An Analytical Method for Estimating and Interpreting Query Time

Hai-Yann Hwang
Bell Laboratories
Murray Hill, New Jersey 07974

Yao-Tin Yu
Bell Laboratories
Middletown, New Jersey 07748

Abstract

This paper presents a general methodology to decompose the processing of relational queries into functional units. Each unit consumes a constant CPU usage, which depends on the DBMS and system configuration, but not on the database or the query. We describe how to measure the unit CPU consumption, as well as how to use it to predict and interpret query time. Two DBMSs were tested to validate and calibrate the model. Its applications on DBMS design, database design, query performance and DBMS comparison are discussed.

1. Introduction

This paper presents a general methodology to analyze the CPU consumption of relational queries on the functional operation level (e.g., input, output, comparison). It attempts to address the following fundamental problem:

Given a query for certain database on certain database system, how much CPU time will each processing step consume?

This problem is a basic issue in many database research and practice arenas, including DBMS design, database and query design, DBMS comparison, system tuning and work scheduling.

Some effort has been made to attack this problem to various extents. For two-variable queries, [YAO 79] describes a general model consisting of processing steps such as indexing, record access, sorting, joining, projection, etc. Unfortunately, this model has not been validated. [HAWT 79] studied the percentage of CPU time that Ingres (university version, [STON 76]) spent in each of its five processes. But the technique only applies to the level of process. In [STON 83], the UNIX® "profile" package was utilized to get a procedure-level breakdown. However, profiling requires access to the source code, and the CPU distribution is by subroutine, instead of by functionality[1]. [MACK 86] presents a validated CPU cost model for the local query processing of R* [LOHM 85] (it also applies to System R [CHAM 81]). But the model is tightly geared to the internal structure of R and R*.

Our objective is to develop a CPU time model for query processing, that isolates functionally independent operations from one another. It can help us understand the underlying timing distribution, the relative weights, the influencing factors, and other dynamics of query processing. To be useful, the model should be as generic as possible with respect to various DBMSs, and can be calibrated readily with common user privilege, e.g., access to the source code should not be prerequisite.

This paper describes a model that decomposes query processing into *elementary operations*. It is assumed that each elementary operation consumes a fixed amount of CPU time (called *coefficient*), which is a parameter of the DBMS and system configuration, but independent of the database and the query. We show how to measure the CPU consumption coefficients. We tested the model on two DBMSs: Ingres and Informix. The tests confirm our assumption on the stability of coefficients. The calibrated model can be used to predicts queries' CPU time. Some applications based on this methodology are discussed.

® UNIX is a Trademark of Bell Laboratories

1. For example, if a program consists of three subroutines A, B and C, where both A and B call C. Profiling can provide the CPU usage and the number of calls of each subroutine. But it is hard to break down C's CPU consumption into the shares of A and B. Partitioning subroutines by functionality is very complicated, if not impossible, especially for big software packages. Profiling is almost useless in DBMS comparison, since each DBMS has its own subroutine structure.

This paper considers only *simple selection queries*, as defined in Section 2.1. This subset of queries covers a set of elementary operations which constitute the basis of more complicated query processing. It is these basic operations we'd like to focus first. General queries will be addressed in a subsequent paper currently being drafted. Their processing involves query optimization issues, which we'd like to defer to the second stage.

The paper is organized as follows. Section 2 presents the model. Section 3 describes how to calibrate it. Section 4 addresses how to use the calibrated model to predict the CPU time of a query. Section 5 discusses the model's potential applications. Section 6 concludes this paper.

2. A Model of Elementary Operation

2.1 Simple Selection Queries

A simple selection query is a query satisfying the following conditions:

1. The query involves only one relation;
2. Its qualification consists of one or zero non-indexed selection condition;
3. The query does not build new relation(s) or eliminate duplicate output tuples.

Some examples in QUEL are:

retrieve (emp.name, emp.dept) where emp.sal>50,000
retrieve (project.name, project.budget)

This paper will focus on simple selection queries, since they cover a set of "basic" elementary operations that are of critical importance to general query processing. The processing of simple selection queries is straightforward. For general queries that contain indexing, multiple selections, joins and relation build-up, the issue of query optimization and some complicated processing are involved. They will be addressed in another paper.

2.2 A Model of Elementary Operation

The most efficient processing strategy for a simple selection query should be:

1. Sequentially retrieve each page of the queried relation;
2. If a selection condition is involved, for each tuple, get the appropriate attribute and compare its value to the given constant;
3. For each qualifying tuple, get the attributes in the target list and output them.

To capture the above processing, we propose the following *elementary operations*:

1. Get a page (*get-page*)
2. Get a tuple (*get-tuple*)
3. Compare an attribute of a certain data type; for example,
   a. Compare a 2-byte integer (*cmp-i2*)
   b. Compare a 4-byte integer (*cmp-i4*)
   c. Compare a 4-byte floating point (*cmp-f4*)
   d. Compare a 1-byte character string (*cmp-c1*)
   e. Compare a character in a string (*cmp-char*)
4. Output a tuple (*out-tuple*)
5. Output an attribute of a certain data type; for example,
   a. Output a 2-byte integer (*out-i2*)
   b. Output a 4-byte integer (*out-i4*)
   c. Output a 4-byte floating point (*out-f4*)
   d. Output a 1-byte character string (*out-c1*)
   e. Output a character in a string (*out-char*)

We assume that for a given database system configuration, each elementary operation consumes a fixed amount of CPU time, which is independent of the database and query. This fixed CPU consumption is called the *coefficient* of that operation. A query can be coded into a vector of operation counts, called *query vector*. The total CPU consumption of a query is the sum of the operation counts in the query vector, each weighted by the corresponding coefficient.

