The Set Query Benchmark

Patrick E. O'Neil
Department of Mathematics and Computer Science
University of Massachusetts at Boston
Boston, MA 02125-3393

1. INTRODUCTION TO THE BENCHMARK

Many of the application systems being designed today, variously known as marketing information, decision support, and management reporting systems, aim to exploit the strategic value of operational data of a commercial enterprise. These applications depart from the row-at-a-time update transaction model of the DebitCredit [1] and TPC benchmarks, and are almost wholly dependent for their performance on what we name "set queries", queries which need to refer to data from a potentially large set of table rows for an answer. The Set Query benchmark chooses a list of "basic" set queries from a review of three major types of strategic data applications: document search, direct marketing, and decision support. In Section 1 of what follows, the data and queries used in the Set Query benchmark are explained and motivated. In Section 2, benchmark results are presented for two leading database products used in large scale operations: IBM's DB2 and Computer Corporation of America's (CCA's) MODEL 204. Surprisingly large performance differences, factors of ten or more for some queries, are observed with respect to I/O, CPU and elapsed time, emphasizing the critical value of benchmarks in this area. In Section 3, a detailed explanation is given of how to generate the data and run the benchmark on an independent platform.

1.1. Rationale

While DebitCredit and TPC do an excellent job of measuring the single-record update paradigm of OLTP, these are not appropriate benchmarks for some of the new systems being implemented by companies in the area of strategic value data applications, an area with various names such as: Marketing Information Systems, Decision Support Systems, Management Reporting, or Direct Marketing. A growing number of executives are thinking in these terms: "We have our operational data under control, the core of our business where we keep track of our orders. Now we want to set up a system to gain insight into the strategic value of the data. Who are our customers and how should we segment our markets? Which of our customers return the most profit? What are product purchasing trends? Can we use our knowledge of our customers' needs to advise them, supply more of our products and simultaneously improve our perceived value to them, before our competitors beat us to it?"

Database functions which support applications of this kind generally use database **queries** (as opposed to **updates**) in their implementation. More specifically, they use **set queries**, meaning that they take into account data from numerous table rows at once in each question. Since the DebitCredit and TPC benchmarks deal only with row-at-a-time updates, a very different type of processing, it fails to match these needs. Experience has shown that Information Systems Managers looking for hard performance numbers in this area are likely to factor in DebitCredit results during product selection, on the theory that some data is better than none. The importance of set query functionality has not been properly articulated up to now, and even sophisticated users have not been sensitive to the distinction. However, measurements from this benchmark show that computer resource usage by set queries can be extremely high, with surprising variation between different products, so that performance can be a critical issue. The Set Query benchmark presented here has been created to aid decision makers who require performance data relevant to strategic value data applications.

Table 1.1. An example of a Set Query used in a strategic value data application. The following query might be used to generate a mailing list for announcements of a new women's racquet sports magazine:

```
SELECT NAME, ADDR FROM PROSPECTS
WHERE SEX = 'F'
AND FAMILYEARN > 40000
AND ZIPCODE BETWEEN 02100 AND 12200
AND EDUC = "COLLEGE"
AND HOBBY IN ( "TENNIS", "RACQUETBALL");
```

1.2 Features of The Set Query Benchmark

The Set Query benchmark has four key characteristics, explained below:

- Portability
- Functional Coverage
- Selectivity Coverage
- Scalability

Portability The queries for the benchmark are specified in SQL which is available on most systems, although different query language forms are permitted which give identical results. The data used is representative of real applications, but is artificially generated using a portable random number algorithm detailed in Section 3; this allows investigators to create the database on any system. Different platforms can then be compared on a price/performance basis.

Functional Coverage As discussed later in detail, the benchmark queries are chosen to span the tasks performed by an important set of strategic value data applications. Prospective users should be able to derive a rating for the particular subset they expect to use most in the system they are planning.

<u>Selectivity Coverage</u> Selectivity applies to a clause of a query, and indicates the proportion of rows of the database to be selected: we say that "`SEX = 'M'`" where half the rows are returned, has very low selectivity, while "`SOCSECNO = '028343179'`" where a single row is returned has very high selectivity. The selectivity usually has tremendous effect on the performance of a query, so the Set Query benchmark specifies measurements for a spectrum of selectivity values within each query type. Prospective users can then concentrate on measurements in the range of selectivity values they expect to encounter.

<u>Scalability</u> The database has a single table, known as the BENCH table, which contains an integer multiple of 1 million rows of 200 bytes each. The default size of one million rows is large enough to be realistic and also to highlight a number of crucial performance issues: set query performance differences are magnified on large databases, and this is significant as commercial databases grow rapidly with time.

1.2.1 Definition of the BENCH Table

To offer the desired variety of selectivity, the BENCH table has 13 indexed columns; in the default case of a one million row table, these columns are named: `KSEQ, K500K, K250K, K100K, K40K, K10K, K1K, K100, K25, K10, K5, K4` and `K2`. Twelve of these columns are unordered (randomly generated), and vary in cardinality (number of distinct values) from 2 to 500,000. Each such column has integer values ranging from 1 to its cardinality, which is reflected in the column name. Thus `K2` has two values, 1 and 2; `K4` has four values, 1,2,3,4; `K5` has five values . . .; up to `K500K`, which has 500,000 values. The remaining indexed column, called `KSEQ`, is a clustered primary key, with values 1, 2, . . ., 1,000,000, the values occurring in the same order that the records are loaded. A Pseudocode program for generating these column values is given in Table 3.1 of Section 3.

Table 1.2. First 10 rows of the BENCH database

KSEQ	K500K	K250K	K100K	K40K	K10K	K1K	K100	K25	K10	K5	K4	K2
1	16808	225250	50074	23659	8931	273	45	4	4	5	1	2
2	484493	243043	7988	2504	2328	730	41	13	4	5	2	2
3	129561	70934	93100	279	1817	336	98	2	3	3	3	2
4	80980	129150	36580	38822	1968	673	94	12	6	1	1	2
5	140195	186358	35002	1154	6709	945	69	16	5	2	3	2
6	227723	204667	28550	38025	7802	854	78	9	9	4	3	2
7	28636	158014	23866	29815	9064	537	26	20	6	5	2	2
8	46518	184196	30106	10405	9452	299	89	24	6	3	1	1
9	436717	130338	54439	13145	1502	898	72	4	8	4	2	2
10	222295	227905	21610	26232	9746	176	36	24	3	5	1	1

In addition to these indexed columns, there are also a number of character columns, `S1` (length 8), `S2` through `S8` (length 20 each) which fill out the row to a length of 200 bytes. These character strings can be generated with the identical values such as, "12345678900987654321", since they are never used in retrieval queries, a reflection of the fact that unindexed retrieval in large, low update tables is extremely inadvisable. Note that certain database systems perform compression on the data stored -- we we do not want to choose values for s1 through s8 which result in compression to less than 200 bytes of data.

Where a BENCH table of more than 1 million rows is created, the three highest cardinality columns are either renamed or reinterpreted in a consistent way. For example, in a table with 10 million rows, the columns would be: `KSEQ, K5M, K2500K, K100K, K40K, K10K, K1K, K100, K25, K10, K5, K4` and `K2.` The column `KSEQ` would then take on the values 1, 2, . . ., 10,000,000 and the next two columns, formerly `K500K` and `K250K`, would be renamed to `K5M` and `K2500K`, so as to have 10 times as many values as in the 1 million row case. The purpose is to achieve consistent scaling in the joins of Query Q6, explained below. The renamed columns would replace the default column names throughout this document wherever they appear.

1.3 Achieving Functional Coverage

In determining the set of queries to include in this benchmark, the first aim was to reflect set query activity in common commercial use. Experience in the field suggested three general strategic value data applications which are detailed below: document search, direct marketing, and decision support/management reporting. After describing the work done in these applications, a number of query types were chosen to support the activity, with a surprising amount of confirming overlap discovered at the lowest level of analysis. In preparation for publishing a Set Query article in Datamation [6], five companies with state of the art strategic information applications were contacted; each company was asked to list the queries from the Set Query benchmark which made up a large portion of their work, and were also asked if they could identify any query type not on the list which they found important. As a result of this survey, one minor addition was made to the query set (to include a Boolean NOT on some equal match condition, Query Q2B). Otherwise, respondents generally felt that the query set chosen was quite representative of the use they experienced. We now describe the three strategic value data applications studied and how these led to the selection of query types for our benchmark; a somewhat more detailed explanation of some queries, together with examples of the reporting form, is given in the results of Section 2.

