Introducing Skew into the TPC-H Benchmark

Conference Paper *in* Lecture Notes in Computer Science · August 2011

DOI: 10.1007/978-3-642-32627-1_10

CITATIONS
16

READS
1,255

2 authors:

Ahmad Ghazal
PingCAP
44 PUBLICATIONS **629** CITATIONS
SEE PROFILE

Alain Crolotte
Teradata
41 PUBLICATIONS **637** CITATIONS
SEE PROFILE

Benchmarking Using Basic DBMS Operations

Alain Crolotte, Ahmad Ghazal

Teradata Corporation, 100 N Sepulveda Blvd.
El Segundo, Ca. 90045
{alain.crolotte, ahmad,ghazal}@teradata.com

Abstract. The TPC-H benchmark proved to be successful in the decision support area. Many commercial database vendors and their related hardware vendors used these benchmarks to show the superiority and competitive edge of their products. However, over time, the TPC-H became less representative of industry trends as vendors keep tuning their database to this benchmark-specific workload. In this paper, we present XMarq, a simple benchmark framework that can be used to compare various software/hardware combinations. Our benchmark model is currently composed of 25 queries that measure the performance of basic operations such as scans, aggregations, joins and index access. This benchmark model is based on the TPC-H data model due to its maturity and well-understood data generation capability. We also propose metrics to evaluate single-system performance and compare two systems. Finally we illustrate the effectiveness of this model by showing experimental results comparing two systems under different conditions.

1 Introduction

In the DSS area the TPC-D then TPC-H benchmarks [1] were quite successful as demonstrated by the large number of companies that ran them. The 3rd normal form schema on which it is based is easy to understand and rich enough to perform interesting experiments. The main advantage provided by this benchmark has been the data generation utility (known as dbgen) that guaranteed repeatability on all platforms. The benchmark is now showing its age and its usefulness to drive the development of new database features is severely diminished. This is particular true in the area of query optimization as the query set is fixed and at this point in time too well understood. It still remains today a de facto standard and its good features are still very useful. In particular, it can be used as a basis for developing interesting workloads that are related but different from TPC benchmarks are used extensively in research by universities [2] or corporations [3, 9,10].

There are several issues associated with the TPC-H benchmark aside from its age. The main problem is the set of rules and restrictions that govern its execution. New database features often need to be adapted to be transparent or hidden to allow publication. As a result of this lack of transparency it is often difficult to assess what techniques are actually used in a published result and how effective these techniques are. A side effect of this state of affairs is an inherent unfairness in comparing results. While shortly after the inception of the TPC-D benchmark it actually made sense for

vendors to partly assess new hardware or a new software releases with TPC-D metrics, it is no longer the case. It was still possible after the inception of the TPC-H benchmark as the few changes over TPC-D kept the interest alive for a while. It is still somewhat possible to get a glimpse at a new architecture by looking at the individual times in the power test for queries 1 and 6 since usually, they are respectively CPU and I/O bound.

In this paper, we propose a very simple benchmark approach based on the TPC-H workload allowing a global and fair evaluation of new features. This approach could be of interest especially for hardware vendors as well. Hardware vendors have many tools at their disposal to determine the raw power of their equipment but these approaches usually do not take into account the database layer. What we propose here instead is an approach that takes the database layer into account.

Many of the database optimization techniques designed for performance actually avoid utilizing the hardware. A good example is materialized structures such as materialized views, join indexes or summary tables since they practically bypass the I/O subsystem. Our benchmark is designed in such a way to exercise fully the hardware through basic database functions such as scans, joins and aggregations. Although the benchmark presented can be used "as is" the approach can also be adapted to define more involved queries that would lend themselves to comparisons between an optimized and a non-optimized workload.

There were several attempts to define new benchmarks around TPC-D and TPC-H Some of these attempts modified the data model or added a missing element or added new queries. In general it would be easy to introduce skew with the customer or supplier distributions as was done in [6] a private benchmark built on top of TPC-D that introduced skew. Another example is [7] in which the authors present a star schema around the TPC-H benchmark. We follow a similar approach without modifying the TPC-H schema nor the data generation utility.