2.3 Discussions on the Model

The above model was built up empirically. Some of its features are discussed below.

2.3.1 Data Access

For data access, two factors, i.e., page count and tuple count, are explicitly spelled out in the model. Another factor, *get an attribute*, is captured implicitly in attribute comparison and attribute output[2].

2.3.2 Get-Page

Get-page is a complex operation. It is rather simple if the operating system buffering is bypassed (i.e., raw disk). However, when operating system buffering is involved (as in most UNIX DBMSs that employ UNIX file system), disk access involves two steps:

1. reading data from disk to system buffer;
2. copying data from system buffer to user space.

If the page to be accessed exists already in system buffer, Step 1 will be skipped. Moreover, a DBMS may manage its own buffer pool (e.g., Ingres [STON 81]). This makes the scenario more complicated.

2. When an attribute is accessed more than once, *get an attribute* may be involved 1) for each access or 2) only once. The latter case will fail this implicit approach, especially when the coefficient for *get an attribute* is significant. We did not observe this effect in our test.

We choose to define *get-page* uniformly as the "full fetch" from disk to user space[3]. The buffering effect, if exists, could be estimated and discounted from the count of *get-page*, as illustrated in Section 4.

2.3.3 Numerical Comparison Condition

For numerical attribute comparison, the CPU usage is assumed to depend only on data type; but not affected by the identity of relational operator (e.g., be it "<", "=" or ">="), nor by the constant value to be compared with[4]. This conjecture is confirmed with test queries.

2.3.4 Character String Comparison and Output

For comparing character strings, the complexity is assumed to be linear in the number of bytes actually compared (which may be less than the string length), captured by elementary operations *cmp-char*. For outputting strings, the complexity is assumed to be linear in the number of output bytes, captured by elementary operations *out-char*. Test queries that confirm this conjecture are described in Sections 3.2.4 and 3.2.5.

2.3.5 Other Aspects

This model focuses on data processing. Other DBMS activities, e.g., query input, query parsing and query optimization, appear to consume negligible CPU in our measurements. To avoid the issue of concurrency control at this stage, DBMSs were set to lock at the database level, and all tests were run without other database users. There is a stable overhead in initializing DBMS.

In a multi-user environment, CPU usage may be impacted by total system load (due to job switching, timing granularity, etc). At this stage, this impact is not explicitly expressed in the model, but will be reflected in the coefficient measurement result.

3. Granularity other than page may be allowed in different stages of disk access. For instance, data can be copied from system buffer to user space by byte instead of a whole page. This approach may be beneficial if only limited bytes (e.g., a tuple) in a page have to be accessed. This byte-driven complexity will appear in operations such as *get an attribute* , instead of *get-page*. We did not observe significant byte-driven complexity for data access. It indicates the DBMSs we tested choose to move data by page, presumably due to the high overhead of each buffer access (see Section 5.1.1 and [STON 81]).

4. The constant is assumed to be a legitimate value for the attribute. Otherwise, a smart query optimizer may be able to detect and skip the comparison completely, as we found in Ingres.

3. Measurement of Coefficients

This section describes how to measure the coefficients of the above elementary operations. Section 3.1 addresses the general principles. Section 3.2 presents a design of test database and queries, illustrated with Ingres and the query language QUEL. There are many designs. These designs apply generally to relational DBMSs, but may need slight adjustment from one DBMS to another[5].

Section 3.3 reports our measurement for Ingres and Informix on a VAX 11/785 running BSD 4.3 UNIX operating system.

3.1 General Principles

The principle of coefficient measurement for elementary operation is to design a series of queries that isolate the impact of an operation, and amplify this impact to a measurable extent. All other influencing factors must be carefully controlled[6].

For measuring CPU time, timing tools provided by the operating system are more appropriate than those in DBMSs, since

1. the outcomes of different timing tools from different DBMSs may not be comparable.
2. many DBMSs, e.g., Informix, do not provide timing tools.

Since total system load affects CPU usage, it should be specified as a parameter of measurement, and then well controlled throughout the test. When CPU time shows fluctuation, queries should be run repeatedly to average out background noise. To minimize the impact of system load and noise, test queries should be designed to minimize constant overhead[7], but maximize the target processing that varies from query to query. The difference of query time in a series should be significantly larger than the background noise.

3.2 A Test Database and Query Set

This section describes a design of test database and queries, illustrated with Ingres and QUEL.

5. For example, since Ingres and Informix use different methods to allocate tuples into pages, the definition of relations *t1* to *t5* in Section 3.2.2 should be changed slightly as applying to Informix, such that the five relations still occupy the same number of pages.

6. For example, buffering may cause the discrepancy of the number of *get-page*, as discussed in Section 2.3.2. To avoid this adverse effect, queries that address small relations (compared to the size of system buffer) should be interleaved properly to ensure each page is freshly fetched from disk.

7. For example, query series for input (*get-page, get-tuple*) avoid generating output.

3.2.1 Get-Page

The coefficient for *get-page* can be measured with the following relations:

Relation	Attributes Name (Data Type[8])	Tuple Width	No. of Tuples	No. of Pages[9]
p1	s(c1),i(i4),v(c1)	6 bytes	64,000	256
p2	s(c1),i(i4),v(c33)	38 bytes	64,000	1280
p3	s(c1),i(i4),v(c73)	78 bytes	64,000	2561
p4	s(c1),i(i4),v(c121)	126 bytes	64,000	4268
p5	s(c1),i(i4),v(c153)	158 bytes	64,000	5335

Attribute *i* is populated with integers between 0 and 9. No index is built.

