3	17079.2	8389.1
Run 2	6082.2	1848.5	5683.4	8997.1	21504.5	3180.5	10371.1	11024.1
Stream 2								
Run 1	1880.4	1385.6	12029.4	7370.42	5557.2	7174.0	25244.2	15264.4
Run 2	7005.5	1398.2	4091.6	40992.5	7891.2	7397.4	6963.4	15164.5
Stream3								
Run 1	2329.2	1874.5	3982.24	26777.1	10873.2	2680.0	17297.2	13068.1
Run 2	3231.5	1651.6	5238.3	27401.2	10325.3	2521.2	10703.3	13513.0

Query	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
Stream 0								
Run 1	14544.3	1618.5	1057.4	916.1	946.5	723.3	872.0	233.57

Run 2	19795.1	1642.5	1077.1	925.5	1214.0	735.3	874.4	239.4
Stream 1								
Run 1	43430.4	24796.6	2605.2	2584.3	3037.1	4663.4	7287.5	2096.1
Run 2	28425.3	27474.5	2602.3	2559.6	8011.0	7685.4	5442.1	1741.5
Stream 2								
Run 1	23948.6	7815.0	3743.1	5537.5	2658.3	4590.2	4348.1	1470.1
Run 2	24615.5	7367.0	1981.4	24251.5	3330.2	4714.4	4489.3	1486.3
Stream 3								
Run 1	22853.5	9373.6	1962.6	27910.1	8465.0	5063.2	4351.4	1536.2
Run 2	27247.5	12310.6	2103.3	18869.1	8445.2	4503.2	3598.1	1196.3

Query	Q17	Q18	Q19	Q20	Q21	Q22	RF1	RF2
Stream 0								
Run 1	*	1334.5	855.2	*	6151.1	1392.4	760.5	96.0
Run 2	*	1258.5	824.1	*	7396.2	1399.5	691.2	98.6
Stream 1								
Run 1	*	6358.2	11529.0	*	16951.6	2440.4	191852.1	98.19
Run 2	*	6900.1	4155.3	*	17453.1	3162.3	189284.2	68.5
Stream 2								
Run 1	*	3244.2	3196.2	*	31492.3	6895.5	549.1	97.1
Run 2	*	4889.6	3059.5	*	10477.1	8124.1	502.5	74.4
Stream 3								
Run 1	*	1371.2	11668.3	*	9217.1	7051.1	382.1	100.5
Run 2	*	1367.0	4969.2	*	14015.5	4020.0	537.4	69.2

* [1]

Table 2.11 Execution Time of DB2 Performance Test (10GB)

Run ID	QppH@10GB	QthH@10GB	QphH@10GB	Ts
Run 1	20.0	2.2	6.6	1074236.1
Run 2	19.3	2.2	6.5	1064874.4

Table 2.12 Comparison of Runs in DB2 (10GB)

Since the value of QphH@1GB in Run 2 is lower than that in Run 1, we adopted Run 1:

QphH@10GB =	**6.5**
TPC-H Price/Performance Metric	**7336**

The value of QphH@1GB on Oracle is 28.4 queries per hour, whereas the value of QphH@1GB on DB2 is 10.4 queries per hour. As a result, the performance of Oracle9i is better than that of DB2 Universal Database on Windows2000 OS and IBM 6849-32U. Figure 2.1 illustrates the power test difference between Oracle and DB2 with 1GB size factor and we can see that most of queries need more execution time on DB2.

[1] * Represents the execution time that is more than 96 hours. Here it represents 96 hours. This value also is used to calculate the QphH@10G for DB2.

2.3.3.3 Comparison

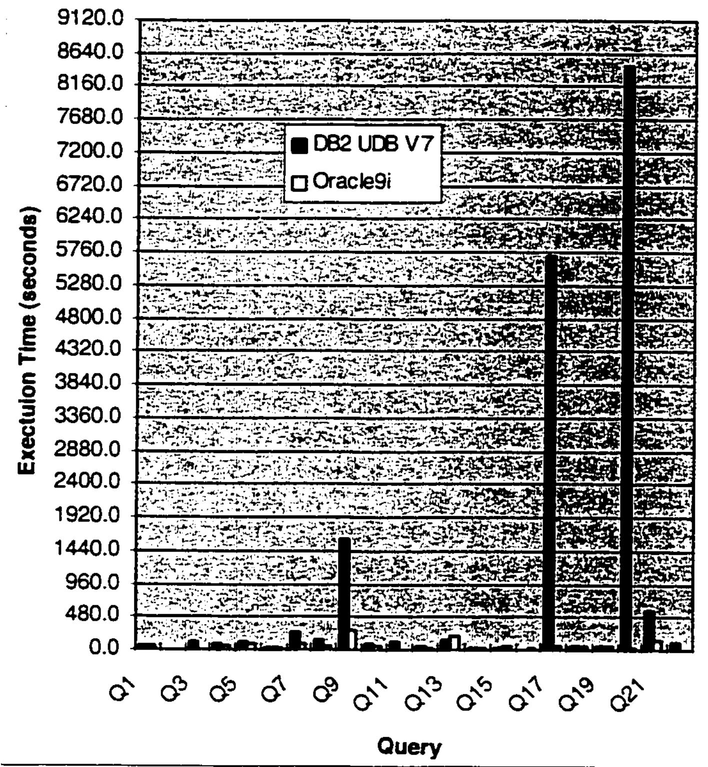


Figure 2.1 Power Test Comparison of DB2 and Oracle in 1GB database

In Figure 2.1, we can see Oracle need far less time for most queries in the TPC-H. Especially for query 20, the DB2 needs 8465.6 seconds, whereas Oracle just needs 37.1 seconds.

The value of QphH@10GB on Oracle is 19.7 queries-per-hour, whereas the value of QphH@10GB on DB2 is 6.5 queries-per-hour. As a result, the performance of Oracle is better than that of DB2 on Windows2000 OS and IBM 6849-32U too. Figure 2.2 illustrates the power test difference between Oracle and DB2 with 10GB size factor and we can see that most of queries need more execution time on DB2.

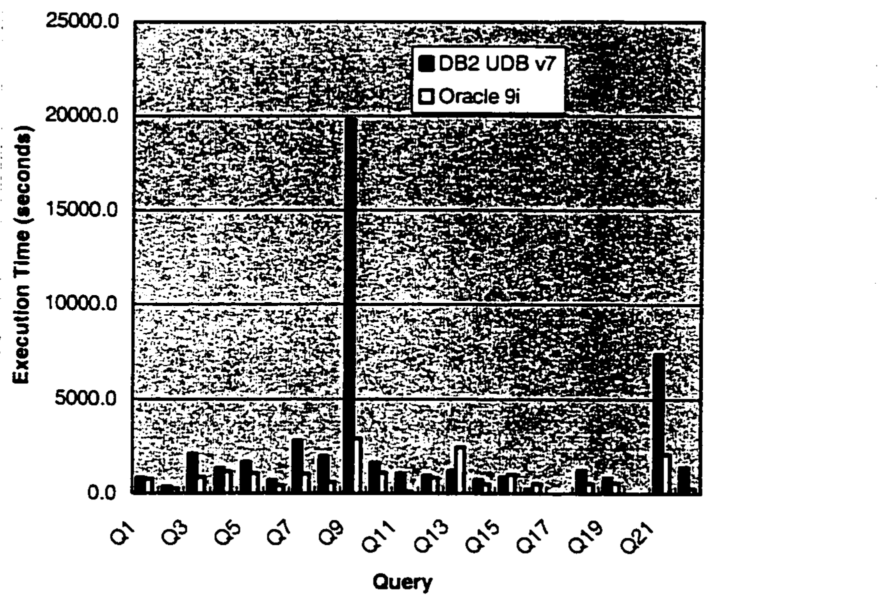


Figure 2.2 Power Test Comparison of DB2 and Oracle in 10GB Database

Note: In DB2 10GB test, the executions of Query17 and Query20 last more than 96 hours respectively without output result even when the temporary table space, which is used to store the intermediate result of query, is adjusted to 18GB. When we monitored the DB2's status, using Performance Monitor Tool, the system 'appeared' normal. Furthermore, we tried to contact IBM Company to solve this problem, but no response.

Considering the statistics collection time in the load test, we can see that Oracle needs far more time than that needed by DB2. It seems that the Oracle gathers more information during the statistics collection phase; however, it pays when answering queries.

Part II Performance Tuning

Performance tuning is a vital part of the management and the administration of successful database systems.

Database tuning is both an easy and a difficult task. It is easy, because common sense can be applied without use of theorems, and difficult, because it requires a deep understanding of the principles and knowledge of the application domain [5].

An optimizer may use indices to access table more efficiently. Indexing impacts how an optimizer chooses an access path to the table. In addition, the different kinds of indices have different effects on the access plan. In Chapter 3, we will study how an optimizer chooses an access plan, based on factors such as available statistics, database size and operator methods. In Chapter 4, we will study how an optimizer uses indices, considering the cost of query. In Chapter 5, query rewriting will be reviewed. In Chapter 6, some tools provided by specific DBMS will be reviewed. Those tools are helpful in doing database system performance tuning.

3. Execution Plan

Understanding how the optimizer works is the basis for solving performance problems. After SQL statements are submitted to the DBMS, the query processor must follow three steps: parsing, creating a logical query plan, and converting the logical query plan into a physical query plan [4].

In this section, we first discuss the importance of statistics in a DBMS. Then we study the theoretical basis of the physical query plan and its implementation in a commercial DBMS. Furthermore, some details, such as join methods, in the real commercial query plan will be analyzed.

3.1 Statistical Information

Statistics include various information about the database, such as the number of rows in each table, the average length of rows in each table, data distribution of a specific field, selectivity, and other useful information. Using these statistics, the optimizer can determine an "optimal" execution plan. Without such information, the database manager could make a decision that may adversely affect the performance of an SQL statement.

3.1.1 Oracle Statistics Information

In Oracle9i DBMS, there are almost 100 tables in the DBMS dictionary to store information about the user's database. An optimizer will use this information to create a lowest-cost plan if the statistics information exists; otherwise, the DBMS will use the default method with little or no optimization. Oracle generates statistics using estimation based on random data sampling and exact computation.

The following test illustrates the important role of statistics. A simple SQL statement is used for the testing:

```
select count(*) from lineitem
```

- Before collecting statistics

The statistics related information is checked and the result is also listed out in Figure 3.1.

Here, we see that there is no detailed information, such as the number of rows, in each table. Similarly, other system tables can be checked. We can now observe, how the Oracle DBMS chooses the execution plan without statistics.

```
SQL> select table_name, num_rows from user_tables;

TABLE_NAME                       NUM_ROWS
------------------------------ ----------
CUSTOMER
LINEITEM
NATION
ORDERS
PART
PARTSUPP
REGION
```

Figure 3.1 Results From Oracle System Table user_tables Before Collecting Statistics

```
SQL> select count(*) from LINEITEM;

  COUNT(*)
----------
   6001215

Elapsed: 00:00:32.03

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'LINEITEM'
```

Figure 3.2 Execution Result and Plan of Statement without Statistics (Oracle)

In Figure 3.2, we can see that the table lineitem was accessed by a full-table scan method and the execution time of this statement was 32.03 seconds.

- After collecting statistics

The following command was issued in Oracle9i to collect the statistics information and the result information was listed too.

```
   SQL> execute dbms_stats.gather_schema_stats('tpch');

PL/SQL procedure successfully completed.
```

At this time, the same SQL statement was issued as before and we get the following result.

```
SQL> select table_name, num_rows from user_tables;

TABLE_NAME                       NUM_ROWS
------------------------------ ----------
CUSTOMER                           150000
LINEITEM                          6001215
NATION                                 25
ORDERS                            1500000
PART                               200000
PARTSUPP                           800000
REGION                                  5
SUPPLIER                            10000
```

Figure 3.3 Selection Result From System Table user_tables after Collecting Statistics (Oracle)

All the information about each table can be found in the statistics table, including the number of rows in each table, the index information, the length of rows, the number of blocks for each table, and other information.

The following command was issued as before

```
SQL> select count(*) from LINEITEM;
  COUNT(*)
----------
   6001215

Elapsed: 00:00:05.03

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4908 Card=1)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (FAST FULL SCAN) OF 'PK_LINEITEM' (UNIQUE) (Cost=4
          908 Card=6001215)
```

Figure 3.4 Execution Result and Plan of Statement After Statistics (Oracle)

In Figure 3.4, we can see that Oracle DBMS chooses an index on primary key to count the number of rows in the table lineitem. Because the pages in index on primary key are fewer than the pages in the table lineitem data, fewer numbers of blocks are accessed and that causes a lower I/O cost. The execution time is short too, just 5.03 seconds.

Two TPC-H power tests were taken in Oracle. One has statistics and the other has no statistics. The following diagram shows the difference in the results.