In section 2 we present XMarq, our benchmark proposal. The discussion is broken into different database classes such as scans and aggregations. This includes a description of the queries to be run in each class and the rules under which these queries should be run. In section 3 we provide benchmark results and propose a methodology to evaluate those results absolutely and comparatively. In section 4 we analyze the XMarq benchmark in light of ideal conditions proposed at the 2009 TPC Conference [8]. The actual SQL associated with the queries is provided in the Appendix.

2 Benchmark Specifications

The TPC-H data model is a retail model fully described in [1]. TPC-H consists of 8 tables with history over 8 years. We are presenting queries using mainly ORDERS (primary key (PK) =o_orderkey) containing the orders, LINEITEM (PK=l_orderkey, l_linenumber) containing the line items associated with these ORDERS and Customer (PK=c_custkey) contains customers placing orders. Customers have a nation they belong to (FK=c_nationkey). There is a PART table (PK=p_partkey) and LINEITEM has a FK=l_partkey.

Even though the XMarq benchmark can be defined for any scale factor supported by dbgen, not just the official TPC-H scale, we will define it here with scale factor 1000 for the sake of simplicity. XMarq is designed to be run right after the database has been materialized with dbgen. Because the benchmark must represent an ad hoc environment, only primary and foreign key columns may be indexed, and no optimizer hints or extra pre-calculated structures may be used. Also pre-staging of the data including indexes is disallowed and so is vertical partitioning.. Only raw system performance through simple database queries is to be measured not the sophistication of the optimizer. Similarly, no techniques to limit the number of rows returned are allowed

The benchmark is organized into five query groups: scans, aggregations, joins, CPU-intensive and indexed. The main idea is to utilize known properties of the TPC-H data to define basic queries. For a full table scan for instance we can use a constraint that is known to be impossible to satisfy. Examples of this can be l_linenumber < 0 since we know that that field is always between 1 and 7. In the sequel we go trough the proposed queries. Queries are named with a 2-letter code in which the first letter identifies the query group (e.g. S to scans, A for aggregation).

2.1 Scan Queries

ST – LARGE TABLE SCAN

This query consists of scanning the LINEITEM table while returning no rows This is accomplished by selecting all the columns based on a constraint that is never satisfied namely l_linenumber < 0. For this query we assume that there are no statistics or index on l_linenumber.

SI – MATCHING INSERT/SELECT

This query populates PARTX a copy of the PART table distributed by p_partkey with an insert/select. Matching merely means that the original table and its copy use the same DDL including the distribution field, in this case p_partkey.

SN – NON MATCHING INSERT/SELECT

This query populates PARTX with DDL similar to PART but distributed differently, in this case, by the non-unique combination of p_size, p_brand and p_container instead of p_partkey. This will likely cause more data movement than the matching insert/select in most products.

SU – SCAN UPDATE 4% OF THE ROWS

This query utilizes the fact that column p_brand in PARTX has only 25 distinct vales so that, due to the uniform distribution of data produced by dbgen, any specific value of p_brand will be represented in about 4% of the rows.

SP – SCAN UPDATE 20% OF THE ROWS

This query utilizes the fact that p_mfgr has only 5 distinct values. As a result a particular value of p_mfgr will involve about 20% of the rows. At scale factor 1000 exactly 40,006,935 rows out of 200 millions are updated.

2.2 Aggregation Queries

AR – ROW COUNT

This query does a simple count of the rows in the largest table in the database, LINEITEM. The number of rows returned is well-documented in the TPC-H spec.

AD – DISTINCT COUNT

This query counts the distinct values of the column l_quantity in the LINEITEM table. A single row with a value of 50 should be returned at all scale factors and l_quantity must not be indexed for this query.

AS – 15-GROUP AGGREGATE

This query utilizes ORDERS and the fact that the combination o_ordertatus, o_orderpriority has only 15 distinct combinations at all scale factors.