1.3.1 Document Search. In this application, the user begins by specifying one or more qualities desired in a set of retrieved rows; the application `COUNT`s the number of rows thus selected and returns this number to the user. Usually the user then adds new qualities to the ones specified, and once again requests a count of rows so qualified. The ultimate object is to winnow down to a small number of documents (from 1 up to a few hundred) which deserve closer scrutiny, at which point more detail from the record is printed out.

It is important to realize that the "documents" can represent any information. Well known online database services, such as DIALOG and LEXIS, structure access to journal abstracts. For an application used in the field, we surveyed a company known as Petroleum Information, or PI, of The Dun & Bradstreet Corporation, which provides a retrieval application and oil well data to most companies in the Oil Industry. There are two million oil wells involved, with 20 Gigabytes of data on drilling permits, test data, drilling costs, production volumes, and so on. The method explained above of successive refinement of the set of qualities desired has become a standard for oil industry analysts, and the activity is predominant in PI applications.

The first thing we notice about this application is that it is much more common to COUNT than to actually retrieve data from a set of rows; a series of COUNT operations are performed to get to the point where a selected set of rows is printed out. This may be at variance with common perception of SQL which emphasize queries such as SELECT *... rather than SELECT COUNT(*)..., but it is a pattern we see over and over. Aggregate functions, most significantly the COUNT function, provide an overview of the data which lets us grasp its significance.

Consideration of the document search application led us to specify three general query types, which are:

(i) A COUNT of records with a single exact match condition, known as query Q1:

```
   Q1:   SELECT COUNT(*) FROM BENCH
         WHERE KN = 2;
```

(Here and in later queries, KN stands for any member of a set of columns. Here, KN ε {KSEQ, K100K,..., K4, K2}. The measurements are reported separately for each of these cases.)

(ii) A COUNT of records from a conjunction of two exact match condition, query Q2A:

```
   Q2A:    SELECT COUNT(*) FROM BENCH
           WHERE K2 = 2 AND KN = 3;
```

For each KN ε {KSEQ, K100K,..., K4, K2}

or an AND of an exact match with a negation of an exact match condition: query Q2B:

```
   Q2B:    SELECT COUNT(*) FROM BENCH
           WHERE K2 = 2 AND NOT KN = 3;
```

For each KN ε {KSEQ, K100K,..., K4}

(iii) A retrieval of data (not counts) given constraints of three conditions, including range conditions, (Q4A), or constraints of five conditions, (Q4B).

```
  Q4:      SELECT KSEQ, K500K FROM BENCH
           WHERE  constraint with (3 or 5) conditions ;
```

Details of the constraints are given in Section 2.2.4. Several of these query types recur in considerations of other applications, a good sign that they are fundamental queries. The outputs of these queries and all others are directed to an ASCII (or EBCDIC) file. This implies that the query engine must format the numeric answers in printable form.

1.3.2 Direct Marketing. This is a class of applications whose general goal is to identify a list of households which are most likely to purchase a given product or service. The approach to selecting such a list usually breaks down into two parts: (i) preliminary sizing and exploration of possible criteria for selection, and (ii) retrieving the data from the records for a mailing or other communication.

R.L. Polk, a respondent to our survey, is the largest direct marketing list compiler in North America, with demographic information on 80 Million U.S. households. A typical query in the preliminary sizing phase might be to count the households from a set of Zip-codes, with income $50,000 per year and up, which own a car from a list of make/year categories. In most cases the mailing to be performed is relatively explicit as to size, and a count above or below the target will mean that a new query must be specified, perhaps with a change in the set of Zip-codes or the income constraint. When the list of households in the list has been selected, the individual record data is retrieved in an offline batch run, together with other lists generated.

Saks Fifth Avenue has a very effective customer tracking application, with 3.8 million records and 30 million individual customer purchases for the prior 24 months, maintained as repeating fields. Analysts at Saks often create mailings out of several smaller subject profiles such as this: a customer in the Chicago area, with total purchases of more than $1000.00 per year, and purchases in the last six months in handbags. A different profile might require purchases six to twelve months ago in luggage. Each profile is chosen on the basis of a crosstabs report on cross purchasing, explained later. The counts desired in the mailing from each of the individual profiles are pre-chosen, and the intent is usually to choose the customers with largest total purchases per year within these constraints.

Consideration of the preliminary sizing phase of the direct marketing application reinforced our selection of the `COUNT` queries, Q2A and Q2B with two clauses, and the data retrieval phase reinforced queries Q4A and Q4B, which have three and five clauses respectively. In addition it led us to specify a pair of queries where a `SUM` of column `K1K` values is retrieved with two qualifying clauses restricting the selection, one an equal match condition, and one a range query.

```
Q3A: SELECT SUM(K1K) FROM BENCH
       WHERE KSEQ BETWEEN 400000 AND 500000 AND KN = 3;
```
For each `KN ε { K100K,..., K4}`

In addition, Query Q3B captures a slightly more realistic (but less intuitive) OR of several ranges corresponding to a restriction of Zip-codes:

```
Q3B:  SELECT SUM(K1K) FROM BENCH
      WHERE (KSEQ BETWEEN 400000 AND 410000
            OR    KSEQ BETWEEN 420000 AND 430000
            OR    KSEQ BETWEEN 440000 AND 450000
            OR    KSEQ BETWEEN 460000 AND 470000
            OR    KSEQ BETWEEN 480000 AND 500000)
      AND   KN = 3;
```

For each KN ε { K100K,..., K4}

The SUM aggregate in queries Q3A and Q3B requires actual retrieval of up to 25,000 records, since it cannot be resolved in index by current commercial database indexing methods; thus, a large data retrieval is assured.

1.3.3 Decision Support and Management Reporting. This application area represents a wide class of applications, usually involving reports to aid operational decisions. One common example is a crosstabs report: a two-dimensional factor analysis table, where each two-coordinate cell is filled in with a count or sum of some field from the record set chosen. In this way, the effect of one factor on another can be analyzed.

The direct marketing subsidiary of the advertising firm of Young & Rubicam obtains initial mailing lists of perhaps two million records for clients, and then subjects these lists to a great deal of further analysis to add value. A list is analyzed and segmented by the demographic and psychographic data available; then a set of survey mailings with different promotional messages are sent to a statistical subset of each of the target segments; a large response of about 15% is obtained, and the response is analyzed, for example to see if the message sent would tend to increase purchases. The results are often presented in crosstab reports, for example showing how hobbies (boating, hiking, etc.) might affect use of some product, or how each of a set of promotional messages affects individual segments.

Saks Fifth Avenue, has a large set of reports on buying habits of their customers. Reports which return numbers of customers with total sales by category serve to identify classes of customers who return the greatest profits to Saks; these customer classes can then be individually targeted. A two dimensional array of cross-buying patterns by department supports decisions of what customer profiles should be used for mailings. For example, if we notice that customers with large luggage purchases often purchase new coats shortly later, we can include recent luggage buyers in a mailing for a coat sale.

The queries which arose out of this application area include various COUNT queries already mentioned, as well as queries Q3A and Q3B which return a SUM of a specified column from a set of selected records. The Crosstabs application so common in this category is also modelled by Query Q5, a SQL GROUP BY query which returns counts of records which fall in cells of a 2-dimensional array, determined by the specific values of each of two fields.

```
Q5:   SELECT  KN1, KN2, COUNT(*)  FROM BENCH
      GROUP BY KN1,KN2;
 For each (KN1, KN2) ε { (K2,K100),(K10,K25), (K10,K25)}
```

This is as close as SQL comes in a non-procedural statement to a crosstabs report.

Queries Q6A and Q6B, exercise the join functionality that would be needed in second two applications above, when data from two or more records in different tables must be combined.

```
Q6A:  SELECT COUNT(*) FROM BENCH B1,BENCH B2
      WHERE B1.KN = 49 AND B1.K250K = B2.K500K;
 For each KN ε { K100K, K40K, K10K, K1K, K100}

Q6B:  SELECT B1.KSEQ, B2.KSEQ FROM BENCH B1,BENCH B2
      WHERE B1.KN = 99
      AND   B1.K250K = B2.K500K
      AND   B2.K25 = 19;
 For each KN ε {K40K, K10K, K1K, K100}
```

Note that although a COUNT is retrieved, such join queries cannot be resolved via an index without reference to a large number of table rows.

1.4 Running the Benchmark

In the following Section, Section 2, we present an application of the Set Query benchmark to two different IBM System/370 commercial databases, DB2 and MODEL 204. The presentation should serve as a model for how the results are to be reported and as a good illustration of the kind of performance considerations which arise; further details on how to run the benchmark and how to interpret the results will be covered in Section 3. But before diving into the welter of detail of a benchmark report of this kind, a few high level observations are in order.