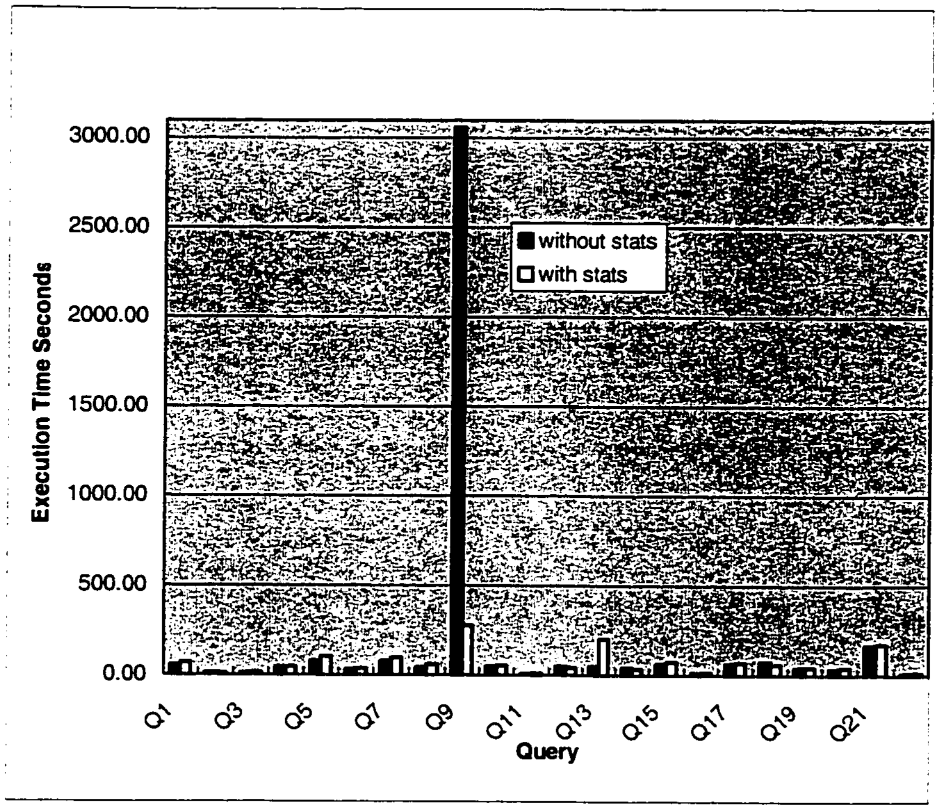


Figure 3.5 Throughput Comparisons of Power Test with Statistics and Power Test Without Statistics

As the result, the total execution time of the power test with statistics is 1257.13 seconds, whereas the execution time is 3380.71 seconds without statistics. The first one is 1.7 times faster than the latter one. In Figure 3.5, we can see that it takes less time for most queries to run when statistics information exists.

3.1.2 DB2 Statistics

In this subsection, we will analyze the importance of statistics in DB2. The same procedure performed for Oracle was done on the DB2 database.

- Before collecting statistics

```
select count(*) from LINEITEM

1
--------------
     6001215


Number of rows retrieved is:        1
Number of rows sent to output is:   1

Elapsed Time is:            62.309     seconds
```

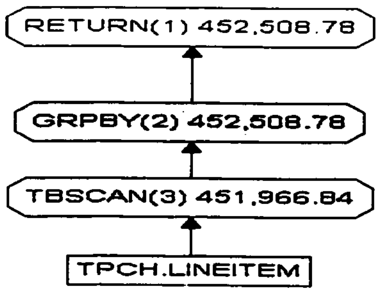


Figure 3.6 Execution Result and Plan of Statement without Statistics (DB2)

In Figure 3.6, the full-table access method is used by DB2 when no statistics information is collected.

- After collecting statistics

```
select count(*) from LINEITEM

1
--------------
     6001215


Number of rows retrieved is:        1
Number of rows sent to output is:    1

Elapsed Time is:            5.098       seconds
```

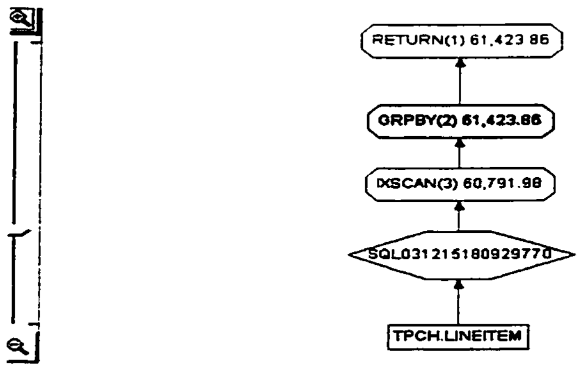


Figure 3.7 Execution Result and Plan of Statement with Statistics (DB2)

One of the available indices on primary keys is utilized by DB2, when statistics information is collected. Furthermore, the cost 61,423.86 shown in Figure 3.7, is lower than the cost 452,507.78 shown in Figure 3.6, because fewer pages are accessed when using index information.

3.1.3 Conclusions

Statistical information is very important for the DBMS to estimate the cost of each candidate execution plan when this information is available. Otherwise, the default method (full-table scan) will be used.

3.2 Cost-based and Rule-based Approaches

Normally, there are two techniques for an optimizer to formulate execution plans: a cost-based approach and a rule-based approach. The goal of the cost-based approach is the best throughput, or a minimal resource use, necessary to process all rows accessed by the statement, whereas the goal of the rule-based approach is the best response time, or a minimal resource use, necessary to process the first row accessed by the SQL statement.

For Oracle 9i, when statistics are available, the optimizer will choose a cost-based approach even if the statistics are partial on the tables. The cost-based approach generally chooses an execution plan that is as good as, or better than the plans chosen by the rule-based approach, especially for queries with multiple joins or multiple indices [8]. Choosing which approach to follow can be done by setting a system parameter in the initial file or by adding hints in the query statement. Cost-based approach improves the query processing productivity by eliminating the need for manually tuning the SQL statements because one does not need to specify the order of joins in the WHERE clause [8].

In Oracle, the optimization approach can be set on the session level and the application level [2].

For DB2, the cost is derived from a combination of CPU cost (by the number of instructions) and I/O (by the numbers of seeks and page transfers) [10].

The rule-based approach is an alternative to the cost-based approach for the optimizer and is available for backward compatibility. This approach will cause the optimizer to choose the execution plan without considering statistics [8]. Queries in TPC-H benchmark are very complex and statistics information is required to be collected, so the cost-based approach is more appropriate for the experimenting.

3.3 Theoretical Execution Plan and Real Execution Plan

In theory, we can define an optimized physical execution plan based on the well-known public rules for a given SQL statement. Of course, there are some rules that are adopted by an individual DBSM provider.

Several SQL statements are examined in the following. We are interested in finding out the differences between the execution plans determined by Oracle and DB2, and comparing them with optimized execution plans that we found out by using well-known rules [4].

In the following section, we will illustrate some queries in TPC-H. There is no index except on the primary key of each table. We classify these queries as regular and nested. The latter may be further classified as Non-correlated subquery and correlated subquery.

3.3.1 Regular Statements

First, let us review Query3, a simple SQL statement in the TPC-H. This query is shown in Figure 3.8.

```
       SQL> select
      l_orderkey,
      sum(l_extendedprice * (1 - l_discount)) as revenue,
      o_orderdate,
      o_shippriority
from
      customer,
      orders,
      lineitem
where
      c_mktsegment = 'MACHINERY'
      and c_custkey = o_custkey
      and l_orderkey = o_orderkey
      and o_orderdate < to_date (date '1995-03-26')
      and l_shipdate > to_date (date '1995-03-26')
      and rownum < 11
group by
      l_orderkey,
      o_orderdate,
      o_shippriority
```

```
order by
        revenue desc,
        o_orderdate;
```

Figure 3.8 Query3 of TPC-H

- Execution Tree Plan Created Manually

According to the theorems in [4], we have the following rules to create the tree plan.

1) Smaller tables will be joined first, because the intermediate result would be small.
2) Pushing the duplicate elimination operator δ as down as possible in the tree

As a result, we can get the following query plan.

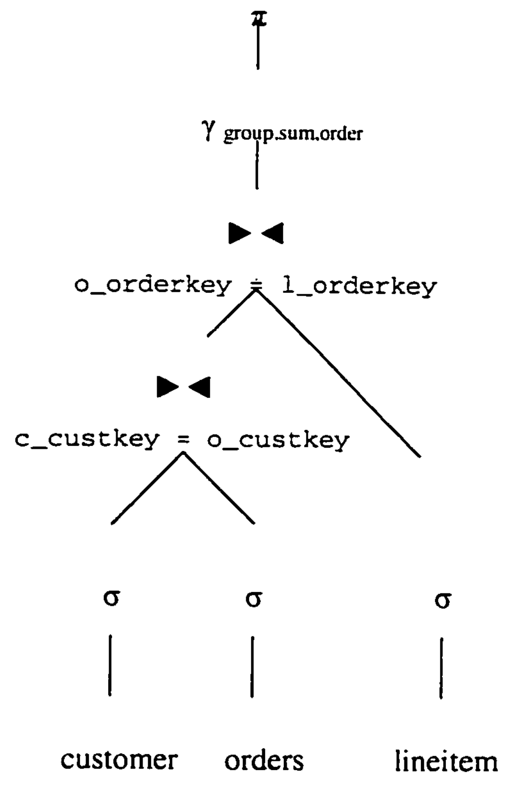


Figure 3.9 Theoretical Query Plan of Query 3

- Observation in Oracle

The corresponding execution plan in Oracle is listed in Figure 3.10

```
    Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=98331 Card=10 Bytes=
          26747064)

   1    0   SORT (ORDER BY) (Cost=98331 Card=10 Bytes=26747064)
   2    1     SORT (GROUP BY) (Cost=98331 Card=10 Bytes=26747064)
   3    2       COUNT (STOPKEY)
   4    3         HASH JOIN (Cost=69846 Card=495316 Bytes=26747064)
   5    4           HASH JOIN (Cost=10291 Card=221026 Bytes=7514884)
   6    5             TABLE ACCESS (FULL) OF 'CUSTOMER' (Cost=1073 Car
          d=30000 Bytes=420000)

   7    5             TABLE ACCESS (FULL) OF 'ORDERS' (Cost=7504 Card=
          736703 Bytes=14734060)

   8    4           TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33893 Card
          =3202275 Bytes=64045500)
```

Figure 3.10 Execution Plan of Query3 in Oracle

We can see that the table customer and the table orders are accessed by a full scan method, and a hash-join is done on them. The result is used to join with another table lineitem.

- Observation in DB2

The upper part

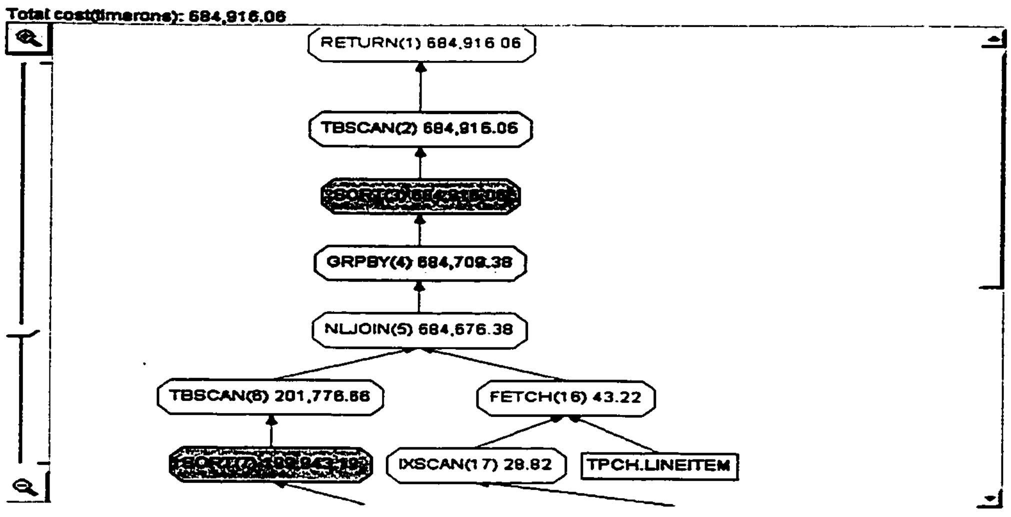


Figure 3.11 Execution Plan of Query3 in DB2 (Upper Part)

The lower part

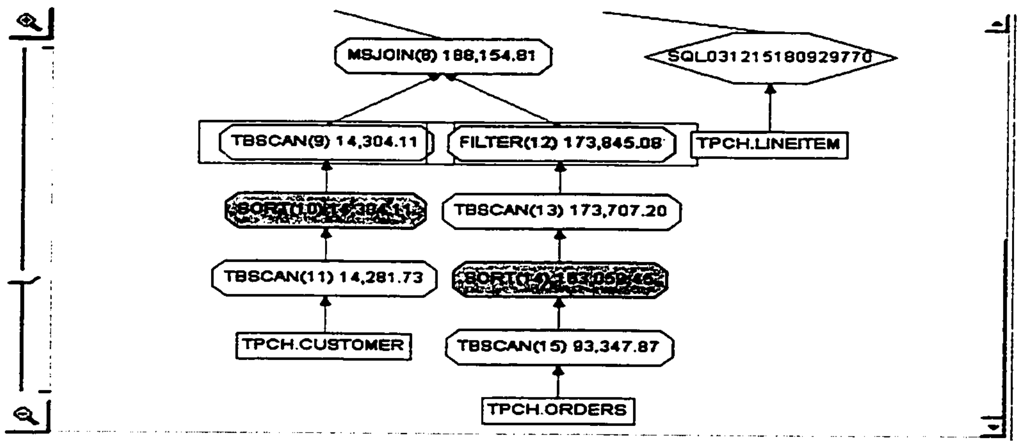


Figure 3.12 Execution Plan of Query3 in DB2 (Lower Part)

Similarly, DB2 uses the same sequence to join table, but a merge-join is used rather than a hash-join.