AM – THOUSAND GROUP AGGREGATE

This query uses LINEITEM and l_receipdate that has only a limited number of values (2555 at scale factor 1000). While l_shipdate is more predictable (exactly 2406 distinct values at all scale factors) this field plays too central of a role in the TPC-H queries. No index should be placed on l_receiptdate for this query.

AL – HUNDRED THOUSAND GROUP AGGREGATE

This query is meant to build over 100000 aggregate groups. By using the first 15 characters of o_comment one can build exactly 111517 groups at scale factor 1000. To further limit the number of rows actually retrieved we added a limit on o_totalprice.

2.3 Join Queries

JI – IN-PLACE JOIN

In this query we join ORDERS and LINEITEM on the key common to both tables without constraints while performing a calculation ensuring that the join is performed but only one row is returned.

JF – PK/FK JOIN

This query joins PART and LINEITEM on partkey which is the PK for PART and the FK for LINEITEM while performing an operation involving columns in both tables. No index on l_partkey is allowed for this query.

JA – AD-HOC JOIN

This query joins PART and LINEITEM on unrelated columns (p_partkey and l_suppkey) while performing a sum so that only one row is returned. Because of the fact that the join columns are sequential integers there will be some matching.

JL – LARGE/SMALL JOIN

This query joins CUSTOMER and NATION on nationkey while performing a group by operation. This is also an FK/PK join but the salient feature here is the size discrepancy between the tables since NATION has only 25 rows.

JX – EXCLUSION JOIN

This query calculates the total account balance for the one-third of customers without orders. For those customers the CUSTOMER rows do not have a customer key entry in the ORDER table. The inner query selects customers that do have orders so that by leaving these out we get the customers that do not have orders.

2.4 CPU Intensive Queries

CR – ROLLUP REPORT

This query covers the group by operator. To accomplish that, the query is applied on a single table with a simple predicate. The query involves 7 columns and 12 aggregations to make it CPU intensive.

CF – FLOATS & DATES

This query maximizes CPU activity while minimizing disk or interconnect overhead. Floating point conversion is done multiple times for each of the rows, as well as repetitive complex date conversions. Because of its emphasis on repetitive CPU-intensive operations, it will highlight products that offer machine readable evaluation code, as opposed to the more common interpretive code. It will also tend to favor products that are capable of caching and reusing repetitive instructions, and are coded to perform such activities efficiently. To minimize I/O, a predicate is used to filter out all but one row.

2.5 Index Queries

IP – PRIMARY RANGE SEARCH

This query measures the ability of database system to select from a table based on a range of values applied to the table's primary (or clustering) index. In this case the range constraint is on the same column as the partitioning key (clustering, primary index). It is designed to highlight capabilities such as value-ordered or value-sensitive indexes.

A second constraint has been added to the WHERE clause, based on P_RETAILPRICE. Its purpose is to prevent large numbers of rows from being returned. An index on this constraint column is disallowed, so as to make sure that the primary index is the only index available for data access.

This primary key range search query is different from the next query, I2, which uses secondary index access, because many database systems organize these different types of indexes differently, with differing performance characteristics. Often a primary index will determine the physical location or the physical sequencing of a row within a table. Some database systems, however, do not differentiate between partitioning (or primary) indexes and secondary indexes. This query returns 44 rows at Scale Factor 1000.

IR – SECONDARY RANGE SEARCH

In this query, the range constraint column is not the same column as the partitioning key, as was the case with query IP above. Secondary index access usually requires traversing a secondary structure which points to the location of the physical rows required. This physical structure may or may not be sequenced, or may or may not be efficient to navigate. This query attempts to measure those differences. This query is expected to favor products that have value-ordered secondary indexes, or cost-based optimizers intelligent enough to consider scanning an index structure. An index must be placed on l_shipdate for this query.

IL – LIKE OPERATOR

This query uses the text string search capabilities of the LIKE clause, searching for all rows that have a value in their O_CLERK column that begin with that string. It then returns each distinct value for the O_CLERK column satisfies the LIKE clause condition.

