Analyzing and Improving the Performance of POSTGRES

Peter K. Lai

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

During the development of POSTGRES, little concern has been directed at improving performance. A project was carried out to measure the actual performance of POSTGRES, determine where it is slow, and improve system performance. The project was divided into 3 subtasks. The first was to obtain profile data and optimize the system according to those results. This resulted in improvement in speed by a factor of three. In the second subtask, POSTGRES was benchmarked using the Wisconsin Benchmark and the TP1 Benchmark. By fixing various problems discovered during the benchmarking phase, the system's TP1 rating was doubled. The third subtask involved deriving tests to verify the decisions made by the optimizer when choosing among various scan methods and join methods. Loop-holes found in the optimizer were corrected. Various suggestions were given to the development team for enhancing both the performance and the features of POSTGRES.

1. Introduction

POSTGRES [7] is a next-generation extensible relational database management system being developed at the University of California. Since its design was published in early 1986, much effort has been expended to enhance the design as well as to implement the system. As of early 1990, a large portion of POSTGRES is operational. The parser, optimizer, executor, buffer manager, various access methods, and communication routines are completed. Most rules described in the early design are supported by the rule manager and the query rewrite system. POSTQUEL and C functions are also supported. The main design goals of the system are fulfilled. However, little concern has been directed at improving the performance of POSTGRES. The poor performance of POSTGRES makes it impractical for any really useful work except as a research protocol. As a result, a performance analysis of POSTGRES was conducted to discover the causes of its poor performance and to improve that performance.

The main goal of this performance analysis project is to pinpoint the parts of the code that slow the system down and to either improve that code or make suggestions for improvement to the development team. Though the operating system may also be partially responsible for slowing down the DBMS, it was treated as a "given". Except for changes in adjustable operating system parameters, modifications were limited to POSTGRES itself. Besides speeding up the system, the project is also intended to benchmark POSTGRES to better assess its performance and to provide a common ground for comparisons with other DBMS's as well as a reference for future work on POSTGRES itself.

To achieve these goals, the project was divided into three subtasks. The first was to obtain a profile of execution times for all routines in POSTGRES running a fixed set of queries and to optimize routines that take up too much execution time. The second subtask was to benchmark POSTGRES using the Wisconsin Benchmark and the TP1 Benchmark, and to compare the performance of POSTGRES with that of the university version of INGRES. (INGRES is a relational database management system implemented at the University of California during the period 1975-1977 [8].) The last subtask was to derive and carry out a set of tests to ensure that the optimizer arranges reasonable execution plans to avoid degradation of performance.

The rest of the report is organized as follows. Section 2 describes the approaches and methodology employed in completing the subtasks. In Section 3, the results of the project are presented and explained. Solutions for the problems discovered through out the project are suggested in Section 4. Section 5 is a summary.

2. Methodology

This section describes the approaches and methods used to analyze and improve the performance of POSTGRES and to do the benchmarking. It is further divided into four sections. Section 2.1 describes the environment in which all the tests were performed. In Section 2.2, the profiling work is presented. Section 2.3 explains what benchmarks were chosen and why they were used. Section 2.4 gives the tests used to search for faults in the optimizer.

Before going into the details of the approaches and methods used, it is worthwhile to point out the differences between the profiling work and the benchmarking work. Although both profiling and benchmarking were done on the same databases with similar queries, their goals, the ways of carrying out the subtasks, and the data collected were completely different. The profiling work was aimed at finding out the breakdown of execution time among the routines and the subsystems so that optimizations could be made accordingly. As a result, only the distribution of execution time and the number of times each routine was called were of interest. On the other hand, the benchmarking work measured the performance of POSTGRES on well-known benchmarks so that the system could be compared with other DBMSs. The execution time for the whole system instead of for each routine was recorded. Moreover, the absolute execution time rather than the distribution of execution time was important. The two subtasks were carried out parallelly and repeatedly throughout the project.

2.1 Test Environment

Currently, POSTGRES runs on DECstations 3100 running ULTRIX, SUN workstations running SUN OS 3.5, Sparc stations running SUN OS 4.0.3, and Sequent parallel machines running DYNIX(R) V3.0.17. The work for comparing the performances of POSTGRES and university INGRES on the Wisconsin Benchmark was done on a SUN 3/280 workstation. All other tests done in this project were run in single user mode on a DECstation 3100. The DECstation 3100 used has a 10 MIPS processor, 16M of main memory, and is equipped with a local disk which can handle an average of 35 to 40 I/O accesses per second. The executable code of POSTGRES and all the databases used in collecting the data

were stored on the local disk to minimize the effect of network traffic.

2.2 Profiling POSTGRES

The first step in speeding up a system is to find out in which routines the system spends most of its execution time. The greatest improvements are likely to result from optimizing those frequently called functions and from reducing the number of calls to those functions. This is exactly the approach taken to improve the performance of POSTGRES.

The profiling work was done by using the DYNIX calls "prof" and "pixie". After the system was compiled into executable code (called "postgres"), the command "pixie postgres" was used to produce another executable file "postgres.pixie". When "postgres.pixie" was executed, the profile data were written to a file called "postgres.Counts". By issuing the command "prof -pixie postgres," the data were interpreted and a report of the execution time for each routine called was generated.