The test query series is as follows:

q.get-page.1: retrieve (p1.i) where p1.i>10
q.get-page.2: retrieve (p2.i) where p2.i>10
q.get-page.3: retrieve (p3.i) where p3.i>10
q.get-page.4: retrieve (p4.i) where p4.i>10
q.get-page.5: retrieve (p5.i) where p5.i>10

These five queries require the same number of tuple fetching (*#get-tuple*=64,000), the same number of attribute comparison (*#cmp-i4*=64,000), and generate no output. Their only difference resides in how many pages each query needs to retrieve. (since the length of attribute *v* varies.) We found the measured CPU consumption can be linearly correlated to the page count; the slope is taken as the coefficient for *get-page*.

3.2.2 Get-Tuple

One way to measure the coefficient of *get-tuple* is through the following five relations:

Relation	Attributes Name (Data Type)	Tuple Width	No. of Tuples	No. of Pages
t1	s(c1),i(i4),v(c1)	6 bytes	80,000	320
t2	s(c1),i(i4),v(c3)	8 bytes	64,000	320
t3	s(c1),i(i4),v(c13)	18 bytes	32,000	321
t4	s(c1),i(i4),v(c33)	38 bytes	16,000	321
t5	s(c1),i(i4),v(c150)	155 bytes	3,832	321

Notice that the tuple width and tuple counts are adjusted such that each relation occupies identical number of pages. Attribute *i* is populated with integers between 0 and 9. No index is built.

The test query series is as follows:

q.get-tuple.1: retrieve (t1.i) where t1.i>10
q.get-tuple.2: retrieve (t2.i) where t2.i>10
q.get-tuple.3: retrieve (t3.i) where t3.i>10
q.get-tuple.4: retrieve (t4.i) where t4.i>10
q.get-tuple.5: retrieve (t5.i) where t5.i>10

These five queries fetch the same number of pages (*#get-page*~320) and generate no output. For each query, the number of tuples to get (*#get-tuple*) is the number of tuples in the queried relation, as listed above, and so is the number of i4 comparison (*#cmp-i4*). We found the measured CPU time is linear in the tuple count. By subtracting the coefficient of *cmp-i4* (as measured in Section 3.2.3) from the slope, we can get the coefficient of *get-tuple*.

3.2.3 Attribute Comparison

The following relation can be used to measure the CPU usage for comparing an attribute:

Relation	Attributes Name (Data Type)	No. of Tuples
m	i2(i2),i4(i4),f4(f4) c1(c1),out(c1)	16,000

Each numerical attribute is populated with values evenly distributed between 0 and 9. Attribute *c1* is populated with strings between "0" and "9".

The test queries for comparing a numerical or 1-character attribute are:

q.dummy: retrieve (m.out)
q.cmp-i2: retrieve (m.out) where m.i2<10
q.cmp-i4: retrieve (m.out) where m.i4<10
q.cmp-f4: retrieve (m.out) where m.f4<10
q.cmp-c1: retrieve (m.out) where m.c1<"a"

Each query scans through the whole relation and outputs attribute *out* for each tuple. Attribute comparison is the only difference that the *comparing queries* perform in addition to query *q.dummy*. Subtracting the CPU usage of query *q.dummy* from that of each comparing query, then dividing the difference by the tuple count of relation *m*, the result is the coefficient of comparing an attribute of the corresponding data type.

3.2.4 Character Comparison

For character strings of various length, the following relation and queries can be used to check the relationship between comparison time and string length.

8. Denoted by one character for type ("c" for string, "i" for integer, "f" for floating), and the number of bytes.

9. In Ingres, each page has 2K bytes.

Relation	Attributes Name (Data Type)	No. of Tuples
c	c1(c1),c8(c8),c16(c16) c32(c32),c64(c64),out(c1)	16,000

For each attribute, all the bytes are set to "0", except the last byte, which is set to between "0" and "9". This forces each comparison in the following queries go to the last byte of the attribute.

q.cmp-c01: retrieve (c.out) where c.c1<"a"
q.cmp-c08: retrieve (c.out) where c.c8<"0000000a"
q.cmp-c16: retrieve (c.out) where c.c16<
"000000000000000a"
q.cmp-c32: retrieve (c.out) where c.c32<
"00000000000000000000000000000000a"
q.cmp-c64: retrieve (c.out) where c.c64<
"0000000000000000000000000000000000000000
00000000000000000000000000000000a"

Each of the above queries involves the same input and no output, but their character comparison counts vary with attribute length. We observed a linear relationship between the CPU time and the string byte count. The slope, after being normalized by tuple count, is the time required for comparing an additional character, i.e., the coefficient of *cmp-char*.

3.2.5 Output Attribute

The measurement of the coefficients for output an attribute can use relation *m* and the following queries:

q.dummy: retrieve (m.out)
q.out-i2: retrieve (m.out,m.i2)
q.out-i4: retrieve (m.out,m.i4)
q.out-f4: retrieve (m.out,m.f4)
q.out-c1: retrieve (m.out,m.c1)

The strategy is analogous to that of attribute comparison.

In parallel to the linearity of character string comparison, we found that the CPU usage for outputting a character string is proportional to its output length. The queries used are as follows:

q.out-c01: retrieve (c.c1)
q.out-c08: retrieve (c.c8)
q.out-c16: retrieve (c.c16)
q.out-c32: retrieve (c.c32)
q.out-c64: retrieve (c.c64)

The coefficient of *out-char* can be computed similarly to that of *cmp-char*.

DBMS' output formatting policy varies substantially. For example, Ingres has default output format for each data type, which users can override. In Informix, the output format varies on data type and the length of attribute name. When output width exceeds roughly 80 characters, Informix switches from tabular output to a vertical representation consisting of one name-value pair for each attribute, whose complexity is quite different from that of tabular output. Those factors have to be carefully controlled to ensure the format used coincides with the format targeted.