The number of distinct clerks in the Order table is determined at data generation time by multiplying the scale factor by 1000. For Scale Factor 10 there will be 10,000 Clerks randomly assigned to ORDERS. As there are 9 significant digits making up O_CLERK, and 7 of them are to the left of the percent sign (with 2 remaining, we know there are a possible 100 Clerks that meet the criteria of this query.

The query returns 100 rows at all scale factors. An index must be placed on o_clerk for this query.

IB – BETWEEN OPERATOR

This query uses a third type of constraint with an index: the between clause. This query requests from the LINEITEM table only rows that have a ship date of a single month in 1992. After accessing those particular LINEITEM rows, it returns only the distinct quantities associated with the rows l_quantity as a column is a random value from 1 to 50. As there are approximately 7 years of data at all volume points, one month constitutes 1/84, or 1.2% of the LINEITEM rows which are processed by this query. Similarly to query IR an index must be placed on l_shipdate for this query.

II – MULTIPLE INDEX ACCESS

While previous index access queries tested either primary or secondary index strategies, this query combines them both.

This query utilizes the OR operator between two constraints: one constraint is on a non-unique index on the Clerk column (as we mentioned above, there are 10,000 Clerks at Scale Factor 10, about 1500 orders have the same Clerk) and the second is on a unique index o_orderkey. There are 1462 actual Clerks with this Clerk_ID in the ORDERS table.

Since this is OR condition, a row will qualify for inclusion in the answer set if either side of the OR is satisfied, so some database optimizer decisions will need to accommodate this possible complexity. This query tests the flexibility and diversity of the optimizer in a controlled environment with mandatory indexes.

Some database systems might be able to satisfy this query by only traversing the index structures, and not actually doing physical I/O on the base table itself. This is a possibility, in part, because of the query's select list. The query only returns a count of how many rows pass either of the selection criteria, and does not request data from the base table itself. A single row with a count of 1462 is returned. An multiple column index on o_clerk and o_orderkey must be exist for this query.

IC – COUNT BY INDEX

This query lists each distinct Discount in the largest table in the database, and counts how many rows are associated with each of the Discounts it finds. This is a common activity useful to validate the database information, or to find out how your customer base is skewed geographically.

This query is both a simple aggregation query and a query that demonstrates flexibility in the use of the secondary index structure. Assuming an index is placed on the l_discount column, it may be possible to accommodate this request without having to read the base table. This greatly reduces I/O overhead in this case, because this query can only be satisfied by looking at information stored in the index structure and simply counting rows from there. There are 11 different discount values in l_discount. Therefore this query will return 11 rows for all Scale Factors.

IM – MULTI-COLUMN INDEX -- LEADING VALUE ONLY

It is not uncommon that in index created for a broader purpose may be useful for queries other than for which it was intended. In this query we have a 2-column index on p_type and p_size but the condition is on p_type only.

This query tests the ability of the index to respond to broader needs than anticipated. There are 150 different p_type values in the PART table. This query determines what the average retailprice is for all the Parts of that type. A single row is returned by this query

IT – MULTI-COLUMN INDEX -- TRAILING VALUE ONLY

A somewhat related need is measured by this query, which provides a value for the second, or trailing value in a 2-column index. There are 50 distinct values in p_size, of which only one is selected here. This query, similar to the one above, averages the retailprice in all qualifying parts that have the size of 21. Instead of accessing the

leading field of the 2-column index p_size, p_type this query accesse the trailing field of the index. Only a single row is returned.

3 Experimental Results

In order to illustrate the use of the benchmark in a real-life case we ran the benchmark with the same version of Teradata on two different platforms. The columns marked system A and system B portray actual performance numbers. In those columns the summary per category is an arithmetic mean. For instance, for scans, what we have in row SCANS is the arithmetic mean of all scan type performance for system A and system B respectively. The numbers in column A/B are ratios and the method to aggregate those numbers is the geometric mean (see [4] and [5] for an explanation of the definition and proper use of arithmetic and geometric means). For instance, for scans, the score of 4.8 is the geometric mean of the five numbers above. The overall scores follow the same logic. The overall score for system A and system B are the arithmetic means of their individual raw performance at the individual query level while the overall ratio score is the geometric mean of all ratios at query level.