• **Architectural Complexity**. Measuring these two products exposes an enormous number of architectural features peculiar to each. Even the form of information reported is affected by this. The three modalities of resource utilization reported for each query are: elapsed time, CPU time, and I/O use. MODEL 204 uses standard (Random) I/O, with a certain amount of optimization for sequential scans transparent to the user, but DB2 has two different types of I/O which it performs: Random I/O (of a single page) and Prefetch I/O (of a block of pages chained together for optimal access efficiency). Clearly we must report these two measures separately in the DB2 case. Similarly, MODEL 204 has a number of unique features, such as an Existence Bit Map, by which it is able to perform much more efficient negation queries. A good familiarity with the architecture of any system is a necessary preliminary to a benchmark, if nothing else to assure good tuning. But more than this, a perfect apples-to-apples comparison between two products in every feature is generally impossible. Thus judgements must be made about such features in order to achieve the best comparison possible among all products measured. For example, although in this example we generally tried to flush database buffers between queries, it turns out to be difficult to flush the pages of the Existence Bit Map of MODEL 204. On consideration, it seems appropriate to allow this small set of pages to remain in memory since they would be consistently present in buffer in any situation where queries involving the represented rows are at all frequent and therefore of concern from a performance standpoint.

• **A Single Unifying Criterion.** We are aided in our desire to compare different query engines with variant architectural features by the fact that our ultimate aim is to sum up our benchmark results with a single figure: Dollar Price per Query Per Second ($PRICE/QPS); this is comparable to the aim of the DebitCredit and TPC benchmarks to measure each platform in terms of Dollar Price per Transactions Per Second.($PRICE/TPS) In looking through the many detailed query results of Section 2, the reader should keep in mind that this relatively simple criterion will guides the judgements. Indeed, as is shown in Section 3, the queries of this benchmark are meant to form a "spanning set" in terms of functionality and selectivity, so that a site-specific set of applications can be compared between two platforms in terms of a weighted sum of the queries from the benchmark which are most representative. Thus a single *custom* measure to compare performance of two systems in a customized application environment can be constructed from a pre-existing benchmark.

• **Confidence in the Results.** The results of Section 2, show occasional variations in performance between two products which are startling: Table 2.2.1, containing Q1 measurements, displays a maximum elapsed time of 39.69 seconds in one case and 0.31 seconds in the other. How can we have confidence that we have not made a measurement or tuning mistake when confronted with variation of this kind? The answer lies in understanding the two system architectures. With sufficient understanding of the architectures involved, one can step back and confirm the measurements in terms of more detailed measures implicit in other queries. This is a form of "multiple entry bookkeeping" which helps validate the conclusions on "multiple strands" of evidence, rather than the "links of a chain" favored in certain areas, where a failure of a single link invalidates the result. For example, the actual number of pages of I/O required by each architecture can be predicted in each of the queries of Section 2. This is why the benchmark has such a careful discussion following the various Query Tables: a consistency check must be performed whenever possible to increase confidence in the result. By analogy, if the period of recorded history is 4,000 years, then there is a record of the sun rising on 730,000 mornings. Yet we would probably be willing to offer 10,000,000 to 1 odds in a bet that the sun will rise tomorrow and think it a safe bet. Our understanding of the underlying scientific principles involved makes it inconceivable that we are in error on this point. Benchmark measurements on complex software systems of the kinds described here share many aspects of a scientific investigation.

• **Output Formatting.** The query results are directed to a user file which represents the answers in printable form. About half the reports return very few records (only counts), and about half return a few thousand records.

2. AN APPLICATION OF THE BENCHMARK

The Set Query benchmark was applied to two commercial databases for IBM System/370 machines, DB2 and MODEL 204. The results show variations of surprising magnitude in performance, much larger differences than are usually seen in DebitCredit measurements for example; these differences are often due to fundamental variations in basic architecture, which the basic queries of the benchmark amply illustrate.

2.1 Hardware/Software Environment

We ran DB2, Version 2.2 with 1200 4K byte memory buffers, and MODEL 204 Version 2.1 with 800 6K memory buffers, accessing the BENCH database loaded for each system on two 3380 disk drives. All measurements were taken on a standalone 4381-3, a 4.5 MIPS dual processor. Since all tests are single user type, only one 2.25 MIPS CPU was ever utilized by the queries. Both database systems were running under the MVS XA 2.2 Operating System. Interactive query interfaces were used in both cases: DB2 queries were run using the SPUFI interface on VTAM with the TSO attachment; MODEL 204 queries were also run on VTAM, with access through its own built-in interface.

The `EXPLAIN` command was used in DB2 to determine the access strategy employed by the system for each query. `RUNSTATS` was first run to update the various tables, such as `SYSCOLUMNS` and `SYSINDEXES` on which the DB2 query optimizer bases its access strategy decisions. During the runs reported here, accounting trace Class 1 and Class 2 were turned on, resulting in CPU increase of about 10%. The SMF output was analyzed by DB2PM. Class 1 statistics reported here reflect inter-region communication time and time spent by the SPUFI application as well as DB2 time.

MODEL 204 statistics were put out to the interactive screen and the audit trail as each query was performed, using the `TIME REQUEST` command. This results in a CPU overhead of about 2%-3%.

2.2 Statistics Gathered

To start with, we present the time required to load the BENCH table on the two systems

Table 2.1. Time to load BENCH database

Elapsed	DB2 CPU	DB2 Elapsed	M204 CPU	M204
TABLE LOAD	124m 19s	114m 43s	108m 34s	142m 13s
RUNSTATS	41m 17s	33m 47s	0s	0s
TOTAL	**165m 36s**	**148m 30s**	**108m 34s**	**142m 13s**

For DB2, the `RUNSTATS` utility is viewed as an element of the load time, since `RUNSTATS` is necessary in order to obtain the query performance measured in the benchmark. The `RUNSTATS` utility examines the data and accumulates statistics on value distributions which are subsequently used by the DB2 query optimizer in picking query plans. MODEL 204 also performs some optimization on the basis of value distributions, the rest through programmer decision in the procedural query language: it gathers needed information for the optimization it performs as an integral part of loading, without employing a separate utility function.

The MODEL 204 elapsed time is shorter than the CPU time recorded in this table because the MODEL 204 loader is able to overlap work on the dual processors of the 4381, the only multi processor use seen for either product in the current benchmark.

The disk space occupied by the BENCH database for the two products was:.

Table 2.2. Disk space utilization in bytes

DB2	M204
313,660,000	265,780,000

The rows of the BENCH database were loaded in the DB2 and MODEL 204 database in identical fashion. All the indexed columns, KSEQ, K500K,..., K2, were given a B-tree index. The indices in both cases were loaded with leaf nodes 95% full. Since the lengths of the columns sum to 200 bytes, the rows in both databases were slightly in excess of 200 bytes in length as a result of required row storage overhead.

Table 2.3. DB2 Index and Data Characteristics

Max number of Index pages (K500K)	2290
Min number of Index pages (K2-K100)	1110
Row Length in Bytes	224

Rows per tablespace page	17
Total tablespace pages	58,824

The number of B-tree leaf pages is 1110 for many indices, increasing to 2290 pages as the structure becomes more complex with a large number of values and a varying number of rows per value.

MODEL 204 databases deal with somewhat larger pages (6K bytes), and B-tree indices have a complex indirect inversion structure as explained in [5]. Basically, MODEL 204 B-tree entries need not contain lists of RIDs of rows (records) with duplicate values, but may instead point indirectly to a list or "bitmap" of such RIDs, whichever MODEL 204 finds more efficient.

In MODEL 204, the native "User Language" was used to formulate these queries, rather than SQL. User Language is a procedural 4GL recommended by CCA for its flexibility in application building over the host language embedded approach. Query Q1, given below in SQL, would be written in User Language as in Figure 2.4. The difference for most of the queries reported here is negligible and only the SQL form is presented here.

Figure 2.4. M204 User Language version of Q1

```
BEGIN
FIND AND PRINT COUNT  KN = 2
END
```

For each KN ε { KSEQ, K100K, ..., K4, K2}

2.2.1. Q1: Count, Single Exact Match

To restate Query Q1, mentioned above:

```
SELECT COUNT(*) FROM BENCH
WHERE  KN  =  2;
```

For each KN ε {KSEQ, K100K,..., K4, K2}

This is a typical early query in document retrieval search, to find the number of Journal articles with some property (published in the JACM, containing the key word "Computer", etc.).