We found that for each data type, the CPU usage for output shows a linear dependency on the byte count of output format, and the slope is equal to the coefficient of *out-char*. This implies outputting an attribute involves some type-dependent conversion, plus some type-independent outputting. The latter is proportional to the number of characters, with a coefficient common to all data types.

3.2.6 Out-Tuple

To measure the coefficient of *out-tuple*, the following queries can be used:

q.out-tuple.00: retrieve (m.i4) where m.i4<0
q.out-tuple.01: retrieve (m.i4) where m.i4<1
q.out-tuple.02: retrieve (m.i4) where m.i4<2
q.out-tuple.04: retrieve (m.i4) where m.i4<4
q.out-tuple.08: retrieve (m.i4) where m.i4<8
q.out-tuple.10: retrieve (m.i4) where m.i4<10

Each query involves identical input and comparison. However, since the attribute *i4* is evenly distributed between 0 and 9, the number of output tuples increases in the sequence 0, 1600, 3200, 6400, 12800 and 16000. Linearly regressing the CPU time of the queries against the sequence for output tuple count, the slope should be the sum of *out-tuple* and *out-i4*, from which the coefficient of *out-tuple* can be derived easily.

3.3 Coefficient Measurement and Consistency

Table 1 lists the coefficients we measured for two DBMSs: Ingres 4.0 and Informix 2.00, on VAX 11/785 running BSD 4.3 UNIX operating system. The measurement was run in a semi-single-user environment[10].

We used the UNIX *time* command [UURM 86] to measure the elapsed time and CPU time. All queries[11] were run 10 times. The resultant CPU usage was averaged.

To test the stability of the coefficients, we conducted the following cross checking:

10. It was actually multi-user environment during off-peak time, with few other users competing for CPU, and no other database users. All test queries can achieve a CPU utilization of 95%. As utilization rate dropped to as low as 40%, CPU time could vary 10-15%. The data reported in this paper reflect a screened result with CPU utilization above 80%, which shows stable consistency.

11. Each query is encapsulated in a UNIX shell program that initializes the DBMS.

1. For each regression, conducting the goodness of fit test. It indicates the linear regression line fit the data adequately.
2. Repeating the 10-run measurement. Each operation was measured at least five times.
3. Varying query design. For example, for *get-page*, *get-tuple* and *out-tuple*, attributes other than *i* were used to generate parallel query series for test.
4. Varying database design, including relation definition and size.
5. Checking the coefficients that are mutually related. For example, if the model is valid, the following relationships should hold:

   cmp-c1 + (8-1) * cmp-char = cmp-c8

   out-c1 + (8-1) * out-char = out-c8

Throughout the above checking, the observed discrepancy is within 10%, which confirms our assumption about the consistency of coefficient. 10

Table 1. The Coefficients of Elementary Operations for Ingres 4.0 and Informix 2.00 (on VAX 11/785 running BSD 4.3 UNIX operating system)

Elementary Operation	Coefficient (μsec) Ingres	Informix	Notes	
get-page	5122.2	3056.6	Page size: 2KB for Ingres, 1KB for Informix	
(norm.)	2561.1	3056.6	Normalized to 1KB page	
get-tuple	244.2	805.0		
cmp-i2	123.0	749.3		
cmp-i4	118.1	478.4		
cmp-f4	115.0	865.9		
cmp-c1	252.8	461.3		
cmp-c8	383.0	462.1		
cmp-char	17.5	0.3		
out-tuple	550.0	2219.9		
			Out Bytes	Vert. Coef[12]
out-i2	820.7	625.1	6	653.3
out-i4	1277.4	723.7	11	674.2
out-f4	996.4	972.7	9	890.2
out-c1	230.7	186.8	1	294.1
out-c8	903.8	390.4	8	395.3
out-char	95.9	24.5		14.1
overhead	6.0*sec*	1.9*sec*	for initializing DBMS	

12. Coefficients for Informix vertical output format (Section 3.2.5); Each output includes 1 byte for data and 3 bytes for attribute name, except *out-c8*, whose data length is 8 byte. *#out-char* has to be adjusted for the real length of data and attribute name.

4. Estimating CPU Usage of Queries

The model of elementary operation assumes the CPU consumption of a query is the summation of the counts of elementary operations it performs, each multiplied by the coefficient of that operation. Once the coefficients are measured, we can predict a query's CPU usage from its operation count vector (i.e., *query vector*).

Coding a simple selection query into query vector is straightforward. However, the following operations need special attention:

Get-page: If no buffering or data sharing is involved, the operation count for *get-page* should be the number of pages in the queried relation. However, as mentioned in Section 2.3.2, if pages can be fetched from a buffer, the *get-page* count has to be adjusted accordingly.

Example 1. For query *q.s5n* (described below), Informix has to retrieve 3910 pages (Table 4). If 10% of them are retrieved from the buffer, then 0.6 *msec* (CPU time for reading a 1K page from disk to main memory, see Section 5.1) can be saved from the 3.1 *msec* (Informix' coefficient of *get-page*) procedure for each of those pages. Hence the actual count for *get-page* should be

$$3910 * 90\% + 3910 * 10\% * (1 - \frac{0.6}{3.1}) = 3832$$

Cmp-char: The number of characters actually compared for a string has to be estimated by the distribution of data.

Output attribute: As mentioned in Section 3.2.5, the format of attribute output varies on many factors. If the format chosen by the query differs from that used in coefficient measurement, either the coefficients, or the count of *out-char*, should be adjusted accordingly.