QUERIES/CATEGORIES	System A	System B	A/B
ST -- LARGE TABLE SCAN	95.9	18.6	5.1
SI -- MATCHING INSERT/SELECT	48.0	8.7	5.5
SN -- NON-MATCHING INSERT/SELECT	174.6	25.0	7.0
SU -- SCAN/UPATE 4% OF THE ROWS	191.5	58.4	3.3
SP -- SCAN/UPATE 20% OF THE ROWS	906.3	244.1	3.7
SCANS	**283.2**	**71.0**	**4.8**
AR -- ROW COUNT	0.8	0.3	3.0
AD -- COUNT DISTINCT VALUES	112.3	29.8	3.8
AS -- 15-GROUP AGGREGATE	51.5	14.4	3.6
AM -- THOUSAND-GROUP AGGREGATE	30.2	9.7	3.1
AL -- HUNDRED-THOUSAND GROUP AGGREGATION	61.1	16.6	3.7
AGGREGATIONS	**51.1**	**14.2**	**3.4**
JI -- IN PLACE JOIN	372.8	123.3	3.0
JF -- FOREIGN KEY TO PRIMARY KEY JOIN	2441.5	340.5	7.2
JA -- AD HOC JOIN	891.9	191.9	4.6
JL -- LARGE/SMALL JOIN	0.1	0.0	11.0
JX -- EXCLUSION JOIN	21.3	19.6	1.1
JOINS	**745.5**	**135.1**	**3.2**
CR -- ROLLUP REPORT	312.3	75.2	4.2

CF -- FLOATS & DATES	96.4	16.5	5.9
CPU INTENSIVE	**204.3**	**45.8**	**4.9**
IP -- PRIMARY RANGE SEARCH	3.1	1.0	3.3
IR -- SECONDARY RANGE SEARCH	10.7	0.3	42.7
IL -- LIKE OPERATOR	29.0	7.7	3.8
IB -- BETWEEN OPERATOR	14.1	0.6	22.1
II -- MULTIPLE INDEX ACCESS	29.0	7.5	3.9
IC -- COUNT BY INDEX	6.5	1.8	3.6
IM -- MULTI-COLUMN INDEX -- LEADING VALUE ONLY	1.9	0.3	5.6
IT -- MULTI-COLUMN INDEX -- TRAILING VALUE ONLY	1.7	0.2	7.3
INDEXED OPERATIONS	**12.0**	**2.4**	**7.1**
overall scores	***236.2***	***48.5***	***4.9***

4 Conclusions

In [8] Huppler proposed five criteria for a good benchmark. The most important one is "relevance" and the other four are "repeatability", "fairness", "verifiability" and "being economical". Huppler also made the point that there should be a balance or tradeoff between "relevance" and the other four criteria. We believe that XMarq meets all five criteria. In terms of relevance XMarq scores very high since it focuses on the most common paths in the database code. It also can be run on any hardware supporting the database under test. XMarq is repeatable since it is based on the well-established TPC-H schema, the dbgen code that produces repeatable data and very simple SQL, actually a subset of the TPC-H SQL. These qualities make XMarq easily repeatable and portable. XMarq is also fair since it focuses on basic SQL operations and does not favor exotic optimizer features that are specific to certain database products. Verifying XMarq results is also straightforward. The metric proposed is actually better than the TPC-H power metric since it uses the arithmetic mean while the geometric mean is used appropriately for ratios i.e. normalized data. Other measurements can be performed e.g. CPU, I/O or network consumption. XMarq is meant to measure the basic performance of existing hardware and software and therefore does not require any additional development. Finally, the overall cost is much lower than TPC-H since the process of running and auditing the benchmark is straightforward. It would be very interesting for hardware vendors to correlate TPC-H and XMarq results for a given database.