- Conclusion

Both Oracle and DB2 use similar execution plans except for the details of the join method. Oracle prefers hash-join, whereas DB2 prefers merge and nested-loop join method.

3.3.2 Non-correlated Subquery

```
SQL> select count(*)
       from
         orders
      where o_custkey In
         ( select c_custkey from customer where c_nationkey = 24);
```

In theory, the logical query execution plan of above query is as follows:

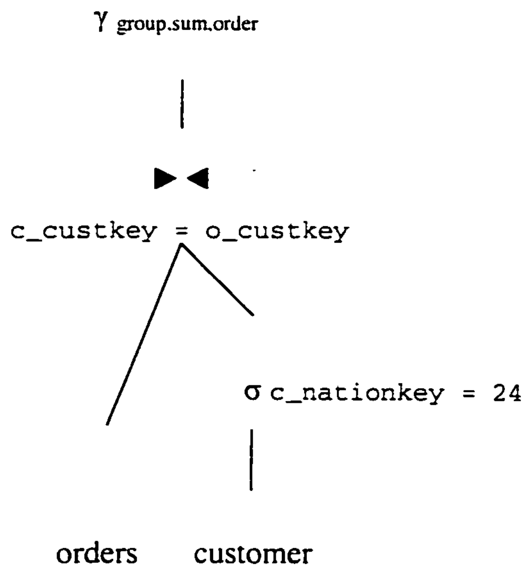


Figure 3.13 Query Plan of above SQL (In theory)

- Observation in Oracle

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4341 Card=1
Bytes=13
          )

   1    0   SORT (AGGREGATE)
   2    1     HASH JOIN (Cost=4341 Card=90004 Bytes=1170052)
   3    2       TABLE ACCESS (FULL) OF 'CUSTOMER' (Cost=521
Card=6000
          Bytes=48000)

   4    2       TABLE ACCESS (FULL) OF 'ORDERS' (Cost=3651
Card=150000
          0 Bytes=7500000)
```

Figure 3.14 Execution Plan of Non-correlated Query (In Oracle)

This execution plan in Oracle9i is the same as that in theory.

- Observation in DB2

When this query is executed on DB2, we get the following access plan.

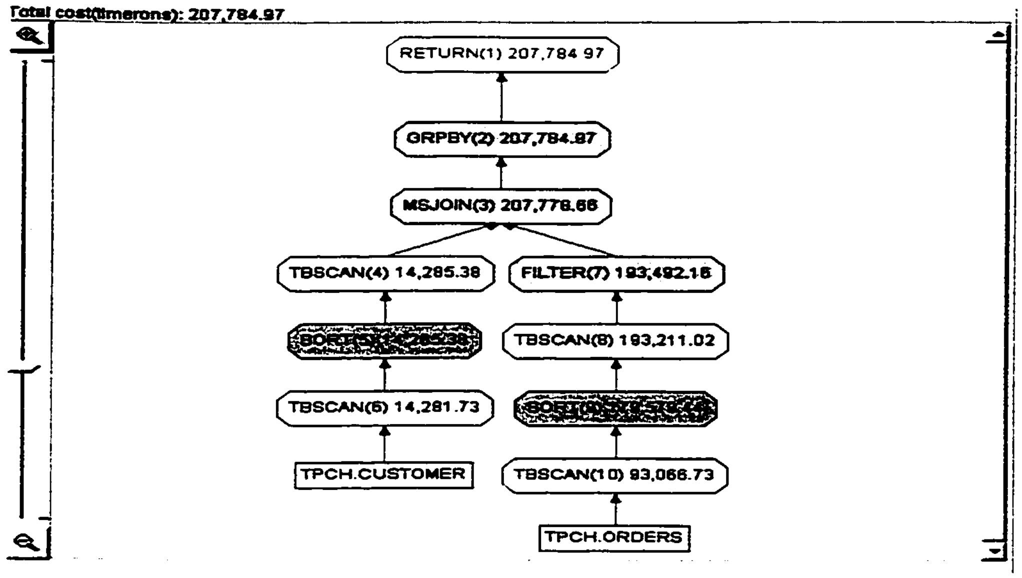


Figure 3.15 Execution Plan of Non-correlated Query (In DB2)

This execution plan in DB2 is the same as that in theory.

- Conclusions

Both Oracle and DB2 use similar execution plans except for details of the join method. Oracle prefers hash-join, whereas DB2 prefers merge-join.

3.3.3 Correlated Subquery

Query17 in TPC-H is a correlated subquery, shown as follows:

```
SQL> select
  2        sum(l_extendedprice) / 7.0 as avg_yearly
  3  from
  4        lineitem,
  5        part
  6  where
  7        p_partkey = l_partkey
  8        and p_brand = 'Brand#35'
```

```
    and p_container = 'JUMBO BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

Figure 3.16 SQL statement of Query17

- Theoretical Execution Tree Plan

First, we get the following execution plan tree manually

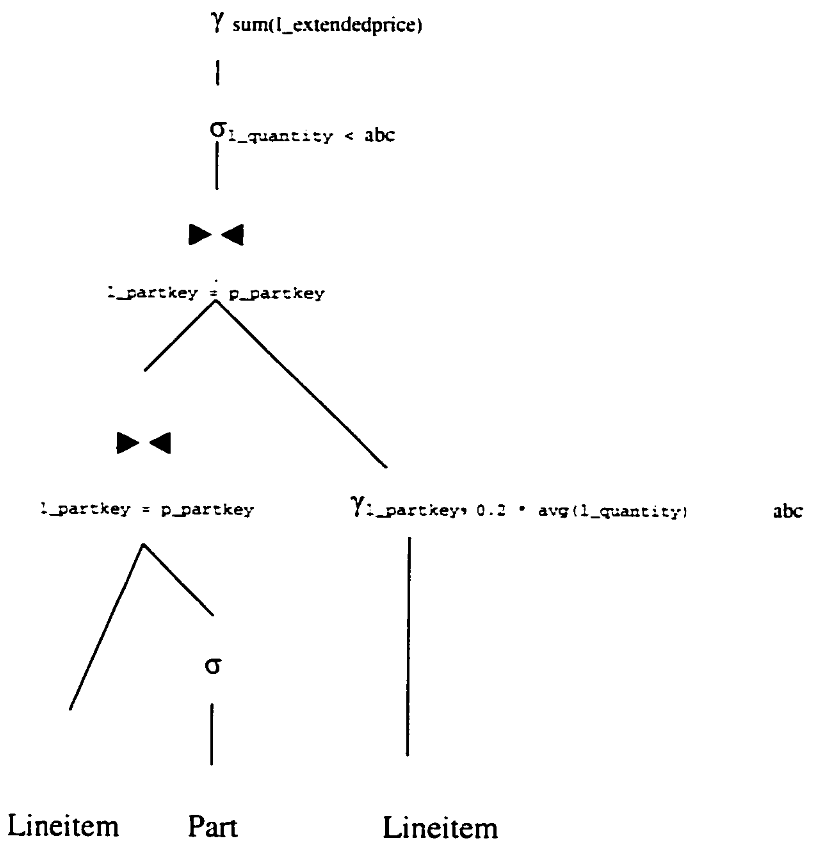


Figure 3.17 Query Plan of Query17 (In theory)

- Observation in Oracle

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=76679 Card=1 Bytes=1
          3)

   1    0   SORT (AGGREGATE)
   2    1     VIEW (Cost=76679 Card=9022 Bytes=117286)
   3    2       FILTER
   4    3         SORT (GROUP BY) (Cost=76679 Card=9022 Bytes=532298)
   5    4           HASH JOIN (Cost=72696 Card=180435 Bytes=10645665)
   6    5             HASH JOIN (Cost=35224 Card=6007 Bytes=312364)
   7    6               TABLE ACCESS (FULL) OF 'PART' (Cost=1183 Card=
          200 Bytes=6400)

   8    6               TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33893
          Card=6007239 Bytes=120144780)

   9    5             TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33893 Ca
             rd=6007239 Bytes=42050673)
```

Figure 3.18 Execution Plan of Query17 (In Oracle)

The execution plan tree in Oracle is similar to the theoretical one.

- Observation in DB2

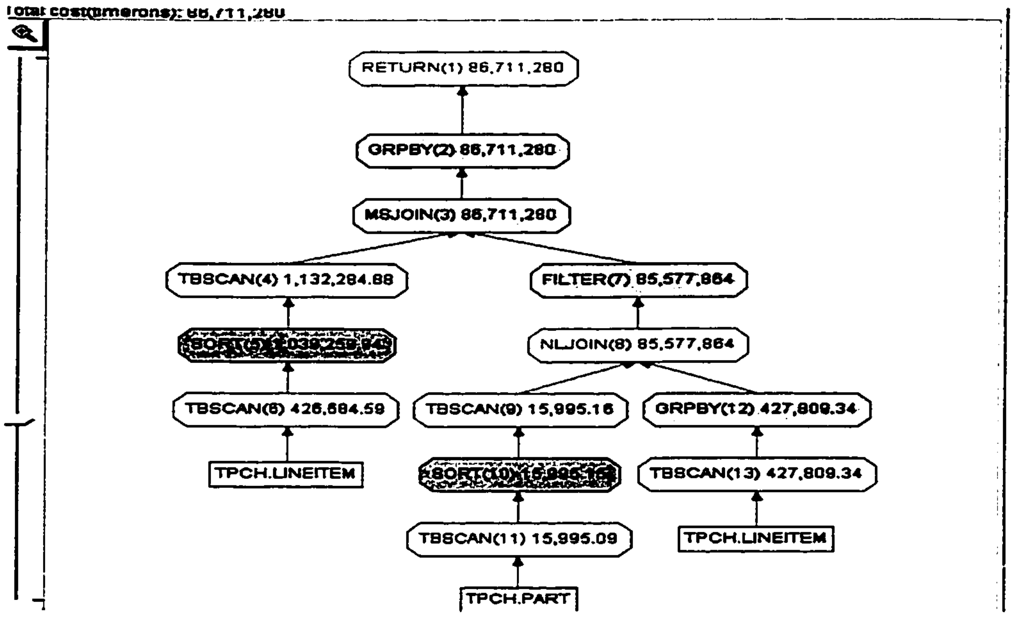


Figure 3.19 Execution Plan of Query17 (In DB2)

The execution plan tree in DB2 is similar to the theoretical one.

- Conclusion

  Both Oracle and DB2 create a similar plan tree to deal with the subquery.

3.4 Hashing and Sorting Join

Hash_based algorithms are often superior to sort-based algorithms since they require only one of their arguments to be "small", whereas sort-based algorithms work well when the size of their argument relations is large [4].

For example, we use the Query21 of TPC-H queries, which is shown as:

```
SQL> select
2           s_name,
3           count(*) as numwait
4   from
5           supplier,
6           lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
        and o_orderkey = l1.l_orderkey
        and o_orderstatus = 'F'
        and l1.l_receiptdate > l1.l_commitdate
        and exists (
              select
                    *
              from
                    lineitem l2
              where
                    l2.l_orderkey = l1.l_orderkey
                    and l2.l_suppkey <> l1.l_suppkey
        )
        and not exists (
              select
                    *
              from
                    lineitem l3
              where
                    l3.l_orderkey = l1.l_orderkey
                    and l3.l_suppkey <> l1.l_suppkey
                    and l3.l_receiptdate > l3.l_commitdate
        )
        and s_nationkey = n_nationkey
        and n_name = 'PERU'
        and rownum < 101
group by
        s_name
order by
        numwait desc,
        s_name;
```

Figure 3.20 SQL Statement of Query21 (In Oracle)

• Observation on Oracle 9i

In 1GB database, the optimizer selects the hash-based algorithm rather than the sort-based (merge sort) algorithm.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=148694 Card=100 Byte
          s=1320000)

   1    0   SORT (ORDER BY) (Cost=148694 Card=100 Bytes=1320000)
   2    1     SORT (GROUP BY) (Cost=148694 Card=100 Bytes=1320000)
   3    2       COUNT (STOPKEY)
   4    3         HASH JOIN (ANTI) (Cost=146373 Card=43112 Bytes=56907
          84)

   5    4           HASH JOIN (SEMI) (Cost=96745 Card=43117 Bytes=4570
          402)

   6    5             HASH JOIN (Cost=44907 Card=43117 Bytes=4139232)
   7    6               TABLE ACCESS (FULL) OF 'ORDERS' (Cost=7497 Car
          d=500000 Bytes=4000000)

   8    6               HASH JOIN (Cost=34163 Card=121716 Bytes=107110
          08)

   9    8                 HASH JOIN (Cost=67 Card=400 Bytes=24800)
  10    9                   TABLE ACCESS (FULL) OF 'NATION' (Cost=1 Ca
          rd=1 Bytes=29)

  11    9                   TABLE ACCESS (FULL) OF 'SUPPLIER' (Cost=65
           Card=10000 Bytes=330000)

  12    8                 TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=3385
          9 Card=3042903 Bytes=79115478)
  13     5              TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33859 Ca
           rd=6001215 Bytes=60012150)

  14     4            TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33859 Card
           =3042903 Bytes=79115478)


Elapsed: 00:02:35.09
```