Table 2.2.1, Q1 measurements (elapsed times in seconds)

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
KSEQ	1.13 s	0.05 s	0.52 s	0.01 s	3		2
K100K	1.11 s	0.06 s	0.53 s	0.03 s	3		2
K10K	1.14 s	0.12 s	0.54 s	0.07 s	3		2
K1K	1.21 s	0.15 s	0.58 s	0.11 s	4		2
K100	1.76 s	0.22 s	1.06 s	0.16 s	12		5
K25	5.02 s	0.33 s	3.56 s	0.15 s	3	6	22
K10	9.63 s	0.31 s	8.14 s	0.15 s	3	10	23
K5	17.65 s	0.35 s	15.77 s	0.15 s	3	12	22

K4	20.36 s	0.29 s	19.48 s	0.15 s	2	13	22
K2	39.69 s	0.31 s	38.42 s	0.15 s	1	19	22

As mentioned earlier, DB2 times come from DB2PM Class 1 statistics, which include time spent by the SPUFI application and cross-region communication. Class 2 statistics include only time spent within DB2, and compare to Class 1 in most of the queries reported by consistently subtracting almost exactly 0.8 seconds elapsed time and 0.3 seconds CPU time. This makes a large percentage difference for the smaller measures, but is relatively insignificant for the larger ones.

Note that the I/O measures for DB2 are given in terms of the number of random single-page I/Os, as well as the number of sequential prefetch I/Os. With classic prefetch reads, the disk head sweeps through a read of 32 blocks at once, performing the work of 32 random reads in about twice the time. The data to be accessed must lie on successive pages on disk for such reads to be of value, as here where large numbers of successive equal match entries in a B-tree have been loaded in sort order. Note that if we represent the total number of pages brought into memory by N, the prefetch reads by P and random reads by R, we see that: N = R + 32 * P. In the measured case where `KN` stands for `K2` above, we find that N = 1 + 32 * 19 = 609, a bit more than half the number of pages in the 1110 pages of the `K2` index B-tree, as we would expect.

We see in Table 2.2.1 a large disparity between DB2 and MODEL 204, with palpable slowdown in DB2 retrieval as the selectivity of the match column decreases. The `EXPLAIN` command gives the access plan of the DB2 `SELECT` statement, that it uses the index of the column in question, and resolves `COUNT(*)` solely in the index, without access to the row data. This is perfectly appropriate. The entries in the B-tree which must be accessed are exactly that set of duplicate values in each index equal to 2. Note that, as mentioned above, all of the low selectivity column indices have 1110 pages (Table 2.3), and the total number of pages read in each case is what we would expect, i.e., in the case of `K2 = 2`, approximately half the index pages.

The DB2 slowdown is not caused by the I/O, however; the CPU time is the culprit, as can be seen from the fact that CPU time is most of the elapsed time. Considering the "K2 = 2" case, shows a time of 38.42 seconds on a 2.25 MIPS CPU to count 500,000 B-tree entries. This means that DB2 uses about 173 instructions for each entry counted. This is an improvement from an earlier architecture where successive entries in the B-tree were accessed by one software layer and counted by another, but it is still surprisingly CPU consuming. In the MODEL 204 architecture by contrast, in cases where a large number of records must be counted, MODEL 204 simply counts bits in several blocks of memory, an operation which can be highly optimized. This serves to explain the large difference seen.

2.2.2. Q2: Count ANDing two Clauses

This test query type performs a COUNT of records from an AND of two exact match conditions (query Q2A), or an AND of an exact match condition with a NOT of an exact match condition (query Q2B).

```
    SELECT COUNT(*) FROM BENCH
    WHERE K2 = 2 AND KN = 3;
```

For each KN ε {KSEQ, K100K,..., K4}

Query Q2A constrains two columns at once, one of low selectivity, the other varying. It is a typical query in the early stages of document retrieval search. It might also be used in direct marketing applications to get an early estimate of the number of prospects with a given set of qualities.

Table 2.2.2A, Q2A measurements

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
KSEQ	1.39 s	0.10 s	0.53 s	0.01 s	1		3
K100K	1.43 s	0.19 s	0.54 s	0.04 s	1	1	9
K10K	2.34 s	0.56 s	0.58 s	0.12 s	1	4	23
K1K	12.41 s	0.57 s	1.00 s	0.13 s	2	31	24
K100	74.50 s	0.71 s	5.41 s	0.20 s	15	287	26
K25	119.77 s	0.97 s	19.16 s	0.19 s	16	929	43
K10	143.80 s	1.31 s	38.32 s	0.19 s	4	1541	43
K5	165.87 s	0.98 s	65.09 s	0.19 s	4	1811	43
K4	138.17 s	0.98 s	65.92 s	0.19 s	3	1840	43

With the release of V2.2, DB2 was able to combine two or more indices before accessing the tablespace, by sorting the RIDs selected by individual match predicates and performing a merge-intersection. However, the EXPLAIN command reveals that DB2 avoids this here, using the index of higher selectivity of its ANDed conditions to cut down the number of rows to be searched, then accessing each of the rows in question to resolve the second condition (check if K2 = 2). In the last line of the query where K2 = 2 AND K4 = 3, a tablespace scan would be used. The decision not to use index merge-intersect, even though it saves significant in I/O cost, seems to reflect a costly CPU implementation of this feature.

In the heavy resource use of the last Table 2.2.A row of DB2 measures, the I/O wait time accounts for the elapsed time: each sequential prefetch of 32 pages requires over 3 full rotations of the disk for transfer alone, about 55 milliseconds; adding in 10 ms for seek time and 8 ms for rotational latency, we arrive at an elapsed time for 1837 prefetch I/Os: 1837 * (.055 + .010 + .008) = 134.10 seconds. Earlier rows of Table 2.2.2A, such as the one for K100, demonstrate the new List Prefetch feature of V2.2. With K100, it retrieves about 10,000 rows, on about 10,000 * exp(-10,000/58824) = 8,437 pages, scattered randomly through the tablespace. This is accomplished with 287 list prefetches (reported by DB2PM as sequential prefetches), where 32 pages at a time are retrieved through “chained” commands to the disc controller and channel, pages which are not sequentially placed but merely as clustered as possible! Note that the average time for each list prefetch in the K100 case is about 74.50/287 = .2596 seconds, more than three times as long as a sequential prefetch.

In the case of MODEL 204, all of the information necessary for this query is resolved by combining the information of two indices, basically a matter of `AND`ing the bitmaps of the two indices.
Now consider the related Q2B query which involves a `NOT` clause.

```
   SELECT COUNT(*) FROM BENCH
   WHERE K2 = 2 AND NOT KN = 3;
```

For each `KN ε {KSEQ, K100K,..., K4}`

Table 2.2.2B, Q2B measurements

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
KSEQ	137.68 s	0.46 s	84.76 s	0.16 s	13	1840	23
K100K	137.69 s	0.45 s	84.76 s	0.16 s	13	1840	22
K10K	137.93 s	0.66 s	84.82 s	0.18 s	13	1840	24
K1K	137.50 s	0.66 s	84.79 s	0.19 s	13	1840	24
K100	137.99 s	0.71 s	84.57 s	0.21 s	13	1840	26
K25	137.53 s	0.96 s	83.98 s	0.20 s	13	1840	43
K10	137.83 s	1.32 s	82.59 s	0.20 s	13	1840	43
K5	138.99 s	0.98 s	80.63 s	0.20 s	13	1840	43
K4	137.72 s	0.98 s	79.30 s	0.20 s	13	1840	43

In all DB2 measures above, the `EXPLAIN` command tells us that a Tablespace Scan is being used. Clearly, the DB2 query optimizer sees no way to use clauses such as `NOT KSEQ = 2` to reduce the work through the indices. MODEL 204 repeats the work of Q2A in performing the queries of Q2B, except that the Bitmap of records found in the `KN = 2` clause is XORed with the Existence Bitmap, which represents a list of all records in the table, resulting in a Bitmap of `NOT KN = 2`.

2.2.3. Q3: Sum, Range and Match Clause

This query type retrieves the sum of the values of an integer binary column, K1K, for a given set of rows. The first query set of this type, Q3A, is:

```
Q3A: SELECT SUM(K1K) FROM BENCH
       WHERE KSEQ BETWEEN 400000 AND 500000
       AND KN = 3;
```

For each `KN ε { K100K,..., K4}`

This is an example of a query which MODEL 204 cannot resolve totally in the indices -- all the rows involved in the sum must actually be touched by both systems -- and thus the work performed emulates a direct mailing application where name and address information is written out from each of about 25,000 rows. It is also representative of a class of management reporting application queries, such as finding average incomes of various selected classes of prospects.