Appendix

ST – LARGE TABLE SCAN
SELECT * FROM LINEITEM WHERE L_LINENUMBER < 0;

SI – MATCHING INSERT/SELECT
INSERT INTO PARTX SELECT * FROM PART;

SN – NON MATCHING INSERT/SELECT
INSERT INTO PARTX SELECT * FROM PART;

SP – SCAN UPDATE 4% OF THE ROWS
UPDATE PARTX SET P_RETAILPRICE = (P_RETAILPRICE + 1) WHERE P_BRAND = 'Brand#23';

SP – SCAN UPDATE 20% OF THE ROWS
UPDATE PARTX SET P_RETAILPRICE = (P_RETAILPRICE + 1) WHERE P_MFGR = 'Manufacturer#5';

AR – ROW COUNT
SELECT COUNT(*) FROM LINEITEM;

AD – DISTINCT COUNT
SELECT COUNT (DISTINCT L_QUANTITY) FROM LINEITEM;

AS – 15-GROUP AGGREGATE
SELECT O_ORDERSTATUS, O_ORDERPRIORITY,
AVERAGE (O_TOTALPRICE FROM ORDERS GROUP BY 1, 2;

AM – THOUSAND-GROUP AGGREGATE
SELECT L_RECEIPTDATE, COUNT (*) FROM LINEITEM
GROUP BY 1 ORDER BY 1;

AL – HUNDRED THOUSAND GROUP AGGREGATE
SELECT SUBSTRING (O_COMMENT,1,15), ,COUNT(*) FROM ORDERS
GROUP BY 1;

JI – IN-PLACE JOIN
SELECT AVERAGE (L_QUANTITY) FROM LINEITEM, ORDERS
WHERE L_ORDERKEY=O_ORDERKEY;

JF – PK/FK JOIN
SELECT AVERAGE (P_RETAILPRICE*L_QUANTITY)
FROM PART, LINEITEM WHERE P_PARTKEY=L_PARTKEY;

JA – AD-HOC JOIN
SELECT SUM(L_QUANTITY) FROM PART, LINEITEM
WHERE P_PARTKEY=L_SUPPKEY;

JL – LARGE/SMALL JOIN

SELECT N_NAME, AVERAGE(C_ACCTBAL) FROM CUSTOMER, NATION WHERE C_NATIONKEY=N_NATIONKEY GROUP BY N_NAME;

CR – ROLLUP REPORT

```
SELECT L_RETURNFLAG,
L_LINESTATUS,
L_SHIPMODE,
SUBSTRING (L_SHIPINSTRUCT, 1, 1),
SUBSTRING (L_LINESTATUS, 1, 1),
((L_QUANTITY - L_LINENUMBER) + (L_LINENUMBER - L_QUANTITY)),
(L_EXTENDEDPRICE - L_EXTENDEDPRICE),
SUM ((1 + L_TAX) * L_EXTENDEDPRICE),
SUM ((1 - L_DISCOUNT) * L_EXTENDEDPRICE),
SUM (L_DISCOUNT / 3),
SUM (L_EXTENDEDPRICE * (1 - L_DISCOUNT) * (1 + L_TAX)),
SUM (L_EXTENDEDPRICE - ((1 - L_DISCOUNT) * L_EXTENDEDPRICE)),
SUM (DATE - L_SHIPDATE + 5),
SUM (L_SHIPDATE - L_COMMITDATE),
SUM (L_RECEIPTDATE - L_SHIPDATE),
SUM (L_LINENUMBER + 15 - 14),
SUM (L_EXTENDEDPRICE / (10 - L_TAX)),
SUM ((L_QUANTITY * 2) / (L_LINENUMBER * 3)),
COUNT (*)
FROM LINEITEM
WHERE L_LINENUMBER GT 2
GROUP BY
 L_RETURNFLAG,
 L_LINESTATUS,
 L_SHIPMODE,
 SUBSTRING (L_SHIPINSTRUCT,1,1),
 SUBSTRING (L_LINESTATUS,1,1),
  ((L_QUANTITY - L_LINENUMBER) + (L_LINENUMBER - L_QUANTITY)),
  (L_EXTENDEDPRICE - L_EXTENDEDPRICE);
```