Figure 3.21 Execution Plan for Query21 (1GB) in Oracle

We can see that, a hash-join was used in this query's execution plan in the 1GB database, whereas a merge-join is used at the outer join in the execution plan of the same query for a 10GB size.

```
      Execution Plan
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=2708976 Card=100 Byt
           es=13600000)

   1    0    SORT (ORDER BY) (Cost=2708976 Card=100 Bytes=13600000)
   2    1      SORT (GROUP BY) (Cost=2708976 Card=100 Bytes=13600000)
   3    2        COUNT (STOPKEY)
   4    3          MERGE JOIN (SEMI) (Cost=2665344 Card=430882 Bytes=58
           599952)

   5    4            MERGE JOIN (ANTI) (Cost=1557285 Card=430882 Bytes=
           53860250)

   6    5              MERGE JOIN (Cost=539931 Card=430887 Bytes=422269
           26)

   7    6                SORT (JOIN) (Cost=425577 Card=1216156 Bytes=10
           9454040)

   8    7                  HASH JOIN (Cost=371129 Card=1216156 Bytes=10
           9454040)
```

```
  9    8                    HASH JOIN (Cost=655 Card=4000 Bytes=252000
         )

 10    9                      TABLE ACCESS (FULL) OF 'NATION' (Cost=1
         Card=1 Bytes=29)

 11    9                      TABLE ACCESS (FULL) OF 'SUPPLIER' (Cost=
         652 Card=100000 Bytes=3400000)

 12    8                    TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=34
         2785 Card=30403889 Bytes=820905003)

 13    6                SORT (JOIN) (Cost=114355 Card=5000000 Bytes=40
         000000)

 14   13                  TABLE ACCESS (FULL) OF 'ORDERS' (Cost=75333
         Card=5000000 Bytes=40000000)

 15    5              FILTER
 16   15                SORT (JOIN)
 17   16                  TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=3427
         85 Card=30403889 Bytes=820905003)

 18    4            FILTER
 19   18              SORT (JOIN)
 20   19                TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=342785
          Card=59986052 Bytes=659846572)
```

Figure 3.22 Execution Plan for Query21 (10G) on Oracle 9i

Furthermore, the order of applying *antisemijoin* and *semijoin* (line 4-3, 5-4) is different in Figure 3.22 and Figure 3.23 because the latter order can eliminate the size of the result, whereas the former one is good for relatively small sized relations.

In order to understand why Oracle optimizer chooses a merge-join rather than a hash-join, at the outside level join for a 10GB size database, some hints are given to the optimizer. Thus, the hash-join operator is always used. So HASH_SJ is put into the EXISTS subquery, and HASH_AJ is put into the NOT IN subquery. Thus, query21 is modified as following:

```
SQL> select
             s_name,
             count(*) as numwait
     from
             supplier,
             lineitem l1,
             orders,
             nation
     where
             s_suppkey = l1.l_suppkey
             and o_orderkey = l1.l_orderkey
             and o_orderstatus = 'F'
             and l1.l_receiptdate > l1.l_commitdate
             and exists (
                     select [redacted]
                             *
                     from
                             lineitem l2
                     where
                             l2.l_orderkey = l1.l_orderkey
                             and l2.l_suppkey <> l1.l_suppkey
             )
             and not exists (
                     select
                             *
                     from
                             lineitem l3
                     where
                             l3.l_orderkey = l1.l_orderkey
                             and l3.l_suppkey <> l1.l_suppkey
                             and l3.l_receiptdate > l3.l_commitdate
             )
             and s_nationkey = n_nationkey
```

```
         and n_name = 'PERU'
         and rownum < 101
group by
         s_name
order by
         numwait desc,
         s_name;
```

Figure 3.23 SQL Statement of Query21 with hints (10GB) on Oracle

In Figure 3.24, the cost of the execution plan with hints is 3430785 blocks, which is larger than 2708976 (in Figure 3.23), the cost of the execution plan chosen by Oracle9i optimizer.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3430785 Card=100 Byt
          es=13600000)

   1    0   SORT (ORDER BY) (Cost=3430785 Card=100 Bytes=13600000)
   2    1     SORT (GROUP BY) (Cost=3430785 Card=100 Bytes=13600000)
   3    2       COUNT (STOPKEY)
   4    3         MERGE JOIN (ANTI) (Cost=3387153 Card=430882 Bytes=58
          599952)

   5    4           SORT (JOIN) (Cost=2369800 Card=430887 Bytes=469666
          83)

   6    5             HASH JOIN (SEMI) (Cost=2346789 Card=430887 Bytes
          =46966683)

   7    6               MERGE JOIN (Cost=539931 Card=430887 Bytes=4222
          6926)

   8    7                 SORT (JOIN) (Cost=425577 Card=1216156 Bytes=
          109454040)
```

```
 9     8                      HASH JOIN (Cost=371129 Card=1216156 Bytes=
         109454040)

10     9                        HASH JOIN (Cost=655 Card=4000 Bytes=2520
         00)

11    10                          TABLE ACCESS (FULL) OF 'NATION' (Cost=
         1 Card=1 Bytes=29)

12    10                          TABLE ACCESS (FULL) OF 'SUPPLIER' (Cos
         t=652 Card=100000 Bytes=3400000)

13     9                        TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=
         342785 Card=30403889 Bytes=820905003)

14     7                    SORT (JOIN) (Cost=114355 Card=5000000 Bytes=
         40000000)

15    14                      TABLE ACCESS (FULL) OF 'ORDERS' (Cost=7533
         3 Card=5000000 Bytes=40000000)

16     6                  TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=342785
          Card=59986052 Bytes=659846572)

17     4            FILTER
18    17              SORT (JOIN)
19    18                TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=342785
          Card=30403889 Bytes=820905003)
```

Figure 3.24 Execution Plan for Query21 with hints (10GB) on Oracle

Similarly, we force the optimizer to choose a merge-join method for Query21 on the 1GB data, by adding a merge-join hint to the inner subquery, enclosed in the EXISTS clause. As a result, the cost is higher than that of the execution plan the optimizer chooses. We can check the result in the following figure:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=183817 Card=100 Byte
          s=1320000)

   1    0   SORT (ORDER BY) (Cost=183817 Card=100 Bytes=1320000)
   2    1     SORT (GROUP BY) (Cost=183817 Card=100 Bytes=1320000)
   3    2       COUNT (STOPKEY)
   4    3         HASH JOIN (ANTI) (Cost=181496 Card=43112 Bytes=56907
          84)

   5    4           MERGE JOIN (SEMI) (Cost=131868 Card=43117 Bytes=45
          70402)

   6    5             SORT (JOIN) (Cost=46244 Card=43117 Bytes=4139232
          )

   7    6               HASH JOIN (Cost=44907 Card=43117 Bytes=4139232
          )

   8    7                 TABLE ACCESS (FULL) OF 'ORDERS' (Cost=7497 C
          ard=500000 Bytes=4000000)

   9    7                 HASH JOIN (Cost=34163 Card=121716 Bytes=1071
          1008)

  10    9                   HASH JOIN (Cost=67 Card=400 Bytes=24800)
  11   10                     TABLE ACCESS (FULL) OF 'NATION' (Cost=1
          Card=1 Bytes=29)

  12   10                     TABLE ACCESS (FULL) OF 'SUPPLIER' (Cost=
          65 Card=10000 Bytes=330000)

  13    9                   TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33
          859 Card=3042903 Bytes=79115478)

  14    5             FILTER
```

```
15   14                   SORT (JOIN)
16   15                     TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=3385
          9 Card=6001215 Bytes=60012150)

17    4              TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33859 Card
          =3042903 Bytes=79115478)

Elapsed: 00:02:48.06
```

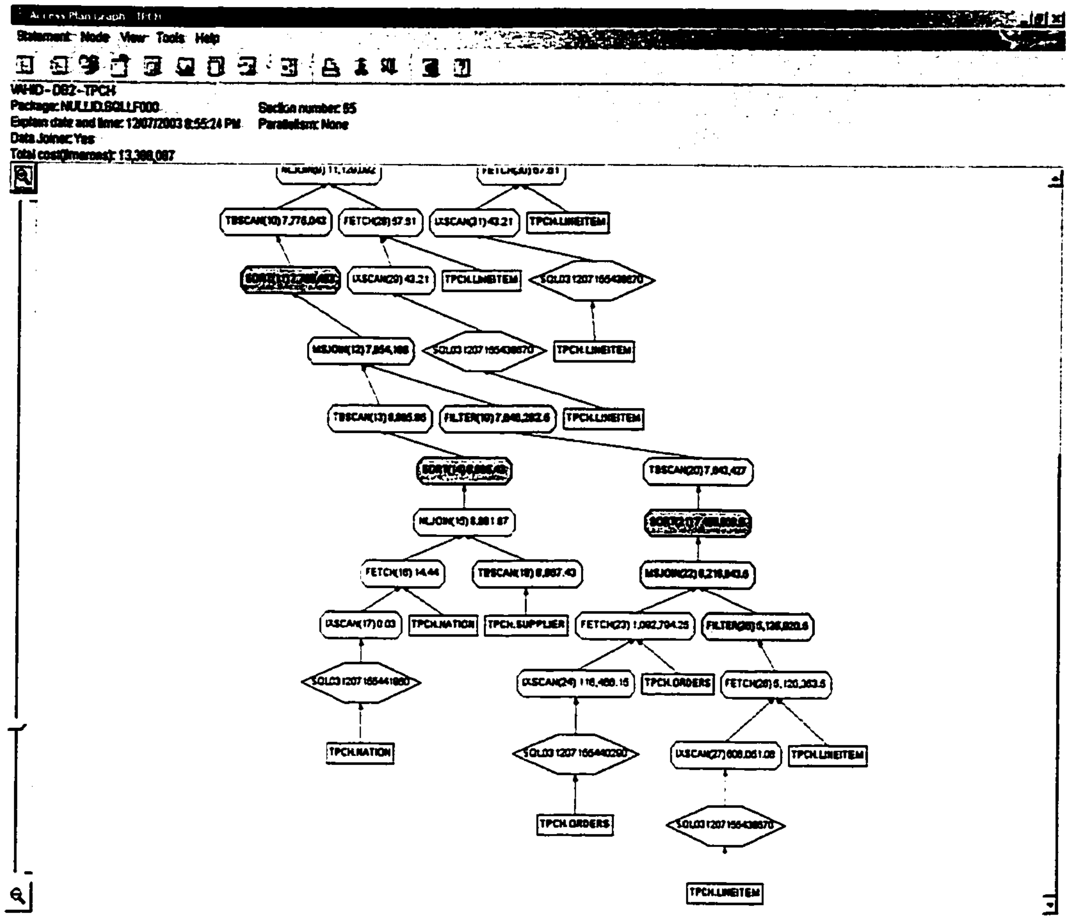
Figure 3.25 Execution Plan for Query21 with hints (1GB) on Oracle

The cost (183817) of the plan using the merge-join method is higher than 139933, the cost of the plan using the hash-join method in the 1GB size database for query 21. Furthermore, the timing of the former is longer than that of the latter.

The DBMS optimizers will use a hash-join rather than the merge-join as the join operator when the size of the table is relatively small because the overflow of the chain of hash table is heavy and leads to high costs of I/O.

- Observation on DB2 UDB 7.2

10GB (the lower part)



VAHID - DB2 - TPCH
Package: NULLID.SQLLF000 Section number: 65
Explain date and time: 12/07/2003 8:55:24 PM Parallelism: None
Data Joiner: Yes
Total cost(timerons): 13,388,087

Figure 3.26 Execution Plan (Lower Part) for Query21 (10GB) on DB2

1GB (the lower part)

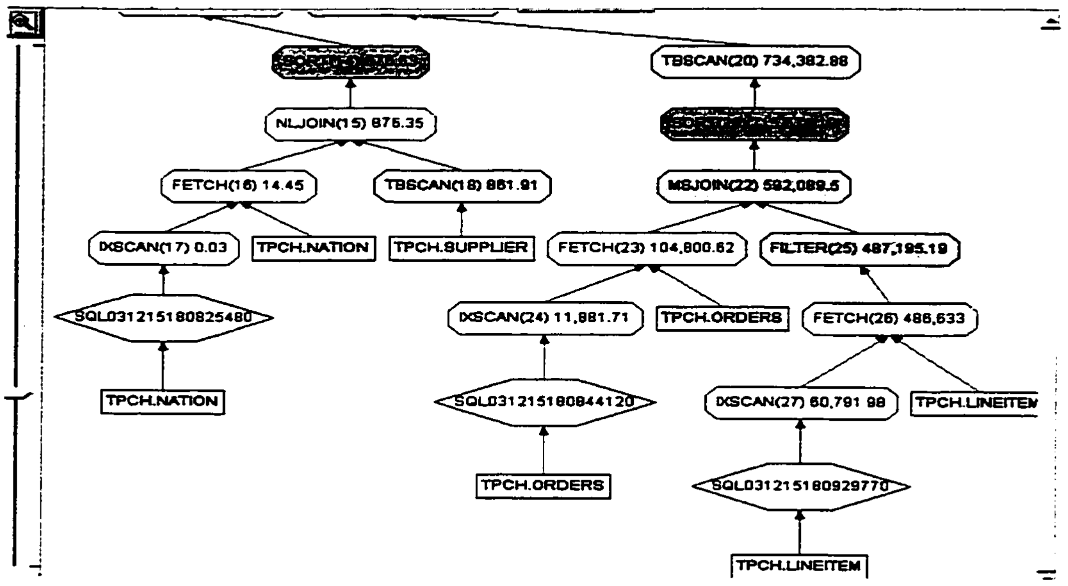


Figure 3.27 Execution Plan (Lower Part) for Query21 (1GB) on DB2

The execution plan tree for a 10GB size is the same as that adopted for a 1GB size on DB2.