Example 2. For query *q.s5w* (described below) which takes Ingres default output format, Ingres outputs the 30 *i2* attributes with 6 bytes, the only *i4* attribute with 13 bytes, and each of the 87 character string attributes with a minimum width 6 bytes. Each string attribute contributes 1 to the count of *out-c1*. The remaining bytes (5 per attribute), plus the 2 extra bytes that the *i4* field outputs, can be put in the count of *out-char* as

$$\#out\text{-}char = (6 - 1) * 87 + (13 - 11) * 1 = 437$$

If the 6-byte threshold of string output is overridden, simply by specifying a parameter, most of the *out-char* operations (accounting for 40% of CPU time) in queries *q.s5w*, *q.s6w* and *q.s7w* can be saved.

We checked the correctness of query time prediction extensively. The relative error is generally less than

Table 2. The Calculated and Observed CPU Usage of Queries on Ingres

DBMS		Ingres											
Query		q.s1n	q.s2n	q.s3n	q.s4n	q.s1i	q.s2i	q.s3i	q.s4i	q.s1w	q.s2w	q.s3w	q.s4w
q v e c t o r	#get-page [13]	1001	1001	1001	1001	1001	1001	1001	1001	1001	1001	1001	1001
	#get-tuple	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
	#cmp-i2 (per get-tuple)	1	1	1	0	1	1	1	0	1	1	1	0
	#out-tuple	100	1000	10000	10000	100	1000	10000	10000	100	1000	10000	10000
	#out-c1 (per out-tuple)	1	1	1	1	0	0	0	0	3	3	3	3
	#out-char (per out-tuple)	51	51	51	51	0	0	0	0	153	153	153	153
	#out-i2 (per out-tuple)	2	2	2	2	4	4	4	4	13	13	13	13
t[14] i m e s e c	get-page	5.13	5.13	5.13	5.13	5.13	5.13	5.13	5.13	5.13	5.13	5.13	5.13
	get-tuple	2.44	2.44	2.44	2.44	2.44	2.44	2.44	2.44	2.44	2.44	2.44	2.44
	cmp-i2	1.23	1.23	1.23	0	1.23	1.23	1.23	0	1.23	1.23	1.23	0
	out-tuple	0.06	0.55	5.50	5.50	0.06	0.55	5.50	5.50	0.06	0.55	5.50	5.50
	out-c1	0.02	0.23	2.31	2.31	0	0	0	0	0.07	0.69	6.92	6.92
	out-char	0.49	4.89	48.88	48.88	0	0	0	0	1.47	14.67	146.65	146.65
	out-i2	0.16	1.64	16.41	16.41	0.33	3.28	32.83	32.83	1.07	10.67	106.69	106.69
	subtotal	9.53	16.11	81.90	80.67	9.18	12.63	47.13	45.90	11.46	35.38	274.56	273.33
	initial. overhead	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
	calculated time	15.53	22.11	87.90	86.67	15.18	18.63	53.13	51.90	17.46	41.38	280.56	279.33
	observed time	14.30	21.15	87.69	86.43	13.86	17.51	53.16	53.95	16.22	40.79	275.49	281.28
	absolute error	1.23	0.96	0.21	0.24	1.32	1.12	-0.03	-2.05	1.24	0.59	5.07	-1.95
relative error		8.60%	4.54%	0.24%	0.28%	9.54%	6.41%	-0.06%	-3.80%	7.63%	1.44%	1.84%	-0.69%

Table 3. The Calculated and Observed CPU Usage of Queries on Informix[15]

DBMS		Informix											
Query		q.s1n	q.s2n	q.s3n	q.s4n	q.s1i	q.s2i	q.s3i	q.s4i	q.s1w	q.s2w	q.s3w	q.s4w
q v e c t o r	#get-page [13]	1831	1831	1831	1831	1831	1831	1831	1831	1831	1831	1831	1831
	#get-tuple	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
	#cmp-i2 (per get-tuple)	1	1	1	0	1	1	1	0	1	1	1	0
	#out-tuple	100	1000	10000	10000	100	1000	10000	10000	100	1000	10000	10000
	#out-c1 (per out-tuple)	1	1	1	1	0	0	0	0	3	3	3	3
	#out-char (per out-tuple)	51	51	51	51	0	0	0	0	250	250	250	250
	#out-i2 (per out-tuple)	2	2	2	2	4	4	4	4	13	13	13	13
t[14] i m e s e c	get-page	5.60	5.60	5.60	5.60	5.60	5.60	5.60	5.60	5.60	5.60	5.60	5.60
	get-tuple	8.05	8.05	8.05	8.05	8.05	8.05	8.05	8.05	8.05	8.05	8.05	8.05
	cmp-i2	7.49	7.49	7.49	0	7.49	7.49	7.49	0	7.49	7.49	7.49	0
	out-tuple	0.22	2.22	22.20	22.20	0.22	2.22	22.20	22.20	0.22	2.22	22.20	22.20
	out-c1	0.02	0.21	2.08	2.08	0	0	0	0	0.09	0.88	8.82	8.82
	out-char	0.13	1.25	12.51	12.51	0	0	0	0	0.35	3.50	35.00	35.00
	out-i2	0.13	1.25	12.50	12.50	0.25	2.50	25.00	25.00	0.85	8.48	84.80	84.80
	subtotal	21.63	26.07	70.43	62.94	21.61	25.86	68.34	60.85	22.65	36.22	171.96	164.47
	initial. overhead	1.87	1.87	1.87	1.87	1.87	1.87	1.87	1.87	1.87	1.87	1.87	1.87
	calculated time	23.50	27.94	72.30	64.81	23.48	27.73	70.21	62.72	24.52	38.09	173.83	166.34
	observed time	21.31	25.86	67.84	59.95	21.39	25.66	64.77	54.60	25.77	39.12	177.88	168.70
	absolute error	2.19	2.08	4.46	4.86	2.09	2.07	5.44	8.12	-1.25	-1.03	-4.05	-2.36
relative error		10.29%	8.04%	6.57%	8.10%	9.78%	8.07%	8.40%	14.87%	-4.86%	-2.63%	-2.28%	-1.40%