Table 2.2.3A, Q3A measurements

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
K100K	1.43 s	0.27 s	0.56 s	0.06 s	4	1	13
K10K	2.41 s	1.84 s	0.59 s	0.30 s	4	4	122
K100	15.64 s	27.99 s	10.36 s	16.65 s	12	213	1148
K25	15.61 s	41.29 s	10.82 s	19.52 s	10	211	2673
K10	15.63 s	46.07 s	11.44 s	23.00 s	10	211	3583
K5	15.63 s	46.36 s	11.89 s	27.13 s	9	209	3752
K4	15.60 s	45.82 s	12.71 s	29.00 s	10	211	3753

Here DB2 performance is superior to MODEL 204 in the final five rows. The `EXPLAIN` command says that the access strategy in these cases is to use the `KSEQ BETWEEN` clause to limit the search, then resolve the remaining condition, `KN = 3`, by access to the records themselves. Since the records are clustered by the `KSEQ` values, this limits us to accessing only one-tenth of the tablespace records: the number of pages read is about 10 + 32 * 211 = 6762, slightly more than one-tenth of the tablespace (5,883 pages) + one-tenth of the `KSEQ` index (220 pages). DB2's Sequential Prefetch provides a performance gain over MODEL 204, which must access the table pages using random I/O which is only partially optimized, leading to greater I/O wait time and CPU time to make more I/O calls.

Query set Q3A was meant to model a query ranging over zipcode values corresponding to a given geographical region. It turns out that such a query usually results in a set of ranges of zipcodes which must be `OR`ed together. This is captured by query Q3B:

```
SELECT SUM(K1K) FROM BENCH
      WHERE (      KSEQ BETWEEN 400000 AND 410000
            OR     KSEQ BETWEEN 420000 AND 430000
            OR     KSEQ BETWEEN 440000 AND 450000
            OR     KSEQ BETWEEN 460000 AND 470000
            OR     KSEQ BETWEEN 480000 AND 500000)
      AND KN = 3;
```

For each `KN ε { K100K,..., K4}`

Table 2.2.3B, Q3B measurements

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
K100K	1.44 s	0.38 s	0.58 s	0.08 s	4	1	22
K10K	2.40 s	1.83 s	0.62 s	0.43 s	4	4	112
K100	14.88 s	17.15 s	9.70 s	10.19 s	110	21	657
K25	20.79 s	25.77 s	14.22 s	12.14 s	4	65	1642
K10	31.44 s	28.64 s	23.67 s	14.20 s	111	104	2182
K5	49.10 s	28.83 s	41.61 s	16.60 s	111	125	2290
K4	57.35 s	27.89 s	50.36 s	17.74 s	111	128	2207

Starting at the row for `K100`, `EXPLAIN` tells us that DB2 combines the five `KSEQ` range clauses with merge-union, then performs a merge-intersect with the `KN = 3` RIDs, and finally reads in each of the rows selected to accumulate a `SUM`. In the final row where `K4 = 3`, the total number of pages read is about (6/100) * (2290 + 58824) + (1/4) * (1110) = 3940, which requires about 120 random reads and 120 prefetch reads of 32 pages to accomplish, close to the figures measured.

MODEL 204 accesses the same indices and data as DB2. MODEL 204 is faster in Elapsed time in the last three rows because of the CPU time expended, since the merge-union and merge-intersect operations of MODEL 204 are much less CPU intensive.

2.2.4. Q4: Multiple Condition Selection

The query type to be examined in this section retrieves data, rather than a count, from records satisfying multiple `AND`ed conditions:

```
SELECT KSEQ, K500K FROM BENCH
WHERE  constraint with (3 or 5) conditions ;
```

The output data is directed to a printable disk file. The constraints of these query sets are generated by `AND`ing 3, or 5, successive conditions chosen from the condition sequence below. In the 3 condition constraints, for example, we use conditions 1-3, then conditions 2-4, 3-5, etc. The total selectivity of each 3 conditions in sequence is such that the number of rows retrieved is between 785 and 10,294, whereas in the case of 5 conditions in sequence, the number of rows retrieved varies from 199 to 465. See Appendix A for the count of rows retrieved in each case.

Condition Sequence.

```
(1) K2 = 1;                            (2) K100 > 80;
(3) K10K BETWEEN 2000 and 3000;        (4) K5 = 3;
(5) K25 IN ( 11, 19);                  (6) K4 = 3;
(7) K100 < 41;                         (8) K1K BETWEEN 850 AND 950;
(9) K10 = 7;                           (10) K25 IN (3 , 4)
```

A conjoint query of this type is clearly a possible final query resulting from document retrieval search. It also might appear in the second phase of direct mail applications.

Table 2.2.4A, Q4 constraint with 3 conditions

COND'N SEQ NOS	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
1-3	147.62 s	114.03 s	102.27 s	20.95 s	15	560	9006
2-4	120.38 s	56.02 s	100.41 s	10.78 s	19	157	4027
3-5	145.32 s	27.99 s	62.07 s	6.03 s	16	552	1854
4-6	195.21 s	50.62 s	108.97 s	7.54 s	16	1826	3959
5-7	174.53 s	94.94 s	110.47 s	15.51 s	13	1839	7408
6-8	163.08 s	114.50 s	114.58 s	19.78 s	16	674	9224
7-9	105.38 s	54.13 s	54.38 s	9.30 s	16	308	4056
8-10	54.10 s	13.99 s	46.76 s	2.38 s	19	54	930

`EXPLAIN` says that DB2 uses merge-intersects of RID sequences to `AND` pairs of clauses before accessing the data, except that clauses 1, 5, and 7 are never merged. Thus the sequence 5-7 only uses `K4 = 3` to limit the search, whereas the sequence 4-6 merge-intersects `K5 = 3` and `K4 = 3`. Sequences 2-4 and 8-10 merge all three indices. The CPU times needed are generally quite large, a reflection of the DB2 cost of merging indices, especially where B-tree scans are needed.

In the MODEL 204 case, the set of records satisfying the `AND`ed conditions is decided at the index level (all clauses are always `AND`ed); following this, the rows involved are accessed to form the `SUM` -- thus, retrieving about 10,000 rows involves 9006 page I/Os in the first M204 row above, at about 13 ms elapsed for each of these random I/Os (a certain amount of optimization is built in, so that random I/Os are faster than standard). The CPU utilized by MODEL 204 here is generally a small fraction of that used by DB2.

Table 2.2.4B, Q4 constraint with 5 conditions

COND'N SEQ NOS	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
1-5	119.64 s	11.75 s	81.99 s	3.93 s	19	157	486
2-6	119.47 s	11.28 s	81.56 s	3.84 s	19	157	411
3-7	133.65 s	13.69 s	89.29 s	4.57 s	20	187	533
4-8	137.56 s	11.80 s	92.77 s	2.60 s	19	191	541
5-9	99.42 s	10.75 s	72.98 s	2.42 s	19	113	448
6-10	54.37 s	9.10 s	43.73 s	2.37 s	19	54	404
7-1	54.11 s	10.45 s	43.87 s	2.55 s	19	54	480

Once again DB2 uses merge-intersects of RID sequences to `AND` pairs of clauses before accessing the data, except at most three clauses are ever used and that clauses 1, 5, and 7 are never merged (Clause 6 is used only once).

MODEL 204 is much faster, both in elapsed time and CPU utilization, because it combines all index clauses very quickly and at the end very few records actually need to be retrieved.

2.2.5. Q5: Multiple Column GROUP BY

The query which is considered in this section is one where a count of rows is retrieved for a matrix of conditions based on two varying parameters.

```
Q5:   SELECT  KN1, KN2, COUNT(*)
      FROM BENCH
      GROUP BY KN1,KN2;
```
For each `(KN1, KN2) ε { (K2,K100),(K10,K25), (K10,K25)}`

This is a reasonably common type of query, used in management reporting and decision support to calculate the effect of one factor on another, as explained in Section 1.

Table 2.2.5, Q5 measurements

KN1,KN2	**DB2 ELAPSED TIME**	**M204 ELAPSED TIME**	**DB2 CPU TIME**	**M204 CPU TIME**	**DB2 RAND I/Os**	**DB2 PREF I/Os**	**M204 DISK I/Os**
K2,K100	535.38 s	43.60 s	526.13 s	36.27 s	3	1840	420
K4,K25	508.25 s	28.36 s	500.99 s	16.72 s	3	1840	595
K10,K25	540.96 s	56.07 s	533.58 s	40.80 s	3	1840	744

In this query, DB2 attempts to scan all rows in the tablespace and put out the `KN1, KN2` values to a scratch file, then sort these value pairs, and gather duplicates with a count. The sort involved is extremely resource consuming. MODEL 204 is programmed to do the job by performing a double loop on the values involved (values in B-Tree indices can always be looped on in the procedural MODEL 204 User Language), and counting the conjoint selection with each pair of values. This is a much more efficient approach than sorting in cases where the table for the end result fits in memory, as is usually the case in a `GROUP BY` query.