- Conclusion

  Oracle can choose a kind of join method based on the size of the database. The hash-join is used when the database size is relatively small. DB2 is not as sensitive as Oracle, to the size of a database when choosing the execution plan.

3.5 Conclusions

The shapes of plan trees created manually (human being) for Oracle, and DB2 are the same. It does not matter whether the statement is a regular statement, a non-correlated subquery, or a correlated subquery. Oracle and DB2 do not follow separate steps to do a selection operation and this operation is done when the DBMS accesses the records in tables and a filter is used for a selection operation.

Which join methods (merge-join, nested-join, and hash-join) will be used by a DBMS, is a detailed strategy decided by the DBMS vector. However, Oracle can change the join method based on the database size.

4. Indices

In order to improve the speed of searching desired blocks on the tables, indices are often created on a relation. An index is any data structure that takes as input a property of records – typically the value of one or more fields – and finds the records with that property more efficiently [4].

In this section, we will study how the indices affect the cost of query. Improper indices will lead to a higher query cost. Bad indexing will cause a higher system maintenance overhead and a high-cost scan and join, without benefiting from the index.

4.1 Index and Non-Index

We will compare the execution result of Query22 with indices and without indices on the tables to which this query refers. We created an index `orders4` on table `orders` because it is a join attribute and created a composite index `customer3` on table `customer` as follows:

```
create index orders4 on orders(o_custkey) tablespace indx compute
statistics;

create index customer3 on customer(c_custkey,c_name,c_acctbal,c_phone)
tablespace indx compute statistics;
```

- Oracle

Without indexing, the cost of this query is 8607 block I/O and the execution time is about 13 seconds (Figure 4.1), whereas the cost of this query, using indexing, is 917 blocks I/O and the execution time is about 4 seconds (Figure 4.2). Furthermore, the optimizer just uses the composite index information to get the result without using an independent index on each field of the composite index fields.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=8607 Card=1 Bytes=32
          )

   1    0   SORT (GROUP BY) (Cost=8607 Card=1 Bytes=32)
   2    1     FILTER
   3    2       HASH JOIN (ANTI) (Cost=8603 Card=1 Bytes=32)
   4    3         TABLE ACCESS (FULL) OF 'CUSTOMER' (Cost=1073 Card=51
          0 Bytes=13770)

   5    3         TABLE ACCESS (FULL) OF 'ORDERS' (Cost=7497 Card=1500
          000 Bytes=7500000)

   6    2       SORT (AGGREGATE)
   7    6         TABLE ACCESS (FULL) OF 'CUSTOMER' (Cost=1073 Card=92
          64 Bytes=203808)

timing for: q22
Elapsed: 00:00:13.04
```

Figure 4.1 Execution Plan of Query22 without Index (1GB) in Oracle

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=917 Card=1 Bytes=25)
   1    0   SORT (GROUP BY) (Cost=917 Card=1 Bytes=25)
   2    1     FILTER
```

```
  3    2          NESTED LOOPS (ANTI) (Cost=913 Card=1 Bytes=25)
  4    3            INDEX (FAST FULL SCAN) OF 'CUSTOMER3' (NON-UNIQUE) (
        Cost=367 Card=510 Bytes=11220)


  5    3            INDEX (RANGE SCAN) OF 'ORDERS4' (NON-UNIQUE) (Cost=2
         Card=1501500 Bytes=4504500)


  6    2          SORT (AGGREGATE)
  7    6            INDEX (FAST FULL SCAN) OF 'CUSTOMER3' (NON-UNIQUE) (
        Cost=367 Card=9264 Bytes=176016)

timing for: q22
Elapsed: 00:00:04.01
```

Figure 4.2 Execution Plan of Query22 with Index (1GB) in Oracle

Furthermore, we create an index on each field in the statement rather than a composite index `customer3` using the following command.

```
create index customer5 on customer(c_phone);
create index customer6 on customer(c_name);
create index customer2 on customer(c_acctbal);
```

Then, we get similar results on other queries of TPC-H.

- DB2

The execution result of Query22 without indexing is shown as below:

```
CNTRYCODE  NUMCUST        TOTACCTBAL
--------------------------------------------------------------
10                 882                               6606081.31
14                 955                               7212285.84
15                 896                               6717441.72
16                 878                               6651791.79
19                 963                               7230776.82
```

```
20                  916                          6824676.02
22                  894                          6636740.03


Number of rows retrieved is:       7
Number of rows sent to output is:  7

Elapsed Time is:            109.55    seconds
```

Figure 4.3 Execution Result for Query22 without index (1GB) in DB2

The execution result of Query22 with indexing is shown in Figure 4.4.

```
CNTRYCODE  NUMCUST       TOTACCTBAL
---------------------------------------------------------------
10                  882                          6606081.31
14                  955                          7212285.84
15                  896                          6717441.72
16                  878                          6651791.79
19                  963                          7230776.82
20                  916                          6824676.02
22                  894                          6636740.03

Number of rows retrieved is:       7
Number of rows sent to output is:  7

Elapsed Time is:            9.54      seconds
```

Figure 4.4 Execution Result for Query22 with index (1GB) in DB2

Just 9.54 seconds are needed for the same query when the proper indices exist. So those indices are proper and useful.

- Conclusions

Undoubtedly, a proper index will improve the performance of the database. Furthermore, if composite index covers all fields in a SQL statement, then the optimizer will use the index to get the relevant rows, instead of using the full-table scan for accessing tables.

4.2 Improper Index

An improper index will not contribute to an improved performance in evaluating a query; instead, it will have an adverse impact on the system performance.

For example, when there are no indices on the tables lineitem and part, the elapsed time of Query19 is 42.01 seconds. The corresponding execution plan is given as follows:

```
Execution Plan
-----------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=40188 Card=1 Bytes=7
          6)

   1    0   SORT (AGGREGATE)
   2    1     HASH JOIN (Cost=40188 Card=315 Bytes=23940)
   3    2       TABLE ACCESS (FULL) OF 'PART' (Cost=1183 Card=200000 B
          ytes=5400000)

   4    2       TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33893 Card=398
          439 Bytes=19523511)

timing for: q19
Elapsed: 00:00:42.01
```

Figure 4.5 Execution Plan for Query19 without indices (1GB) in Oracle

The following indices were created on the table part and the table lineitem respectively and Query19 runs in the same environment. The elapsed time is up by 1391.08 seconds

```
create index part6 on part(p_container) tablespace indx
create index part2 on part(p_brand,p_type,p_size) tablespace indx

create index lineitem11 on lineitem(l_partkey) tablespace indx
create index lineitem9 on lineitem(l_shipinstruct) tablespace indx
```

When we checked the execution plan, we found that the above B+ tree indices are converted into bitmap indices. In order to understand the benefits of Bitmap indices for Query 19, we recreate those four indices as bitmap types using commands:

```
   create bitmap index part6 on part(p_container) tablespace indx
   create bitmap index part2 on part(p_brand,p_type,p_size) tablespace
indx

   create bitmap index lineitem11 on lineitem(l_partkey) tablespace indx
   create bitmap index lineitem9 on lineitem(l_shipinstruct) tablespace
   indx
```

Then we let Query19 run again and the cost is 5910 I/O blocks and the elapsed time is 46.01 seconds, which is much less than 1391.08 seconds needed when the B+ tree indices are created. But execution time (42.01 seconds) without indexing is lower than the execution time (46.01) with indexing. So the indices we have created for Query19 are not proper.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5910 Card=1
Bytes=80
          )
   1    0   SORT (AGGREGATE)
   2    1     CONCATENATION
   3    2         TABLE ACCESS (BY INDEX ROWID) OF 'LINEITEM'
 (Cost=1585
           Card=97252 Bytes=5057104)

   4    3           NESTED LOOPS (Cost=1585 Card=98 Bytes=7840)
   5    4             TABLE ACCESS (BY INDEX ROWID) OF 'PART' (Cost=365
          Card=78 Bytes=2184)

   6    5               BITMAP CONVERSION (TO ROWIDS)
   7    6                 BITMAP AND
   8    7                   BITMAP MERGE
   9    8                     BITMAP INDEX (RANGE SCAN) OF 'PART2'
  10    7                   BITMAP OR
  11   10                     BITMAP CONVERSION (FROM ROWIDS)
  12   11                       INDEX (RANGE SCAN) OF 'PART6' (NON-
UNIQU
          E) (Cost=43)

  13   10                     BITMAP CONVERSION (FROM ROWIDS)
  14   13                       INDEX (RANGE SCAN) OF 'PART6' (NON-
UNIQU
          E) (Cost=43)
...
   REVENUE
----------
3888904.26

Elapsed: 00:00:46.01
```

Figure 4.6 Execution Plan for Query19 with indices (1GB) in Oracle

- Conclusions

An improper index will adversely affect the performance of the query. Especially, a bitmap index is suited for columns with a small range and tables with lot of rows.

4.3 Indices on Small Tables

Theoretically speaking, indices on small tables can do more harm than good. However, in the following situations, creating an index on small tables can improve the performance of the system.

- If each record occupies an entire page. In this situation, whole table scan costs are high because an index scan requires few page accesses.

- If many updates are executed on a small table with no index, the table will be a bottleneck if the transactions update a single record [5]. This is because without the index, the full table scan will proceed before the specified record is located and locked.

4.4 Other Factors Affecting the Usage of Indices

Some factors that affect the usage of indexes:

- Out-of-Date Statistics Information

The cost-based optimizer chooses the execution plan based on the data distribution and storage characteristics of the tables, columns, indices, and other information [2].

If the statistics are out of date, then the optimizer may not use the 'old' statistics information. In order to utilize the statistics to improve performance, the statistics should be generated frequently and it should accurately reflect the information of the database. However, the frequent collection of statistics will burden the system, and the system will give a 'bad' performance for a normal job. So we should balance them appropriately.

- Data Type

The data type also affects the usage of index in the execution plan. This is illustrated in the following two examples.

i) Use of String Functions

If a string function is used on a string field over which an index is created, then the index will be useless for some queries.

For example, Query22 uses *substr()* function on the field `c_phone` of table customer and there is an index on field `c_phone`. However, the optimizer does not use the indices when the optimizer chooses an execution plan.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=8614 Card=1
Bytes=31
          )
   1    0   SORT (GROUP BY) (Cost=8614 Card=1 Bytes=31)
   2    1     FILTER
   3    2       HASH JOIN (ANTI) (Cost=8610 Card=1 Bytes=31)
   4    3         TABLE ACCESS (FULL) OF 'CUSTOMER' (Cost=1073
Card=51
          0 Bytes=13260)

   5    3         TABLE ACCESS (FULL) OF 'ORDERS' (Cost=7504
Card=1501
          500 Bytes=7507500)

   6    2       SORT (AGGREGATE)
   7    6         TABLE ACCESS (FULL) OF 'CUSTOMER' (Cost=1073
Card=92
    64 Bytes=194544)
```

Figure 4.7 Execution Plan for Query22 (1GB) on Oracle

- Improper Data Type for Bind Variables

A bind variable (a variable set by the programming language) may have a different type than the attribute to which it is being compared. In this case, the index may not be used.

For example, when we issue the following statement in a TPC-H testing database,

```
Select count(*) from lineitem where l_quantity is NULL
```

The execution plan listed in Figure 4.8 shows that the optimizer does not choose the index on field l_quantity because the attribute type of `l_quantity` is integer, and this type does not match NULL.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=33893 Card=1
Bytes=2
          )

   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=33893
Card=1 Byt
    es=2)
```

Figure 4.8 Execution Plan Of Above Statement (1GB) on Oracle

However, if we use the equality relation (=) instead of *is* in the same query:

```
Select count(*) from lineitem where l_quantity = NULL
```

The index on l_quantity will be used. We can check this in Figure 4.9.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=6 Card=1
Bytes=7)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (RANGE SCAN) OF 'ORDERS1' (NON-UNIQUE) (Cost=6
Car
          d=624 Bytes=4368)
```

Figure 4.9 Execution Plan of Above Statement (1GB) in Oracle

4.5 Date Distribution

Basically, an index on a table will be used when a query statement returns a small percentage of rows in the table. If just a few number of tuples meet the query condition, then using a proper index is useful because only few blocks of a table are accessed.