Table 4. The Calculated and Observed CPU Usage of Queries on Ingres and Informix[15]

DBMS		Ingres						Informix					
Query		q.s5n	q.s6n	q.s7n	q.s5w	q.s6w	q.s7w	q.s5n	q.s6n	q.s7n	q.s5w	q.s6w	q.s7w
query vector	#get-page[13]	1869	1869	1869	1869	1869	1869	3910	3910	3910	3910	3910	3910
	#get-tuple	22338	22338	22338	22338	22338	22338	22338	22338	22338	22338	22338	22338
	#cmp-i2 (per get-tuple)	1	1	0	1	1	0	1	1	0	1	1	0
	#cmp-i4 (per get-tuple)	0	0	1	0	0	1	0	0	1	0	0	1
	#out-tuple	1383	9464	22335	1383	9464	22335	1383	9464	22335	1383	9464	22335
	#out-c1 (per out-tuple)	8	8	8	87	87	87	8	8	8	87	87	87
	#out-char (per out-tuple)	40	40	40	437	437	437	40	40	40	500	500	500
	#out-i2 (per out-tuple)	3	3	3	30	30	30	3	3	3	30	30	30
	#out-i4 (per out-tuple)	0	0	0	1	1	1	0	0	0	1	1	1
time (sec)[14]	get-page	9.57	9.57	9.57	9.57	9.57	9.57	11.95	11.95	11.95	11.95	11.95	11.95
	get-tuple	5.45	5.45	5.45	5.45	5.45	5.45	17.98	17.98	17.98	17.98	17.98	17.98
	cmp-i2	2.75	2.75	0	2.75	2.75	0		16.74	16.74	0	16.74	16.74
	cmp-i4	0	0	2.64	0	0	2.64	0	0	10.69	0	0	10.69
	out-tuple	0.76	5.21	12.28	0.76	5.21	12.28	3.07	21.01	49.58	3.07	21.01	49.58
	out-c1	2.55	17.47	41.22	27.76	189.95	448.28	2.30	15.73	37.13	35.39	242.15	571.48
	out-char	5.30	36.29	85.63	57.93	396.41	935.53	1.36	9.29	21.92	9.68	66.25	156.35
	out-i2	3.41	23.30	54.99	34.05	233.01	549.91	2.59	17.75	41.88	27.06	185.20	437.07
	out-i4	0	0	0	1.77	12.09	28.53	0	0	0	0.93	6.38	15.06
	subtotal	29.80	100.03	211.79	140.04	854.45	1992.21	55.99	110.45	191.13	122.80	567.66	1270.16
	initial. overhead	6.00	6.00	6.00	6.00	6.00	6.00	1.87	1.87	1.87	1.87	1.87	1.87
	calculated time	35.80	106.03	217.79	146.04	860.45	1998.21	57.86	112.32	193.00	124.67	569.53	1272.03
	observed time	34.30	104.00	214.00	141.00	838.00	1894.00	52.70	97.60	166.50	128.00	594.00	1314.00
	absolute error	1.50	2.03	3.79	5.04	22.45	104.21	5.16	14.72	26.50	-3.33	-24.47	-41.97
relative error		4.36%	1.96%	1.77%	3.58%	2.68%	5.50%	9.79%	15.08%	15.92%	-2.60%	-4.12%	-3.19%

13. Each Ingres page is 2KB, whereas Informix page is 1KB. These are the page sizes used for measuring the coefficients of *get-page*, see Table 1.

14. These are the total CPU time (in seconds) taken by each elementary operation (calculated by multiplying the coefficient with the count of operation) or specified catagory.

15. Informix' query series *w* uses vertical format for output (see Section 3.2.5).

15%. Tables 2, 3 and 4 show some results, based on the following relations (with no indices) and queries[16].

Relation	Tuple Width	No. of tuples	Total Size
tenKtup1	182 bytes	10,000	1.8MB
attributes: 13 *i2* fields, e.g., *uniqu1,uniqu2,twofour* 3 *c52* fields, e.g., *stringu1*			
customer	160 Bytes	22,338	5.6MB
attributes: 30 *i2* fields, e.g., *cid,count,size* 1 *i4* field, i.e., *usage* 76 *c1* fields, e.g., the rest fields in *q.s5n* 2 *c2* fields, 6 *c3* fields, 3 *c4* fields			

Queries[17]:
Range of t is tenKtup1

q.s1w: retrieve (t.all) where t.uniqu2<101
q.s2w: retrieve (t.all) where t.uniqu2<1001
q.s3w: retrieve (t.all) where t.uniqu2<10001
q.s4w: retrieve (t.all)

q.s1n: retrieve (t.uniqu1,t.uniqu2,t.stringu1)
where t.uniqu2<101
q.s2n: retrieve (t.uniqu1,t.uniqu2,t.stringu1)
where t.uniqu2<1001
q.s3n: retrieve (t.uniqu1,t.uniqu2,t.stringu1)
where t.uniqu2<10001
q.s4n: retrieve (t.uniqu1,t.uniqu2,t.stringu1)

q.s1i: retrieve (t.uniqu1,t.uniqu2,t.two,t.four)
where t.uniqu2<101
q.s2i: retrieve (t.uniqu1,t.uniqu2,t.two,t.four)
where t.uniqu2<1001
q.s3i: retrieve (t.uniqu1,t.uniqu2,t.two,t.four)
where t.uniqu2<10001
q.s4i: retrieve (t.uniqu1,t.uniqu2,t.two,t.four)