2.2.6. Q6: Join Condition

The final query type considered involves a Join of the bench table with itself, and we measure two different cases, Q6A and Q6B. In Query Set Q6A, there is an equality condition limiting the rows of the the first table only, then a match across to the second table, and a `COUNT` is taken of the number of rows formed. We use a `COUNT` here, rather than retrieve up to 20,000 rows of data into the SPUFI buffer. The Query Q6B, however, retrieves data.

```
Q6A:  SELECT COUNT(*) FROM BENCH B1,BENCH B2
      WHERE B1.KN = 49 AND B1.K250K = B2.K500K;
```
For each `KN ε { K100K, K40K, K10K, K1K, K100}`

In Query Set Q6B, we retrieve data from a join of two tables, where equality conditions limit the rows on both tables of the join. (See below.)

Queries such as Q6A and Q6B might arise in a Direct Marketing mailing list application, where information for individual prospects are for some reason partitioned into separate tables. The join of the BENCH table with itself presents exactly the same problem to the query strategy as the join of two separate (large) tables, since the overlap between the outer join table and the inner join table is relatively insignificant. Overlap in buffering is therefore not a factor and there exists no special-purpose strategy for a table self join which is not comparable to joining two tables.

Table 2.2.6A, Q6A measurements

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
K100K	1.83 s	0.63 s	0.73 s	0.15 s	30	1	33
K40K	2.11 s	1.03 s	0.78 s	0.29 s	33	1	53
K10K	3.41 s	3.17 s	1.14 s	1.09 s	99	4	209
K1K	15.52 s	21.27 s	4.95 s	9.04 s	644	31	1617
K100	131.01 s	196.36 s	47.85 s	87.58 s	5608	292	14167

The DB2 plan finds all rows for which B1.KN is equal to 49, then reads each of these rows (using List Prefetch in later rows) to find its K250K value, and adds to a total count all rows whose K500K value is equal to this. MODEL 204 has a similar strategy. Both systems are forced to access all rows in the first Table to determine the value of the Join column, B1.K250K, and can then count the rows in the join by using the B-tree entries for B2.K500K. DB2 has an edge in the later times measured, because of the list prefetch capability.

```
Q6B:  SELECT B1.KSEQ, B2.KSEQ
      FROM BENCH B1,BENCH B2
      WHERE B1.KN = 99
      AND B1.K250K = B2.K500K
      AND B2.K25 = 19;
```
For each KN ε {K40K, K10K, K1K, K100}

Table 2.2.6B, Q6B measurements

KN	DB2 ELAPSED TIME	M204 ELAPSED TIME	DB2 CPU TIME	M204 CPU TIME	DB2 RAND I/Os	DB2 PREF I/Os	M204 DISK I/Os
K40K	3.01 s	1.34 s	0.87 s	0.27 s	23	41	69
K10K	9.62 s	3.33 s	1.58 s	0.78 s	169	98	232
K1K	75.31 s	21.57 s	9.46 s	6.18 s	1609	888	1710
K100	697.34 s	197.63 s	89.74 s	58.71 s	16338	9006	15245

This query models Name and Address retrieval in a direct marketing application. The access strategy in DB2 was to find all rows for which B1.KN is equal to 99, read each of these rows to find the B1.K250K value, then access all B2 rows whose B2.K500K value was equal to this and qualify them according to whether `B2.25 = 19`. Since there are an average of two `B2` rows for each `B1 row`, Q6B accesses nearly three times as many rows as Q6A, and the elapsed time to perform the query is greater.

MODEL 204 uses the first strategy above, but limits its accesses to table `B2` by performing a `COUNT` of `B2` rows with `B2.K500K` values equal to the `B1.K250K` value which *also* have the property that `B2.K25 = 19`. Since the count is resolved in the index, only about one row in 25 will have to be accessed in Table `B2`. Thus only a slight increase in resources over Q6A is required.

2.3 Single Measure Results

The ultimate measure of the Set Query benchmark for any platform measured is price per query per second ($/QPS). As explained in Section 3.3, below it is possible to select from the spectrum of results of the benchmark to derive such a measure for the customized work of a particular site. As a rough preliminary measure we calculate the average query resource use under the assumption that all queries of the benchmark are of equal weight (certainly an unlikely real weighting since the queries requiring fewer resources will tend to be preferred).

3. HOW TO RUN THE SET QUERY BENCHMARK

3.1 Generating the Data

The following pseudo-code procedure gives the algorithm used to generate the 12 indexed random column values, K2 through K500K, for the BENCH table. The algorithm for this pseudo-random number generator is suggested by Ward Cheney and David Kincaid in [3], Section 9.1. This procedure is available on tape from Morgan Kauffman, programmed in C.

Table 3.1. Generator for Indexed Field Data Generation

```
PROCEDURE FIELDVGEN()         /*  Procedure to generate indexed field values */
INTEGER I, J;                 /*  Looping Variables                          */
DOUBLE SEED INIT (1.);        /*  Seed For Random Number Generation          */
INTEGER ARRAY COLVAL(1:12),   /*  Array to hold one row of values            */
   COLCARD(1:12)              /*  Array holding largest value in each field */
   INIT (500000,250000,100000,40000,10000,1000,100,25,10,5,4,2);/* Initialize */
FOR I FROM 1 TO 1000000;      /* One Million Rows                            */
   FOR J FROM 1 TO 12;        /* Twelve randomly generated column values    */
      SEED = MOD(16807.0D * SEED, 2147483647.0D);
   /* I.E.: Si+1 = (7**5 * Si ) MOD (2**31-1), where Si is SEED, a random # */
      COLVAL(J) = MOD(SEED, COLCARD(J) )+1;   /* Generate next column value */
   END FOR J                                              /*  Row complete */
   PRINT COLVAL VALUES FOR THIS ROW TO OUTPUT;  /* Output row's values      */
END FOR I                     /*  We have generated One Million Rows        */
END
```

Note that the column values generated are whole integers, but the calculations are performed in double precision arithmetic. This is because the intermediate results in the calculation of successive random numbers can become as large as $(7^5)*(2^{31}-2)$, too large for accurate representation as 32 bit integers. Programmers lacking a `MOD` function which applies to double precision whole numbers should perform their own `MOD` calculation by performing the division and finding the remainder, being careful to choose the nearest whole number, rather than the `TRUNC`. The correct column values to generate for the first ten rows can be found in Table 1.2.

We see that the `KSEQ` values are not generated here (although they could be), since they are simply the successive values 1, 2, 3, . . ., the ordinal number of each row in order, which can often be created while the Load is taking place. Recall too, that in addition to these indexed columns, there are also a number of character columns, s1 (length 8), s2 through s8 (length 20 each) which fill out the row to a length of 200 bytes. These character strings are generated with identical values (not compressed by the database load) since they are never used in retrieval queries, a reflection of the fact that unindexed retrieval in large, low update tables is extremely inadvisable.

3.1.1 Loading the Data

Indexes for these 13 columns, `KSEQ` together with the columns `K2` through `K500K`, are the only indices which should be created for the `BENCH` table; to allow for range search and possible small corrections, the indices should be of B-tree type (if available) and loaded 95% full. While additional concatenated indices may improve query performance in some products, there is difficulty in deciding which pairs or triples of columns should be concatenated; it is inappropriate to index all pairs, for example, since $(13\times12)/2$ indices would be required, and this is extremely wasteful of space. Therefore, in the interests of comparing results between different systems, only these 13 indices are allowed for the benchmark. While it is appropriate in many real situations to use concatenated indices with current database systems, the future of query performance seems to lie with efficient ad-hoc combinations of single column indices. The addition of such a feature with DB2 V2.2 seems to endorse this point.

The Set Query benchmark permits the data to be loaded on as many disk devices as desired. However, only those systems which break down a single query into a number of parts for parallel execution and improved elapsed time performance can keep more than one disk arm busy at one time on behalf of a single user. Since all devices used must be included in the price of the system, two disks is usually sufficient on single-threaded systems, one disk for data and one for indexes. (Since the pattern of page references often alternates between data and index, using two disks for these purposes can often reduce seek time with obvious improvement in performance.) The disks referred to above are rotating platter disks or equivalent devices which contain the data when power is turned off. Semi-conductor disks can also be configured on the system, but the time required to populate them at system startup must be reported.

3.2 Running the Queries

With the advent of much cheaper memory on many systems, we must be extremely attentive to the question of how much of the data and indexes for the BENCH table we allow to be held in memory. There are a number of considerations to take into account.