- Observation in Oracle

In Oracle, when a small number of rows in a table are selected in a query, then the DBMS will use the index rather than the full-table scan. In order to verify this, let us consider the following SQL statement 1.

```
select * from lineitem where l_shipdate <= to_date(date '1992-05-26')
union
select * from lineitem where l_shipdate > to_date(date '1992-05-26')
```

In this case, no index will be used although there is a B$^+$ Tree index on field `o_shipdate`. The corresponding execution plan in Oracle is shown:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=250700 Card=6009617
          Bytes=667067487)

   1    0   SORT (UNIQUE) (Cost=250700 Card=6009617 Bytes=667067487)
   2    1     UNION-ALL
   3    2       TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=16493 Card=347
          348 Bytes=38555628)

   4    2       TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=16493 Card=566
          2269 Bytes=628511859)
```

Figure 4.10 Execution Plan of SQL Statement 1 for Data Distribution Test (Oracle)

Although selectivity of the second statement is very low (0.6), Oracle turned out to be smart in this and it did not use index scan to access the table.

Second, we test the SQL statement 2 as below:

```
select * from lineitem where l_shipdate <= to_date(date '1992-05-26')
union
select * from lineitem where l_shipdate > to_date(date '1998-08-26')
```

The percentage of qualified tuples in table orders (1GB) is below 4% in the second part of statement 2. However, Oracle did not use the index. The predicate in the second subquery of statement 2 is modified to `select * from lineitem where l_shipdate > date('1998-11-24')` and we get a new statement 3 shown as follows:

```
select * from lineitem where l_shipdate <= to_date(date'1992-05-26')
union
select * from lineitem where l_shipdate > to_date(date'1998-11-24')
```

Oracle uses the index and the corresponding execution plan in Oracle9i is shown as following:

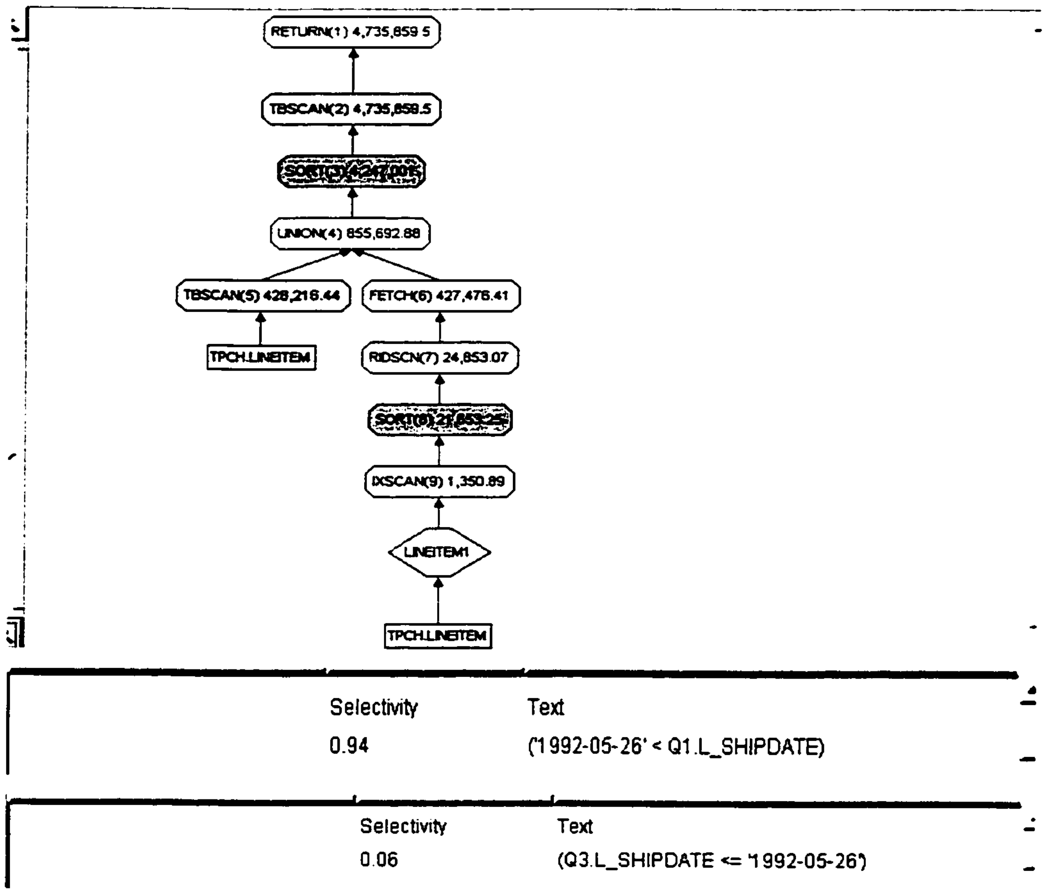
```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=42528 Card=364002 By
          tes=40404222)

   1    0   SORT (UNIQUE) (Cost=42528 Card=364002 Bytes=40404222)
   2    1     UNION-ALL
   3    2       TABLE ACCESS (FULL) OF 'LINEITEM' (Cost=16493 Card=347
          348 Bytes=38555628)

   4    2       TABLE ACCESS (BY INDEX ROWID) OF 'LINEITEM' (Cost=1627
          4 Card=16654 Bytes=1848594)

   5    4         INDEX (RANGE SCAN) OF 'LINEITEM1' (NON-UNIQUE) (Cost
          =47 Card=16654)
```

Figure 4.11 Execution Plan of Statement 3 for the Data Distribution Test in Oracle

- Observation in DB2

In DB2, the same SQL statement is used except for some minor changes in syntax:

```
select * from lineitem where l_shipdate <= date('1992-05-26')
union
select * from lineitem where l_shipdate > date('1992-05-26')
```



Selectivity	Text
0.94	('1992-05-26' < Q1.L_SHIPDATE)

Selectivity	Text
0.06	(Q3.L_SHIPDATE <= '1992-05-26')

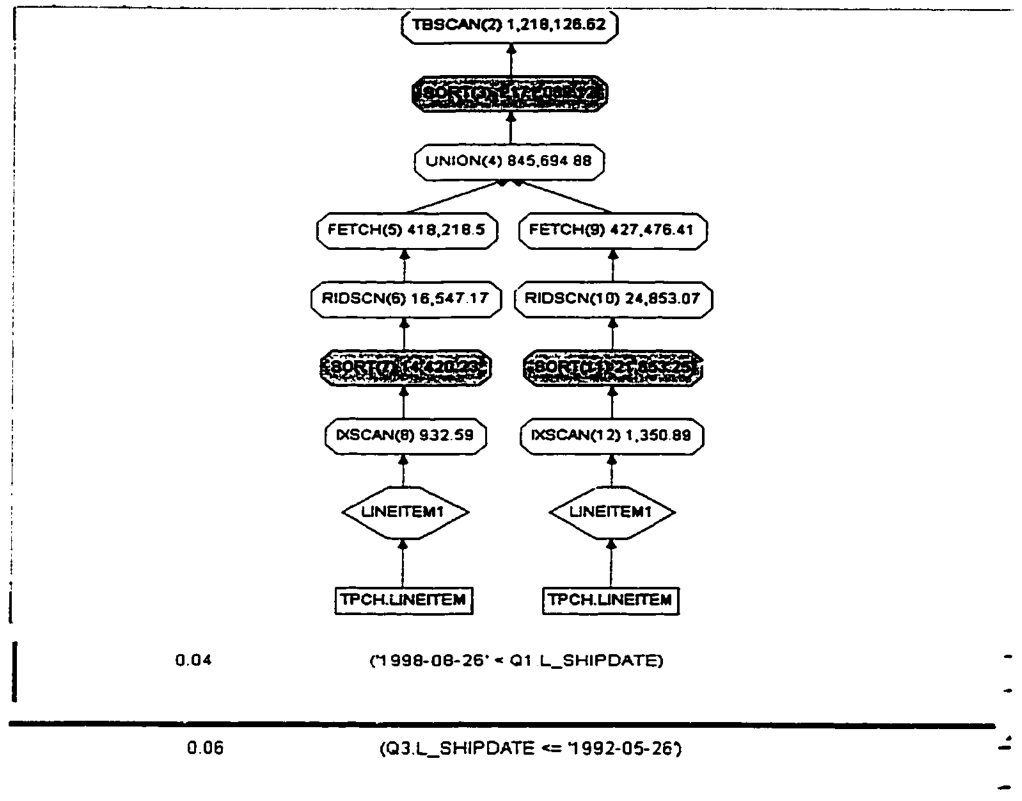
Figure 4.12 Execution Plan of Statement 1 for the Data Distribution Test (DB2)

It is clear that no index is used for SQL statement `select * from lineitem where l_shipdate > date('1992-05-26')` because the selectivity of its predicate is 94%, whereas the index `lineitem1` on field l_shipdate is used for the statement `select * from lineitem where l_shipdate <= date('1992-05-26')` because the selectivity of its predicate is 6%. Based on our experiments, we found that DB2 uses an index wherever the selectivity is below 6%. Once the selectivity is greater than 6%, no index will be used to access the table.

Now, the following statement is checked on the DB2,

```
select * from lineitem where l_shipdate <= date('1992-05-26')
```

```
union
select * from lineitem where l_shipdate > date('1998-08-26')
```



```
0.04     ('1998-08-26' < Q1 L_SHIPDATE)
0.06     (Q3.L_SHIPDATE <= '1992-05-26')
```

Figure 4.13 Execution Plan of Statement 2 for the Data Distribution Test (DB2)

- Conclusions

Whether the optimizer uses indexing or not depends on the data distribution in the predicate. If the statement results in a small portion of data, a proper index on the field in the predicate will be used; otherwise, no indexing will be used.

5. Rewriting SQL Statements

It is known that nested subquery has poor performance. So, converting nested selects to joins is a goal for all RDBMS vendors [9]. No matter, DB2 or Oracle claims to perform automatic transformation when required. When checking the execution plan of correlated queries, we can find that join methods were used. From the above analysis of the execution plan of DB2 and Oracle, we also can see that both convert correlated subqueries into join operations. Furthermore, UNION can substitute operator OR in the WHERE clause for some queries. However, not all rewriting will improve the performance of the query for nested query. We can rewrite Query17 into the following format, using the join operator to substitute the nested subquery.

Rewriting Query17

```
SQL> select
         sum(l1.l_extendedprice) / 7.0 as avg_yearly
  from
         lineitem l1,
         part,
         (select l_partkey, 0.2*avg(l_quantity) abc
         from lineitem
         group by l_partkey) l2
       where
         p_partkey = l1.l_partkey
         and l1.l_partkey = l2.l_partkey
         and p_brand = 'Brand#35'
         and p_container = 'JUMBO BOX'
         and l1.l_quantity < l2.abc
```

Figure 5.1 Rewritten SQL Statement of Query17 in Oracle

- Oracle

When the above query is executed on the database similar to running the original Query17, the execution time is 66.02 seconds, while the execution time of original Query17 is 67.08 seconds. So almost no performance improvement was achieved from rewritten query. The performance improvement depends on the selectivity of the query. If the selectivity of this query is high, then the rewriting pays off, because the intermediate table is small, otherwise the original query is not any worse than the rewritten one.

- DB2

The result of evaluating the rewritten Query17 is listed below. The execution time in this case is just 2973.39 seconds, far less than the original one, 6997.57 seconds.

```
AVG_YEARLY
------------------------------------
                          342461.8


Number of rows retrieved is:       1
Number of rows sent to output is:  1

Elapsed Time is:           2973.39   seconds
```

Figure 5.2 Execution Result of Rewritten Query17 in DB2

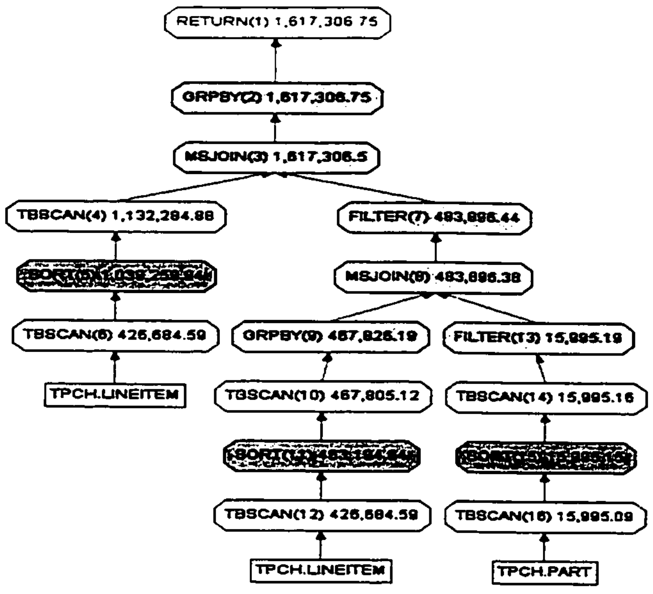


Figure 5.3 Access Plan of Rewritten Query17 in DB2

The execution time of the original Query17 is shown below:

```
AVG_YEARLY
------------------------------------
                          342461.8

Number of rows retrieved is:        1
Number of rows sent to output is:   1

Elapsed Time is:           6997.57   seconds
```

Figure 5.4 Execution result of original SQL statement of Query17 (DB2)

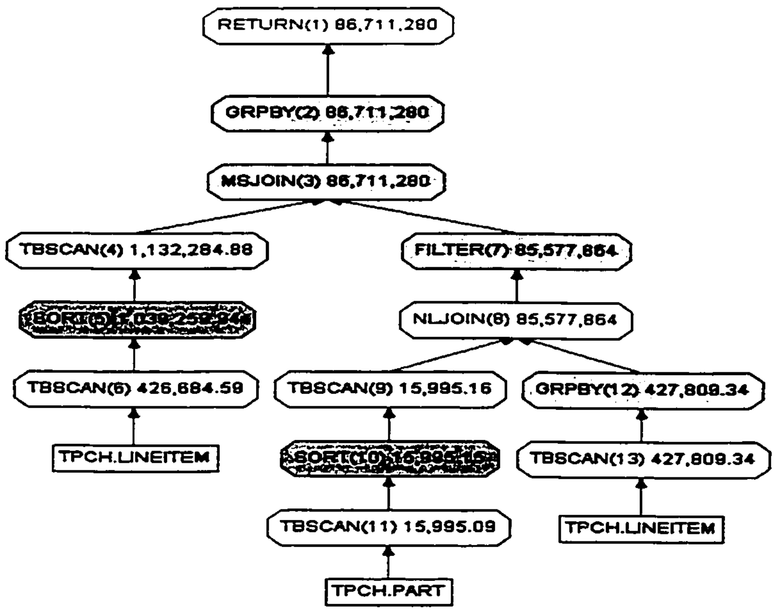


Figure 5.5 Access plan of original SQL Statement of Query17 (DB2)

In DB2, the costs of the two queries are different because they use different access plans. For rewritten statement Query17, the selection operators on the table Part were executed before joining with the table lineitem, on which a Group function was done first before join. So the intermediate relation is small. Whereas for original statement Query17 no filter was done before join operation between the tables part and lineitem, thus, the intermediate relation is bigger than that of the rewritten Query17.

As a result, execution time of rewritten Query17 is 2973.39 seconds and it is shorter than that of the original Query17, whose execution time is 6997.57 seconds.

- Conclusions

Rewriting nested-subquery into join operator usually can achieve a better performance, at least, which is not worse than the performance of original query.

6. Tuning Tools

Modem DBMS normally provide tools to let the DBA or implementers evaluate the performance. Oracle provides Monitor and Hint mechanisms, whereas DB2 provides a Monitor and an Index Advisor.

In this section, we will review some relevant tools in Oracle and DB2.

6.1 Optimizer Hint in Oracle 9i

Optimizer hint provided by Oracle gives the application designer the ability to specify the execution plan (partly) because the designer may know what the optimizer does not know [2]. Furthermore, optimizer recognizes the hint only when a cost-based approach is adopted.

Some join operators (merge-join, hash-join) have been checked in section 2.2 by giving a hint. Here, the hints on index and join order are considered, and the effect of hints will be explored. Actually, we can hint on an index, a join order, and an access path.

6.2 Tools in DB2

6.2.1 Index Advisor

The Index Advisor is a management tool that reduces the need for a user to design and define suitable indices for his data.

The Index Advisor is good for:

- Finding the best indices for a given query.

- Finding the best indices for a set of queries (a workload), subject to resource limits which are optionally applied.
- Testing an index on a workload without having to create the index.

The explain tables, which are used to store the execution plan information, must be created before execution plan can be invoked. Two concepts associated with this facility are *work load* and *virtual indices*. The former concept is a set of SQL statements that should be processed over a given period of time. The index advisor uses this workload information in conjunction with the database information to recommend indices. On the other hand, *virtual indices* refer to indices that do not exist in the current database schema. These indices are recommended or are being proposed to create [10].

Since we did not have access to index advisor in our lab, we could not carry out our TPC-H benchmark experiments to verify the utility of the index advisor.

6.2.2 Performance Monitor

The performance monitor can be used to check an existing problem or to observe the performance of the system. It gives a snapshot of the database activity and performance data at a point in time. This information can help identify and analyze potential problems, or identify exception conditions that are based on thresholds. Use of performance tool is recommended if the performance of the database manager and its database applications at a particular point are needed to be known. It is used also to get a visual overview of what elements are in a state of alarm. This helps to identify which parameters may need tuning. Thus, the DBA or application developer can then look closely at the parameters that have been set for that element and change them to improve performance.

The Performance Monitor provides information about the state of DB2 Universal Database and the data that it controls. DBA or application developer can define thresholds or zones that trigger warnings or alarms when the values that are being collected by the Performance Monitor are not within acceptable ranges.

Many objects can be monitored, such as instances, databases, tables, table spaces, and connections by selecting the object in the Object Tree pane or in the Contents pane and clicking the right mouse button. Different colors are used to represent the different status of the monitored objects.

We can use the information from the Performance Monitor to:

- Detect performance problems
- Tune databases for optimum performance
- Analyze performance trends
- Analyze the performance of database applications
- Prevent problems from occurring

Conclusions

In our TPC-H benchmark experimentation, Oracle demonstrated better performance over DB2 on our IBM desktop computer with Windows2000 operating system, irrespective of whether it has a 1GB or a 10GB database size. Our experiment demonstrated that Oracle is more suitable for small size database than DB2, in the Windows2000 environment, and thus, for small to medium enterprises Oracle would be the system of choice. On the other hand, Oracle needs much more time than DB2 to collect statistical information about the database contents that is used to calculate the cost of query execution plans during query optimization process.

On the application level performance tuning, we can get the following conclusions:

- The statistical information is very important for the DBMS to select an execution plan, which determines the performance of the system. Complete statistics can improve the performance of a DBMS because the SQL compiler has accurate information about the data and hence chooses the best execution plan. Both Oracle and DB2 utilize the statistical information to decide on the query execution plan with the least cost.

  How a DBMS chooses an execution plan depends on many factors: its approach (cost-based or rule-based), available statistics, efficiency of the join algorithms, data distribution, etc.

- Index is another factor that has a great impact on the system performance. A proper index can improve system performance, whereas an improper index will hinder performance. Furthermore, the index on a small table should be avoided in normal situations. Inconsistent data type of bind variable and the use of functions on a field, on which there is an index in SQL statement, may render the index useless. Oracle provides many different kinds of indices that are suitable for different queries while DB2 supports only B+ tree indices.

- Rewriting nested subquery is another way to improve the system performance.

Furthermore, most DBMS provide useful tools for performance analysis and tuning. The DBA and application developers can utilize such tools to get better performance results.

Related Work

One of our graduate students in our database research group is working on the same test on the Linux system. Our results agree closely in that Oracle has better performance for 1GB and 10GB size databases no matter in Windows2000 OS or Linux OS environments. We also observed that both Oracle and DB2 have better performance in the Linux OS than in Windows2000 OS.

IBM and Oracle companies also published some testing results, based on 1000G-database size. However, they used extraordinary servers with multiple CPUs and the memory of the server is up to 1G. Furthermore, they used different values for parameters in the TPC-H queries from what we used in testing.

The Sense of the Project

With the help of this project, we can get several benefits. First, we can learn how to manipulate the commercial DBMS: DB2 Universal Database 7.2 and Oracle 9i and tools provided by them. For example, we can monitor the system performance using the Performance Monitor in DB2, whereas we can use Index Hint tool in Oracle to cite the database engine in Oracle.

Second, the project can help us realize which DBMS has better performance over which operating systems and help us realize which factors, such as values of the DBMS parameters and queries' property (hit ratio), affect the execution performance of the query. At last, the decision about choosing appropriate DBMS on certain operating systems can be made.

Third, we can observe how a specific DBMS realizes the execution plan (tree) and compares the database theories with the real application in commercial DBMS.

Finally, we can learn how to do application level performance tuning.

References:

[1] TPC Organization, *TPC BenchMark^{TM}H (decision support) Standard Specification Reversion 1.5.0*, July 2002.

[2] Oracle Company, *Oracle9i Performance Tuning and Guide Reference – Release 2 (9.2)*, March 2002.

[3] Oracle Company, Oracle9i Database Administrator's Guide – Release 2 (9.2), March 2002.

[4] H. Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, *Database System Implementation*, Prentice Hall, 2000.

[5] D. Shasha and P. Bonnet, *Database Tuning – Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann, June 2002.

[6] Oracle Company, *Oracle9i SQL Reference*, March 2002.

[7] Qi Ch, Jarek G, Fred K, Cliff L, Linqi L, Xiaoyan Q, and Bernhard S, *Implementation of two semantic query optimization Techniques in DB2 Universal Database*, Proc.25th VLDB Conference. 1999: 687-698

[8] Trevor M., *Tuning Oracle Application*, http://www.wrsystems.com/whitepapers/Tuning.pdf, Accessed Nov. 5, 2002.

[9] Kosciuszko E. *Optimizing SQL: Rewriting SQL Subqueries Into Joins*, http://www.oracleprofessionalnewsletter.com /OP/OPmag.nsf/ 0/906E797FF8 6F4E3F852568F00066F6A6, Accessed Nov. 10, 2002.

[10] Waterloo University, *DB2 User Information,* http://www.student.math.uwaterloo.ca/~cs448/db2_doc/html/ db2help/index.htm#cncpve, Accessed Oct. 15, 2002.

Appendix A. **Queries Sequence**

The sequence of the 22 queries is definded in the TPC-H document, seeing [1]. Here, we just listed out the sequences used in our TPC-H experiment as below :

Power Test:

Stream0: 14, 2, 9, 20, 6, 17, 18, 8, 21, 13, 3, 22, 16, 4, 11, 15, 1, 10, 19, 5, 7, 12.

Throughput Test:

Stream1: 21, 3, 18, 5, 11, 7, 6, 20, 17, 12, 16, 15, 13, 10, 2, 8, 14, 19, 9, 22, 1, 4.

Stream2: 6, 17, 14, 16, 19, 10, 9, 2, 15, 8, 5, 22, 12, 7, 13, 18, 1, 4, 20, 3, 11, 21.

Stream3: 8, 5, 4, 6, 17, 7, 1, 18, 22, 14, 9, 10, 15, 11, 20, 2, 21, 19, 13, 16, 12, 3.

Appendix B. **Symbols Used in This Paper**

1. $\bowtie$	Join
2. γ	Aggregation Function
3. π	Projection
4. σ	Selection
5. δ	Distinct

Appendix C. **Examples of Queries and Refresh Functions in TPC-H**

using 807140330 as a seed to the RNG

Query1:

```
select
      l_returnflag,
      l_linestatus,
      sum(l_quantity) as sum_qty,
      sum(l_extendedprice) as sum_base_price,
      sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
      sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
      avg(l_quantity) as avg_qty,
      avg(l_extendedprice) as avg_price,
      avg(l_discount) as avg_disc,
      count(*) as count_order
from
      lineitem
where
      l_shipdate <= to_date(date '1998-12-01' - interval '104' day (3))
group by
      l_returnflag,
      l_linestatus
order by
      l_returnflag,
      l_linestatus;
```

Query 2:

```
select
      s_acctbal,
        s_name,
        n_name,
        p_partkey,
        p_mfgr,
        s_address,
        s_phone,
        s_comment
from
        part,
        supplier,
        partsupp,
        nation,
        region
where
        p_partkey = ps_partkey
        and s_suppkey = ps_suppkey
        and p_size = 6
        and p_type like '%TIN'
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'MIDDLE EAST'
        and ps_supplycost = (
                select
                        min(ps_supplycost)
                from
                        partsupp,
                        supplier,
                        nation,
                        region
                where
                        p_partkey = ps_partkey
```

```
                    and s_suppkey = ps_suppkey
                    and s_nationkey = n_nationkey
                    and n_regionkey = r_regionkey
                    and r_name = 'MIDDLE EAST'
          )
          and rownum < 101
order by
          s_acctbal desc,
          n_name,
          s_name,
          p_partkey;
```

Query 3:

```
select
          l_orderkey,
          sum(l_extendedprice * (1 - l_discount)) as revenue,
          o_orderdate,
          o_shippriority
from
          customer,
          orders,
          lineitem
where
          c_mktsegment = 'MACHINERY'
          and c_custkey = o_custkey
          and l_orderkey = o_orderkey
          and o_orderdate < to_date (date '1995-03-26')
          and l_shipdate > to_date (date '1995-03-26')
          and rownum < 11
group by
          l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate;
```

Query 4:

```
select
        o_orderpriority,
        count(*) as order_count
from
        orders
where
        o_orderdate >= to_date (date '1997-01-01')
        and o_orderdate < to_date (date '1997-01-01' + interval '3' month)
        and exists (
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority;
```

Query 5:

```
select
```

```
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'MIDDLE EAST'
        and o_orderdate >= to_date (date '1993-01-01')
        and o_orderdate < to_date (date '1993-01-01' + interval '1' year)
group by
        n_name
order by
        revenue desc;
```

Query 6:

```
select
        sum(l_extendedprice * l_discount) as revenue
from
        lineitem
where
        l_shipdate >= to_date (date '1993-01-01')
```

```
    and l_shipdate < to_date (date '1993-01-01' + interval '1' year)
    and l_discount between 0.06 - 0.01 and 0.06 + 0.01
    and l_quantity < 24;

Query 7:
select
    supp_nation,
    cust_nation,
    l_year,
    sum(volume) as revenue
from
    (
        select
            n1.n_name as supp_nation,
            n2.n_name as cust_nation,
            extract(year from l_shipdate) as l_year,
            l_extendedprice * (1 - l_discount) as volume
        from
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2
        where
            s_suppkey = l_suppkey
            and o_orderkey = l_orderkey
            and c_custkey = o_custkey
            and s_nationkey = n1.n_nationkey
            and c_nationkey = n2.n_nationkey
            and (
                    ( n1.n_name = 'KENYA' and n2.n_name = 'EGYPT' )
                    or
                    ( n1.n_name = 'EGYPT' and n2.n_name = 'KENYA' )
                )
                and l_shipdate between to_date (date  '1995-01-01') and to_date (date
'1996-12-31')
    ) shipping
group by
    supp_nation,
    cust_nation,
    l_year
order by
    supp_nation,
    cust_nation,
    l_year;