Range of c is customer

q.s5w: retrieve (c.all) where c.count=1
q.s6w: retrieve (c.all) where c.size<4
q.s7w: retrieve (c.all) where c.usage<=100

q.s5n: retrieve (c.cid,c.count,c.size,c.index1,c.index2,
c.index3,c.index4,c.level1,c.level2,c.level3,c.level4)
where c.count=1
q.s6n: retrieve (c.cid,c.count,c.size,c.index1,c.index2,
c.index3,c.index4,c.level1,c.level2,c.level3,c.level4)
where c.size<4
q.s7n: retrieve (c.cid,c.count,c.size,c.index1,c.index2,
c.index3,c.index4,c.level1,c.level2,c.level3,c.level4)
where c.usage<=100

Notice that in Informix, query series *w* takes the vertical output format, hence its timing may not be directly comparable to that of Ingres. Even for queries that do use the same output format, the underlying mechanism may be quite different for the two DBMSs. For example, for queries *q.s5n*, *q.s6n* and *q.s7n*, both Ingres and Informix output 6 bytes for each of the 8 *c1* fields. But for Ingres, it is due to Ingres' default 6-byte threshold in outputting string field; whereas in Informix, it is because each field has a 6-byte name. Manipulating output format can generate drastic performance change in output-intensive queries, as shown in Example 2.

5. Applications

Elementary operation analysis uncovers the microscopic dynamics of query processing. The coefficients of elementary operations measure how fast each processing step is. The breakdown of query time indicates the relative significance of different functions. The capability of predicting query time based on query specification allows users to forecast how long a query would take without running it. In this section we briefly discuss some applications of this technique.

5.1 DBMS Design

The model of elementary operation is useful in analyzing the strengths and weaknesses of DBMS design. The following are some observations about Ingres and Informix, based on Table 1 and some C program tests[18].

5.1.1 Page Retrieval

Both Ingres and Informix need 2.5 to 3 *msec* to retrieve a 1K page. As a comparison, it takes 0.6 *msec* for a C program to call *read* (a UNIX system call [UPRM 86]), *read* takes another 0.6 *msec* to fetch a page of 1KB from disk to system buffer, and another 0.4 *msec* to move 1KB from system buffer to user space; it is 1.6 *msec* in total.

16. Relation *tenKtup1* and relevant queries are based on Wisconsin Benchmark [BITT 83], with some minor modifications. Relation *customer* is from a real-life database.

17. The *w* suffix in query names means *wide output*, *n* for *narrow*, and *i* for *integer*. The system buffer was flushed before each query ran.

18. Run on VAX 11/785 (1.4 MIPS) with BSD 4.3 UNIX operating system.

5.1.2 Character String Comparison

Ingres and Informix appear to reflect quite different designs. Informix' CPU consumption is relatively high, whereas almost insensitive to string length (up to 64 bytes). Ingres' overhead is lower, but grows at a rate of 17.5 *μsec* per character[19]. The breakeven point for the two DBMSs resides at roughly 13 characters. The tradeoff of the two designs should be judged by usage frequency analysis.

As a comparison to the above rates, two C programs were tested: program *CMP-BYTE* explicitly compares strings byte by byte [KERN 78], it takes 6 *μsec* of CPU time per character; program *CMP-STR* calls the C library subroutine *strcmp* ([UPRM 86]) that uses the *long comparison instruction*, it needs 2.5 *μsec* per character.

5.1.3 Character Output

Ingres' character output rate (96 *μsec* per byte) is very slow. This drags down all the attribute output coefficients. Informix takes 25 *μsec* to output a character.

Both rates include:
— buffering for piping;
— piping data from backend process to frontend process (Ingres sends messages in the unit of 1024 bytes [STON 83]);
— outputting data.

We also tested some C programs that perform similar functions. It takes 1.4 *msec* CPU time to send (write and read) a 1-byte message through pipe, whereas 2.5 *msec* to send a 1024-byte message (averaged 2.5 *μsec* per byte). Outputting data in the unit of 1024 bytes takes 2.3 *μsec* per byte.

5.1.4 Summary on DBMS Design

In the above we checked some DBMS processing rates and the rates of C programs that perform similar functions. It appears those DBMSs still have room for enhancement.

5.2 Database and Query Design

Elementary operation analysis provides users with query time prediction, the breakdown of time spent by individual operations, and how it varies as database and/or query changes. These capabilities can be used to measure the performance impact of a design decision. Some examples are given below.

5.2.1 Database Design

Table 1 enables users to judge whether to set an attribute as 2-byte integer, 4-byte integer, or character string. The impact can be estimated for various processing aspects, e.g., input, comparison, output, then weighed with their relative importance, along with other factors such as storage cost.

For logical and physical database design, elementary operation analysis is useful too. We'll discuss the issue in the paper that addresses complex queries.

5.2.2 Query Design

For query and application design, users can investigate the driving factors behind the query performance and write more efficient queries. For example, Tables 2 to 4 illustrate for the DBMSs that we tested, how sensitive the query time is to the selectivity (e.g., query series *q.s1* vs. *q.s2* vs. *q.s3*), to the output attribute(s) (e.g., query series *n* vs. *i* vs. *w*), and even to the output format (as explained in Section 4).

5.3 DBMS Comparison

The discussion in the above sections has already involved the issue of DBMS comparison. This section will summarize it and compare this analytical method with benchmarking.