First, assume that one platform provides much less expensive memory than another. Since our criterion of comparison between two systems is simply to compare the values of $PRICE/QPS, we must prefer a platform which can make a larger portion of the BENCH table memory resident if this translates into a reduced value for this number. At the same time, we need to realize that the various commercial systems we are attempting to model with the Set Query benchmark might have different requirements of size. A one-million row Bench table can be completely loaded into a memory of about 300 million bytes, as we see from Table 2.2. This implies that a 100-million row table will require 30 billion bytes of memory, which is not feasible on current machines. If we are trying to model a commercial system with size requirements comparable to the one-million row table, the best strategy might be to start by loading all data and index into memory -- naturally this preparation time must be reported in the benchmark report. On the other hand, if we are modelling a 100-million row commercial system, we might wish to limit the memory size quite strictly, since memory sizes that permit only a small fraction of data and index residence (10%, say), offer only a small advantage in performance, and this advantage may not be worth the additional memory price.

To support tuning decisions regarding memory purchase, the Set Query benchmark offers two approaches. First, it is possible to scale the BENCH table to any integer multiple of one million bytes, populating memory buffers in advance and thus modelling most closely the eventual size of the commercial system. As we will see in Section 3.3, a special scaling of the results is necessary to compare results of a one-million row table with those of, say, a ten-million row table. The only problem with this approach is that no single benchmark provides appropriate results for all different sizes of databases modelled; in the worst case, a different proportion of data and index should be present in memory for each different size. As a second alternative (the default setup, reported in Section 2), the benchmark measurements are normalized by essentially flushing the buffers between successive queries, so that the number of I/Os is maximum; this approach models an extremely large database, where a significant proportion of data and index residence in memory is out of the question. An advantage is that all different platforms can be compared on this measurement. In addition, it is possible to use the data of this default benchmark and by careful analysis project the performance where various proportions of the data and index are memory resident. In general, the analyst will always know the access strategy and the index and data pages referenced; by estimating the proportion of the various entities which would be present in memory under different conditions, it is possible to project the number of resulting I/Os, reduce the CPU to reflect the overhead saved when I/O pages are found in buffer, and estimate the elapsed time, either CPU or I/O bound, which results.

In the default benchmark setup of Section 2, the amount of space which was used for memory buffering was strictly limited to 4.8 MBytes (1200 4 KByte buffers), and an attempt was made to flush the buffers between queries in cases where a substantial amount of overlap of pages was evident. The reason that 1200 buffers were used was, first, that the DB2 Query Optimizer made a decision on the size of sequential and list prefetch based on the memory buffer size so that the best results were obtained with more than 1000 buffers, and, second, that by limiting the number of buffer pages we were able to flush the buffers in a shorter period of time. One method of flushing buffers used was to run Query Q6B in the K1K case; in this query, the number of (random) pages read was over 1600 for both products, most of them data pages distributed rather randomly over the BENCH table. Another possibility for flushing buffers is to bring down the database and bring it up again, but if this approach is used, the benchmark engineer should perform some short query at the beginning of each run to get database startup work out of the way. Note that flushing buffers is time-consuming, and it is comforting that the entire sequence of queries in Table 2.2.1, for example, can be run without intervening flush actions (at least for the architecture of the two products measured here), since there is no overlap in the pages referenced. If in doubt whether two queries, A and B, have a large overlap in page access, the buffer clearing technique can be followed by Query A and then Query B, in that order, then the buffer clearing again followed by Query B and then Query A. If the measures for the two runs of Query A or the two runs of Query B are significantly different, then it is clear that the queries have an effect on each other.

Recall from the discussion following Query Q1 in Section 2.2.1, that the 22 Existence Bit Map pages of MODEL 204 were allowed to remain in buffer from one query to another. The rationale for clearing buffers is that we don't wish to allow unfeasible buffer residence in the largest databases. A residence set of 22 pages per million rows was considered sufficiently small to be acceptable, and in practice these pages would be present at all times when queries against the table were common enough to make performance a major concern. Readers interested in an objective criterion for determining the economically feasible use of memory buffers should read the interesting article in [4].

The queries of the Set Query benchmark can be run using either an interactive ad-hoc interface or a programmatic embedded interface on a standalone machine. The type of interface used should be reported, along with approximate difference in resource use between the two interfaces, if known.

3.3 Interpreting the Results

A common demand of decision makers who require benchmark results is a single number, a measure "rating" the different platforms. It is all very well to make the case that there is so much diversity among the requirements of different Set Queries that boiling this diversity down to a single measure loses too much detail, but the simplistic demand for a single rating has an important point in its favor. The ultimate purpose of a benchmark is to form a basis for deciding on the acquisition of one platform over another, a simple ranking of the alternatives which calls for a single measure. Viewed in this light, the only appropriate normative measure seems to be price-per-unit-of-throughput, which is the ultimate rating generated by the Set Query benchmark, just as it is for the TPC and the DebitCredit benchmarks. Naturally, requirements for a system may exist which are incomparable with price; a hard real-time requirement that a particular class of queries or transactions cannot take more than a specified length of time is the most obvious example. Requirements of this kind entail extra restrictions on the multi-user load and are outside the realm of current benchmarks.

The practice of reporting a spectrum of measurements along parameters of function and selectivity allows the system manager to generate a single rating on different platforms for the weighted subset of queries peculiar to his/her installation. We don't wish to lose this advantage by reporting a single number which fits no particular case. Fortunately it is simple to specify a single price related measure (with an equal weighting of all queries tested, which is probably quite unrealistic in most cases), and still require that the benchmark report on the details of the measurements so that customized ratings can still be created.

It is important to note that the procedure outlined below to calculate a rating for price per query per second (P$/QPS) may be optimistic! To generalize our single-user results to a multi-user environment assumes that there is no interference between queries run in parallel, and that the query rate will be limited only by the CPU and I/O resources available. This assumption is theoretically justified by the fact that there is no reason why locks of these queries should interfere with one another, since they are read-only queries. Also, there are no requirements for logging or other updates which might create a bottleneck for these systems as the query rate increases. This is quite a different story from the transactional systems measured by DebitCredit and TPC, where bottlenecks of these kinds are quite important. However, in spite of this theoretical justification for assuming non-interference between simultaneously executing queries, it is certainly possible that some improperly implemented database systems exhibit such interference. In general, the user should beware of situations where the assumptions of the following price calculations may be too optimistic.

3.3.1 Calculating the $/QPS Rating

The calculation of the $PRICE/QPS rating for the Set Query benchmark is quite straightforward in concept. There are 69 queries in the benchmark, and if we say that the elapsed time to run all of them in series is T, then we are running 69/T queries per second. The system on which the benchmark is run has a well defined dollar price, P, for hardware and software (where software cost and hardware maintenance charges is given as a rental fee, the common accounting technique is to add up the costs over a five year period and take this as the full price). Now the $PRICE/QPS rating is clearly given by $\frac{P \bullet T}{69}$.

Unfortunately, a certain amount of complexity enters because the elapsed time to run all the queries is not exactly the value for T we wish to use. While we are measuring a single-user sequence of queries, we assume that we ultimately wish to calculate prices and performance for a multi-user system. The usual assumption made with a multi-user system is that multiple disks will be purchased, if necessary, to assure that all the CPUs of the system remain relatively fully utilized. Thus, while some queries of the Set Query benchmark might be heavily I/O bound, utilizing only 10% of a single CPU, it is unfair to take an elapsed time which includes a period when the CPU is underutilized in this way: in a multi-user situation, we would expect that several other queries were occurring simultaneously, accessing data from other disks, and keeping the CPUs utilized. Because of these considerations, our approach to calculating T is to calculate the time we could need to execute the set of queries if all the CPUs were well utilized, then increase the price of the benchmark system by buying more disks to allow this utilization in a multi-user system, and then base out $PRICE/QPS on these values.

The price calculation starts by summing the CPU and I/O resources appearing in each of the reported 69 query table rows of Section 2, resulting in the totals: TOT_{CPU} and $TOT_{I/O}$. We want to calculate an elapsed time period T during which the CPU(s) of the tested machine will be relatively fully utilized in running these queries. To calculate T, we start by identifying all queries measured where the CPU resource is clearly the gating factor, then we sum the CPU utilization for these queries, SUM_{CPU}, and also sum the elapsed time, SUM_{ELA}. Now we calculate the factor $F = SUM_{ELA}/SUM_{CPU}$, to represent the elapsed scale-up of measured CPU under optimal conditions. Now the elapsed period T needed to run all queries is calculated as $T = F \cdot TOT_{CPU}$.