Query 8:
select
    o_year,
    sum(case
        when nation = 'EGYPT' then volume
        else 0
    end) / sum(volume) as mkt_share
from
    (
        select
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
            n2.n_name as nation
        from
            part,
                    supplier,
                    lineitem,
                    orders,
                    customer,
                    nation n1,
                    nation n2,
                    region
               where
                    p_partkey = l_partkey
                    and s_suppkey = l_suppkey
                    and l_orderkey = o_orderkey
                    and o_custkey = c_custkey
                    and c_nationkey = n1.n_nationkey
                    and n1.n_regionkey = r_regionkey
                    and r_name = 'MIDDLE EAST'
                    and s_nationkey = n2.n_nationkey
                    and o_orderdate between to_date (date '1995-01-01') and to_date (date
'1996-12-31')
                    and p_type = 'MEDIUM BRUSHED NICKEL'
     ) all_nations
group by
     o_year
order by
     o_year;
```

Query 9:

```
select
     nation,
     o_year,
     sum(amount) as sum_profit
from
```

```
        (
            select
                n_name as nation,
                extract(year from o_orderdate) as o_year,
                l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
            from
                part,
                supplier,
                lineitem,
                partsupp,
                orders,
                nation
            where
                s_suppkey = l_suppkey
                and ps_suppkey = l_suppkey
                and ps_partkey = l_partkey
                and p_partkey = l_partkey
                and o_orderkey = l_orderkey
                and s_nationkey = n_nationkey
                and p_name like '%light%'
        ) profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc;
```

Query 10:

```
select
        c_custkey,
        c_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        c_acctbal,
        n_name,
        c_address,
        c_phone,
        c_comment
from
        customer,
        orders,
        lineitem,
        nation
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate >= to_date(date '1993-11-01')
        and o_orderdate < to_date(date '1993-11-01' + interval '3' month)
        and l_returnflag = 'R'
        and c_nationkey = n_nationkey
        and rownum < 21
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
        c_comment
order by
        revenue desc;
```

Query 11:

```
select
      ps_partkey,
      sum(ps_supplycost * ps_availqty) as value
from
      partsupp,
      supplier,
      nation
where
      ps_suppkey = s_suppkey
      and s_nationkey = n_nationkey
      and n_name = 'UNITED KINGDOM'
group by
      ps_partkey having
            sum(ps_supplycost * ps_availqty) > (
                  select
                        sum(ps_supplycost * ps_availqty) * 0.0001000000
                  from
                        partsupp,
                        supplier,
                        nation
                  where
                        ps_suppkey = s_suppkey
                        and s_nationkey = n_nationkey
                        and n_name = 'UNITED KINGDOM'
            )
order by
      value desc;
```

Query 12:

```
select
    l_shipmode,
    sum(case
        when o_orderpriority = '1-URGENT'
            or o_orderpriority = '2-HIGH'
            then 1
        else 0
    end) as high_line_count,
    sum(case
        when o_orderpriority <> '1-URGENT'
            and o_orderpriority <> '2-HIGH'
            then 1
        else 0
    end) as low_line_count
from
    orders,
    lineitem
where
    o_orderkey = l_orderkey
    and l_shipmode in ('MAIL', 'FOB')
    and l_commitdate < l_receiptdate
    and l_shipdate < l_commitdate
    and l_receiptdate >= to_date (date '1997-01-01')
    and l_receiptdate < to_date (date '1997-01-01' + interval '1' year)
group by
    l_shipmode
order by
    l_shipmode;
```

Query 13:

```
select
```

```
        c_count,
        count(*) as custdist
  from
        (
              select
                    c_custkey,
                    count(o_orderkey) as c_count
              from
                    customer, orders -- left outer join orders on
                    where
                          c_custkey(+) = o_custkey
                          and o_comment not like '%unusual%deposits%'
              group by
                    c_custkey
        ) -- c_orders (c_custkey, c_count)
group by
        c_count
order by
        custdist desc,
        c_count desc;
```

Query 14:

```
select
      100.00 * sum(case
            when p_type like 'PROMO%'
                  then l_extendedprice * (1 - l_discount)
            else 0
      end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
      lineitem,
      part
```

```
where
       l_partkey = p_partkey
       and l_shipdate >= to_date (date '1996-11-01')
       and l_shipdate < to_date (date '1996-11-01' + interval '1' month);
```

Query 15:

```
create view revenue0 (supplier_no, total_revenue) as
       select
              l_suppkey,
              sum(l_extendedprice * (1 - l_discount))
       from
              lineitem
       where
              l_shipdate >= to_date (date '1997-03-01')
              and l_shipdate < to_date (date '1997-03-01' + interval '3' month)
       group by
              l_suppkey;

-- 'c:\auxitools\out\appendix\stream\15.0'
select
       s_suppkey,
       s_name,
       s_address,
       s_phone,
       total_revenue
from
       supplier,
       revenue0
where
       s_suppkey = supplier_no
       and total_revenue = (
            select
                  max(total_revenue)
            from
                  revenue0
      )
order by
      s_suppkey;

drop view revenue0;

Query 16:
select
      p_brand,
      p_type,
      p_size,
      count(distinct ps_suppkey) as supplier_cnt
from
      partsupp,
      part
where
      p_partkey = ps_partkey
      and p_brand <> 'Brand#23'
      and p_type not like 'PROMO BURNISHED%'
      and p_size in (33, 9, 35, 38, 20, 13, 22, 14)
      and ps_suppkey not in (
            select
                  s_suppkey
            from
                  supplier
            where
                  s_comment like '%Customer%Complaints%'
```

```
        )
group by
        p_brand,
        p_type,
        p_size
order by
        supplier_cnt desc,
        p_brand,
        p_type,
        p_size;
```

Query 17:

```
select
        sum(l_extendedprice) / 7.0 as avg_yearly
from
        lineitem,
        part
where
        p_partkey = l_partkey
        and p_brand = 'Brand#35'
        and p_container = 'JUMBO BOX'
        and l_quantity < (
                select
                        0.2 * avg(l_quantity)
                from
                        lineitem
                where
                        l_partkey = p_partkey
        );
```

Query 18:

```
select
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice,
        sum(l_quantity)
from
        customer,
        orders,
        lineitem
where
        o_orderkey in (
                select
                        l_orderkey
                from
                        lineitem
                group by
                        l_orderkey having
                                sum(l_quantity) > 315
        )
        and c_custkey = o_custkey
        and o_orderkey = l_orderkey
        and rownum < 101
group by
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice
order by
```

```
        o_totalprice desc,
        o_orderdate;
```

Query 19:

```
select
        sum(l_extendedprice* (1 - l_discount)) as revenue
from
        lineitem,
        part
where
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#41'
                and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
                and l_quantity >= 5 and l_quantity <= 5 + 10
                and p_size between 1 and 5
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#45'
                and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
                and l_quantity >= 13 and l_quantity <= 13 + 10
                and p_size between 1 and 10
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#22'
                and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
                and l_quantity >= 20 and l_quantity <= 20 + 10
                and p_size between 1 and 15
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        );
```

Query 20:

```
select
        s_name,
        s_address
from
        supplier,
        nation
where
        s_suppkey in (
                select
                        ps_suppkey
                from
                        partsupp
                where
                        ps_partkey in (
                                select
                                        p_partkey
                                from
                                        part
                                where
                        p_name like 'cornflower%'
                )
                and ps_availqty > (
                        select
                                0.5 * sum(l_quantity)
                        from
                                lineitem
                        where
                                l_partkey = ps_partkey
                                and l_suppkey = ps_suppkey
                                and l_shipdate >= to_date (date '1996-01-01')
                                and l_shipdate < to_date (date '1996-01-01' + interval '1' year)
                )
        )
        and s_nationkey = n_nationkey
        and n_name = 'VIETNAM'
order by
        s_name;

Query 21:
select
        s_name,
        count(*) as numwait
from
        supplier,
        lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
        and o_orderkey = l1.l_orderkey
        and o_orderstatus = 'F'
        and l1.l_receiptdate > l1.l_commitdate
        and exists (
            select
                *
            from
                lineitem l2
            where
                l2.l_orderkey = l1.l_orderkey
                and l2.l_suppkey <> l1.l_suppkey
        )
        and not exists (
            select
                *
            from
                lineitem l3
            where
                l3.l_orderkey = l1.l_orderkey
                and l3.l_suppkey <> l1.l_suppkey
                and l3.l_receiptdate > l3.l_commitdate
        )
        and s_nationkey = n_nationkey
        and n_name = 'PERU'
        and rownum < 101
group by
        s_name
order by
        numwait desc,
        s_name;
```

Query 22:

```
select
    cntrycode,
    count(*) as numcust,
    sum(c_acctbal) as totacctbal
from
    (
        select
            substr(c_phone, 1, 2) as cntrycode,
            c_acctbal
        from
            customer
        where
            substr(c_phone, 1, 2) in
                ('15', '19', '16', '20', '14', '22', '10')
            and c_acctbal > (
                select
                    avg(c_acctbal)
                from
                    customer
                where
                    c_acctbal > 0.00
                    and substr(c_phone, 1, 2) in
                        ('15', '19', '16', '20', '14', '22', '10')
            )
            and not exists (
                select
                    *
                from
                    orders
                where
                    o_custkey = c_custkey
                )
        ) custsale
group by
        cntrycode
order by
        cntrycode;
```

Refresh Function1:

```
insert into orders values ( 9 , 38197 ,'O', 134840.06 ,to_date('1996-09-10','yyyy-mm-dd'),
'1-URGENT', 'Clerk#000000145', 0 , 'carefully regular requests solve furiously.
instructio' );
....
insert into lineitem values ( 5996 , 86497, 6498, 4, 32, 47471.68, 0.10, 0.00,'N','O','1997-
10-21','1997-10-05', '1997-11-14', 'NONE', 'TRUCK', 'final ideas wake foxe' );
insert into lineitem values ( 5996 , 146898, 1927, 5, 43, 83630.27, 0.07,
0.01,'N','O','1997-11-02','1997-09-27', '1997-12-02', 'TAKE BACK RETURN', 'FOB',
'slyly even multipliers haggle. care' );
```

Refresh Function2:

```
DELETE FROM lineitem WHERE l_orderkey = 1;
DELETE FROM orders WHERE o_orderkey = 1;
DELETE FROM lineitem WHERE l_orderkey = 2;
...
DELETE FROM lineitem WHERE l_orderkey = 5987;
DELETE FROM orders WHERE o_orderkey = 5987;
DELETE FROM lineitem WHERE l_orderkey = 5988;
DELETE FROM orders WHERE o_orderkey = 5988;
```

Appendix D. The Summary of Properties of TPC-H Queries.

	Nested	Data Source (The number of tables)	Group	Order	Correlated	Aggregation	conditions	Memo
Q1	N	1	Y	Y		4 sum, 3 avg,1 count	1 (1 exact matching)	
Q2	Y	5	N	Y	N	1 min(inside)	9 (8 exact matchings, 1 range) AND atoms outside 5 (5 exact matchings) AND atoms inside	Return the first 100 rows
Q3	N	3	Y	Y			6 AND atoms	Return the first 10 rows
Q4	Y	1	Y	Y	Y	1 count	3(2 range, 1 exact matchings) AND outside 2 AND(1 range, 1 exact matching) inside	
Q5	N	6	Y	Y			9 AND (7 exact matching, 2 range)	
Q6	N	1	N	N	N	1 sum	4 AND (4 range)	
Q7	Y	1	Y	Y	Y	1 sum	7 AND(inside), 2 OR(inside)	
Q8	Y	1	Y	Y	N	2 sum	10 AND	
Q9	Y	6	Y	Y	N	1 sum	7 AND (inside)	
Q10	N	4	Y	Y		1 sum	7 AND (3 range, 4 exact matchings)	Return the first 20 rows

Q11	Y	3Y	Y	Y	3 sum(2 outside, 1 inside)	3 AND(3 exact matchings)	
Q12	N	2Y	Y		2 sum	6 AND(5 range, 1 exact matching)	
Q13	Y	1Y	Y	N	2 count(1 out, 1 in)	2 AND(1 exact matching, 1 like)	left outer join(inside)
Q14	N	2N	N		2 sum	3 AND(2 range, 1 exact matching)	
Q15	N	2N	Y	N	1 sum(inside view definition)	2 AND(exact matching), 2 AND(range, inside view definition)	One source is a created view
Q16	Y	2Y	Y	N	1 count	5 AND(4 range, 1 exact matching)	
Q17	Y	2N	N	Y	1 sum(outside), 1 avg(inside)	4 AND(outside, 3 exact matchings, 1 rang), 1 AND(inside)	
Q18	Y	3Y	Y	Y	2 sum(1 outside, 1 inside)	4 AND(2 exact matchings, 2 range)	Return the first 100 rows
Q19	N	2N	N		1 sum	3 OR(outside),7 AND(3 exact matchings, 4 range) each	
Q20	Y(3 levels)	2N	Y	Y	1 sum(inside)	3 AND(outside,1 range, 2 exact matchings), 2 AND(inside, 2 range), 1 AND(third level), 4 AND(third level, 2 range, 2 exact matchings)	
Q21	Y	4Y	Y	Y	1 count	9 AND(outside,5 exact matchings, 4 range), 2 AND(inside, 1 range, 1 exact matching), 3 AND(inside,2 range, 1 exact matching)	Return the first 100 rows

					1 sum(outside), 1	
					avg(outside),1	3 AND(inside,3 range), 2
	Y(3				count(inside),3	AND(level 3, range), 1 AND(level
Q22	levels)	1Y	Y	Y	substr()	3, 1 exact matching)