CF - FLOATS & DATES

```
SELECT COUN T(*) FROM LINEITEM
WHERE (L_QUANTITY = 1.1E4
OR L_QUANTITY = 2.1E4
OR L_QUANTITY = 3.1E4
OR L_QUANTITY = 4.1E4
OR L_QUANTITY = 5.1E4
OR L_QUANTITY = 6.1E4
OR L_QUANTITY = 7.1E4
OR L_QUANTITY = 8.1E4
OR L_QUANTITY = 9.1E4
OR L_QUANTITY = 50)
AND (DATE - L_SHIPDATE) GT 0
```

```
AND (L_COMMITDATE + 5) LT (L_RECEIPTDATE + 5)
AND (L_SHIPDATE + 20) LT (L_COMMITDATE + 20);
```

IP – PRIMARY RANGE SEARCH

```
SELECT P_NAME, P_RETAILPRICE FROM PART
WHERE P_PARTKEY LT 50000
AND    P_RETAILPRICE LT 909.00;
```

IR – SECONDARY RANGE SEARCH

```
SELECT L_ORDERKEY, L_LINENUMBER
FROM LINEITEM
WHERE L_SHIPDATE LT 981200;
```

IL – LIKE OPERATOR

```
SELECT DISTINCT O_CLERK
FROM ORDERS
WHERE O_CLERK LIKE 'Clerk#0000067%';
```

IB – BETWEEN OPERATOR

```
SELECT DISTINCT L_QUANTITY
FROM LINEITEM
WHERE L_SHIPDATE BETWEEN 930301 AND 930331;
```

II – MULTIPLE INDEX ACCESS

```
SELECT COUNT (*) FROM ORDERS
WHERE O_CLERK = 'CLERK#000006700' OR O_ORDERKEY = 50500;
```

IC – COUNT BY INDEX

```
SELECT L_DISCOUNT, COUNT (*) FROM LINEITEM
GROUP BY L_DISCOUNT;
```

IM – MULTI-COLUMN INDEX -- LEADING VALUE ONLY

```
SELECT AVERAGE (P_RETAILPRICE) FROM PART
WHERE P_TYPE = 'SMALL PLATED BRASS';
```

IT – MULTI-COLUMN INDEX -- TRAILING VALUE ONLY

```
SELECT AVERAGE (P_RETAILPRICE) FROM PART
WHERE P_SIZE = 21;
```

References

1. TPC BENCHMARK D / H (Decision Support), Transaction Processing Council. www.tpc.org
2. Kun Gao and Ippokratis Pandis "Implementation of TPC-H and TPC-C toolkits", CS and ECE Departments, Carnegie Mellon University http://www.cs.cmu.edu/~ipandis/courses/15823/project_final_paper.pdf
3. Ghazal, A., Seid, D., Ramesh, B., Crolotte, A., Koppuravuri, M., and G, V. 2009. Dynamic plan generation for parameterized queries. In *Proceedings of the 35th SIGMOD international Conference on Management of Data* (Providence, Rhode Island, USA, June 29 - July 02, 2009). SIGMOD '09. 909-916.
4. Crolotte, A. "Issues in Metric Selection and the TPC-D Single Stream Power" www.tpc.org
5. Crolotte, A. "Issues in Benchmark Metric Selection" In Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France.
6. THE DATA WAREHOUSE CHALLENGE, specification document revision 1.4.c, Data Challenge Inc., 9 April 1998.
7. Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen and Stephen Revilak "The Star Schema Benchmark and Augmented Fact Table Indexing" In Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France.
8. Huppler, K. "The Art of Building a Good Benchmark". In Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France.
9. Carrie Ballinger, "Relevance of the TPC-D Benchmark Queries : The Questions You Ask Every Day", NCR Parallel Systems http://www.tpc.org/information/other/articles/TPCDart_0197.asp
10. Meikel Poss, Chris Floyd, "New TPC Benchmarks for Decision Support and Web Commerce". SIGMOD Record, 2000: 64-71