Our assumption that I/O will not be a bottleneck in performance is based on the idea that we purchase enough disks and spread out the data in an even enough fashion that some query thread will always be able to run on all the CPUs while others are performing I/O. As with the price calculation of DebitCredit, we assume an I/O rate on the multiple disks which is half that of the peak measured I/Os of the benchmark. To do this, we select the queries of the benchmark which are clearly I/O limited, calculate $SUM'_{I/O}$ and SUM'_{ELA} for these queries, and take $P = SUM'_{I/O}/SUM'_{ELA}$ as the peak I/O rate on the disks used for the benchmark. Then $S = .25 \cdot P$ is the I/O service rate we will assume for each individual disk in production. (We are clearly not assuming that we will be able to utilize all the disks fully during production).

Next we calculate the number of disks which will be needed in our production configuration so that I/O will not be a bottleneck. The number of I/Os performed in the elapsed period T is $TOT_{I/O}$, and thus the I/O rate per second is $TOT_{I/O}/T$, and the number of disks we will have to purchase, which we represent by D, is therefore the smallest integer exceeding $(TOT_{I/O}/T)/S$. (If the number of disks needed to contain the BENCH table size for the target system exceeds this number, we take the larger number.) Now we calculate the price of the system for the configuration envisioned. Assuming five years of ownership, we add together the price of purchase, service maintenance, and all software licenses, to arrive at the price P. Finally, we calculate the number of queries K performed in the time period T (we will have K = 69 in the unweighted calculation above) and our final measure of \$PRICE/QPS is given by $\frac{P \bullet T}{K}$. This is the true \$PRICE/QPS value for the benchmark measured. However, as the size of the BENCH table scales up, the resources needed for the average query also increases in a (slightly more than) linear fashion. To compare results from a 10-million row table with those from a 1-million row table, we normalize the value \$PRICE/QPS in the 10-million row case by dividing by 10. Any benchmark results for a multi-million row table should also report the default results and optimal price results for a 1-million row table, so that analysts can compare with different platforms and see explicit scaleup factors on the platform measured.

To perform the above calculation for a custom system, we represent the set of queries which are performed during a time period T as some weighted set of the queries, with non-negative weights W_i. We can take this to mean that query i is performed W_i times during the time period. Then in calculating TOT_{CPU} and $TOT_{I/O}$, we multiply the weights by the corresponding CPU and I/O measures. The calculation of the factors F and P can be performed using the entire suite of queries, but the final calculation of D should use a value for K given by: $K = \Sigma_i W_i$. **??end??**

3.4 Configuration and Reporting Requirements - a Checklist

- The data should be generated in the precise manner given in Table 3.1; the first 10 rows should have the values shown in Table 2.1; counts of queries specified in Appendix A for the one-million row BENCH table should be matched by the data.

- The BENCH table should be loaded on the appropriate number of stable devices (rotating media disks), and the resource utilization for the load, together with any preparatory utilities to improve performance (reorganization of table structure, index creation, statistics gathering for Query Optimizer, etc.) should be reported. The rows and 13 indices should be loaded so that individual disk pages are 95% full. In general, when we refer to resource utilization in what follows, we are referring to elapsed time, CPU time, and disk I/Os (possibly, as with DB2, of more than one kind). For the load, we also require that the disk media space utilization be reported, together with the type of index utilization statistics reported here in Table 2.3.

- Naturally, care should be taken that the database system is properly tuned for nearly optimal access to the table; any deviation from this course should be explained.

- The exact hardware/software configuration used for the test should be reported (see first paragraph of Section 2.1), with vendor's MIPS rating, if known, disk models used, OS Version, and Database System Release number. If multiple CPUs exist in the hardware, the exact number of CPUs utilized during the single-user benchmark queries should be reported.

- The access strategy for each Query reported should be explained in the report if this capability is offered by the database system.

- The benchmark should be run from either an interactive or an embedded program and the Page Buffer and Sort Buffer (if separate) space in MBytes should be reported (e.g., 1200 4K Byte buffers, used for both buffering and sort). If a benchmark with memory resident data and index is being performed, a series of queries to populate the buffers should be performed in advance of any benchmark measurements, and the resource usage (elapsed time, CPU and I/Os) required should be reported. If the default setup as measured in Section 2 is being measured, it is necessary to assure that no large amount of useful information remains in Buffers between one Query and the next.

- All resource use should be reported for each of the queries, as reported here in Section 2.2.1 through Section 2.2.6. The overhead of taking statistics is difficult to measure, but any estimates by the vendor or software suppliers should be reported.

• The $/QPS should be calculated and reported as in Section 2.3, following the prescription in Section 3.3.1.

3.5 Concluding Remarks

Variation in resource use seen in Section 2 translates into large dollar price differences. The database industry is only beginning of the cycle of design improvements in the area of Set Query performance. We can expect major new product releases in the next several years as better architectural approaches are consolidated. A standard benchmark by which progress can be measured is therefore of the greatest importance.

Extensions to the Set Query benchmark have been suggested by a number of observers, and a multi-user Set Query benchmark is under study. Because of extremely long response time for some queries, it is difficult to choose a good acceptance criterion (such as the DebitCredit requirement of 95% transaction completion in one second).

ACKNOWLEDGEMENT.
It is appropriate to acknowledge here the debt that the current work owes to the Wisconsin benchmark [2], in particular in recognizing the importance of a 'synthetic' database and a spectrum of selectivity factors. The Set Query benchmark differs from the Wisconsin benchmark mainly in its emphasis on commercial applications. The author would also like to thank Richard Winter, and the editor, Jim Gray, for many valuable suggestions and insights in bringing the Set Query benchmark to its current form.

APPENDIX A - Counts of Rows Retrieved by Queries

What follows is a listing of that the number of rows selected in each of the queries measured, Q1 through Q6B. This is meant to serve as an aid to the researcher attempting to duplicate these results. In a few cases, other measures than number of rows selected are given

QUERY	CASE	ROW-COUNT	
Q1	KSEQ	1	
--	K100K	8	
--	K10K	98	
--	K1K	1,003	
--	K100	10,091	
--	K25	39,845	
--	K10	99,902	
--	K5	200,637	
--	K4	249,431	
--	K2	499,424	
Q2A	KSEQ	1	
--	K100K	5	
--	K10K	58	
--	K1K	487	
--	K100	5,009	
--	K25	19,876	
--	K10	49,939	
--	K5	100,081	
--	K4	125,262	
Q2B	KSEQ	499,423	
--	K100K	499,419	
--	K10K	499,366	
--	K1K	498,937	
--	K100	494,415	
--	K25	479,548	
--	K10	449,485	
--	K5	399,343	
--	K4	374,162	
			SUM
Q3A	K100K	1	434
--	K10K	9	5,513
--	K100	991	496,684
--	K25	3,989	1,978,118
--	K10	9,920	4,950,698
--	K5	2011?	10,027,345
--	K4	24,9??	12,499,521
Q3B	K100K	1	434
--	K10K	6	3,300
--	K100	597	299,039
--	K25	2,423	1,209,973
--	K10	5,959	2,967,225
--	K5	12,000	5,980,617
--	K4	15,031	7,496,733
Q4A	1-3	10,059	
--	2-4	4,027	
--	3-5	1,637	
--	4-6	4,021	
--	5-7	7,924	
--	6-8	10,294	

--	7-9	4,006	
--	8-10	785	
Q4B	1-5	161	
--	2-6	86	
--	3-7	142	
--	4-8	172	
--	5-9	77	
--	6-10	76	
--	7-1	152	
Q5	K2=K100=1	????	\ *** I WILL FIND CORRECT VALUE FOR THESE ?s ***
--	K4=K25=1	9970	| Counts for KN1 = 1 AND KN2 = 1
--	K10=K25=1	4049	/
Q6A	K100K	23	
--	K40K	55	
--	K10K	239	
--	K1K	2,014	
--	K100	19,948	
Q6B	K40K	3	
--	K10K	4	
--	K1K	81	
--	K100	804	

REFERENCES

[1] Anon et al., "A Measure of Transaction Processing Power", *Datamation*, April 1, 1985, p. 112.

[2] Bitton, D., DeWitt, D. J. and Turbyfill, C., "Benchmarking Database Systems: A Systematic Approach", Technical Report #526, C.S. Dept, U of Wisconsin, Dec. 1983 (this is an expanded version of the paper with this title in Proc. VLDB 1983.)

[3] Cheney, W. and Kincaid, D, *Numerical Mathematics and Computing*, Brooks/Cole Publishing Company, 1980.

[4] Gray, J. and Putzolu, F., "The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time", Proceedings of the 1987 ACM SIGMOD Conference, pp 395-398.

[5] O'Neil, P. E., "Model 204 Architecture and Performance", *High Performance Transaction Systems, Proceedings 1987*, Springer-Verlag Lecture Notes in Computer Science 359, 1989, p. 40-59.

[6] O'Neil, P. E., "Revisiting DBMS Benchmarks", *Datamation*, September 15, 1989, pp 47-52