Elementary operation analysis reveals the relative rate of each generic operation. It also can be used to estimate the CPU time for a given benchmark. We do not think it can replace benchmarking completely since the analysis of complex queries is not trivial, as we'll show in another paper. However, the analysis is valuable for benchmark design and result analysis. It can address problems such as:

1. How to characterize an application?
   In addition to the crude category of CPU or I/O intensive, elementary operation analysis provides finer classification based on data processing.
2. How to design a query set that properly benchmarks an application? What are the key factors to control?
3. If a target application involves big databases and/or time-consuming queries, how to "mimic" it with cheaper installation and still get correct comparison?
4. How to interpret the benchmark result?

The discussion in Sections 5.1 and 5.2 has already covered these problems. It is interesting to notice that elementary operation analysis can help benchmark design to focus on the difference of the DBMSs tested. For example, from Table 1 we find that Ingres is 2-3 times

19. Ingres actually provides different functionality: it skips blanks in comparison, e.g., strings "AB", "A B" and "A B " would all match. Thus this rate contains extra processing.

faster than Informix in getting tuple, outputting tuple and numerical comparison, whereas Informix is 3 times faster than Ingres in outputting character. Thus a benchmark designed for these two DBMSs has to pay special attention to these processing steps and ensure they are properly modeled as in the target application. Different combinations of processing steps can lead to contrary comparisons, such as in query pair *q.s3n* vs. *q.s3i*, or query pair *q.s3n* vs. *q.s1n*. For each pair, Informix outperforms Ingres on the first query whereas Ingres outperforms on the second. This also illustrates how important it is to properly interpret benchmark results.

For mimicking application, since elementary operation analysis resolves a query into a query vector, big database/queries can be simulated by small ones with prorated query vectors.

In summary, elementary operation analysis provides decomposition by processing function, guidelines for designing customerized benchmarks, and capability of result interpretation. These are exactly what the conventional benchmarking methodology lacks for. Our analysis indicates it can be quite misleading to apply generalized benchmarks (e.g., [BITT 83, BOGD 83]) to all applications and environments. For the DBMSs that we tested, the following basic design assumptions of [BITT 83] are not true[20];

1. Relative performance of DBMSs is the same for integer comparison and string comparison;
2. it is adequate to test selectivity factor with 1% and 10% [BORA 84].

6. Conclusion

This paper describes an empirical model for decomposing relational query processing into individual functional components, called *elementary operations*. The processing of a query can be decoded into a vector of the counts of elementary operations, where each operation takes a fixed amount of CPU time, dependent on DBMS configuration only. This method lends itself to query time prediction and interpretation, as well as microscopic study of query processing mechanism. These capabilities can be used to enhance our understanding in many theoretical and practical database fields.

Discussing *simple selection queries* only, this paper is aimed primarily at elementary operations for input, comparison and output. Complicated query processing plans are built with these basic operations. For example, indexed selection employs input (and comparison) operations to get through the directory structure; each pass of a sort-merge join consists of input, comparison and output (or temporary file building). Those queries will be addressed in a subsequent paper.

20. Another major drawback of [BITT 83] is its focusing on *retrieve into*, whose complexity is drastically different from data outputting, as we'll discuss in another paper.

7. Acknowledgements

We would like to thank John Walden and Ann Martin for their help.

8. References

[ASTR 76] Astrahan, M., et al., "System R: A Relational Approach to Database Management", ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976.

[BITT 83] Bitton, D., Dewitt,, D., and Turbyfill C., "Benchmarking Database Systems - A Systematic Approach", Technical Report #526, Computer Science Department, University of Wisconsin-Madison, December 1983.

[BOGD 83] Bogdanowicz, R., Crocker, M., Hsiao, D., Ryder, C., Stone, V., and Strawser, P., "Experiments in Benchmarking Relational Database Machines", Database Machines, Springer-Verlag, 1983.

[BORA 84] Boral, H., and Dewitt, D., "A Methodology for Database System Performance Evaluation", Proceedings of 1984 SIGMOD Conference, Boston, MA, June 1984.

[CHAM 81] Chamberlin, D., et al., "Support for Repetitive Transactions and Ad Hoc Queries in System R", ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[HAWT 79] Hawthorn, P., and Stonebraker, M., "Use of Technological Advances to Enhance Data Base Management System Performance", Proceedings of 1979 ACM-SIGMOD Conference on the Management of Data, Boston, MA, June 1979.

[LOHM 85] Lohman, G., et al, "Query Processing in R*", Query Processing in Database Systems, Springer-Verlag, 1985.

[MACK 86] Mackert, L., and Lohman, G., "R* Optimizer Validation and Performance Evaluation for Local Queries", Proceedings of 1986 SIGMOD Conference, Washington D. C., May 1986.

[KERN 78] Kernighan, B., Ritchie, D., "The C Programming Language", Prentice-Hall, 1978, p.101.

[STON 76] Stonebraker, M., Wong, E., Kreps, P., and Held, G., "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976.

[STON 81] Stonebraker, M., "Operating System Support for Database Management", Communication of the ACM, Vol. 24, No. 7, July 1981.

[STON 83] Stonebraker, M., et al., "Performance Enhancements to a Relational Database System", ACM Transactions on Database Systems, Vol. 8, No. 2, June 1983.

[UPRM 86] "UNIX Programmer Reference Manual", 4.3 Berkeley Software Distribution, April 1986.

[UURM 86] "UNIX User Reference Manual", 4.3 Berkeley Software Distribution, April 1986.

[YAO 79] Yao, S., "Optimization of Query Evaluation Algorithms" ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.