Two databases with different sets of queries were used to obtain the profile data. The first database is very similar to the one specified in the Wisconsin Benchmark [3] except that 4-byte integers were used instead of 2-byte integers. The database consists of one 1-K (named "onek") and two 10-K (named "tenk1," "tenk2") tables of 208-byte records. No B-tree index has been created on any field of the relations. The set of POSTQUEL queries executed on this database is shown in Figure 2.1. The set of queries is data-intensive [4], meaning that the time required to process the data should be much greater than the overhead processing time (e.g., communication, parsing, planning, and validity check times). It is data-

intensive because no index has been defined on the big relations, and for each of the few queries, the system has to touch all the records in a relation. The queries are a subset of those included in the Wisconsin Benchmark [3] but do include all the basic query types: retrieve (both selection and projection), replace, append, delete, and destroy. Aggregate functions were not included because they did not exist in POSTGRES at the time the project was undertaken.

```
retrieve into temp0 (tenk1.all) where (tenk1.unique2 > 1) and (tenk1.unique2 < 102)
destroy temp0
retrieve into temp2 (tenk2.all) where (tenk2.unique2 > 101) and (tenk2.unique2 < 202)
destroy temp2
retrieve into temp11 (tenk1.all) where (tenk1.unique2 > 1) and (tenk1.unique2 < 1002)
destroy temp11
retrieve into temp12 (tenk2.all) where (tenk2.unique2 > 101) and (tenk2.unique2 < 1102)
destroy temp12
retrieve (tenk1.all) where tenk1.unique2 = 1
retrieve (tenk2.all) where tenk2.unique2 = 1001
retrieve into temp71 (onek.all)
destroy temp71
append tenk1 (unique1 = 10001, unique2 = 10001, two = 0, four = 0, ten = 0, twenty = 0, hundred = 0,
  thousand = 0, twothousand = 0, fivethousand = 0, tenthousand = 0, odd = 1, even = 0,
  stringu1 = "MXXXXXXXXXXXXXXXXXXXXXXXXXXGXXXXXXXXXXXXXXXXXXXXXXXC",
  stringu2 = "GXXXXXXXXXXXXXXXXXXXXXXXXXXCXXXXXXXXXXXXXXXXXXXXXXXA",
  string4 = "OXXXXXXXXXXXXXXXXXXXXXXXXXXOXXXXXXXXXXXXXXXXXXXXXXXO")
append tenk2 (unique1 = 10001, unique2 = 10001, two = 0, four = 0, ten = 0, twenty = 0, hundred = 0,
  thousand = 0, twothousand = 0, fivethousand = 0, tenthousand = 0, odd = 1, even = 0,
  stringu1 = "MXXXXXXXXXXXXXXXXXXXXXXXXXXGXXXXXXXXXXXXXXXXXXXXXXXC",
  stringu2 = "GXXXXXXXXXXXXXXXXXXXXXXXXXXCXXXXXXXXXXXXXXXXXXXXXXXA",
  string4 = "OXXXXXXXXXXXXXXXXXXXXXXXXXXOXXXXXXXXXXXXXXXXXXXXXXXO")
delete tenk1 where tenk1.unique2 = 10001
delete tenk2 where tenk2.unique2 = 10001
replace tenk1 (unique2 = 10011) where tenk1.unique2 = 1491
replace tenk2 (unique2 = 10011) where tenk2.unique2 = 1491
```

Figure 2.1 A Subset of Queries in the Wisconsin Benchmark

```
create branch (bid = int4, balance = float8, string = text)
create teller (tid = int4, balance = float8, string = text)
create account (aid = int4, balance = float8, string = text)
create history (bid = int4, tid = int4, aid = int4, amount = float8, string = text)
copy branch () from "/a/guest/plai/tp/branch"
copy teller () from "/a/guest/plai/tp/teller"
copy account () from "/a/guest/plai/tp/account"
define index branch_index on branch using btree (bid int4_ops)
define index teller_index on teller using btree (tid int4_ops)
define index account_index on account using btree (aid int4_ops)
```

Figure 2.2 The Template for Setting up the Database for the TP1 Benchmark

The second database is very similar to the one used in measuring TP1 [1]. It consists of a 1-K relation called "branch," two 10-K relations called "teller" and "account," and a relation named "history" to which a record for each transaction is appended. The size of a tuple in the "branch," "teller," and "account" relations is 100 bytes, and a tuple in the "history" relation has 50 bytes. The POSTQUEL commands defining the relations are shown in Figure 2.2. The set of queries executed includes 300 transactions, each of which is composed of the POSTQUEL commands shown in Figure 2.3, except that variables **$1**, **$2**, and **$3** are replaced by numbers generated randomly. There are B-tree indices on the "bid," "tid," and "aid" fields of the "branch," "teller," and "account" relations respectively. The query set is overhead-intensive [4], meaning that the time spent in the operating system and data management overhead is significant when compared with the time required to process the data. It is overhead-intensive because there are a large number of commands (300 * 7 = 2100), each requiring the fetching of a single record from a relation with useful storage structures (i.e. B-tree indices).

```
begin
retrieve (account.all) where account.aid = $1
replace account (balance = account.balance + 10.0) where account.aid = $1
replace teller (balance = teller.balance + 10.0) where teller.tid = $2
replace branch (balance = branch.balance + 10.0) where branch.bid = $3
append history (bid = $3, tid = $2, aid = $1, amount = 10.0,
  string = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx")
end
```

Figure 2.3 The Definition of a Transaction

By profiling POSTGRES on two completely different sets of queries, the high traffic routines under different access patterns could be identified. From the profile report, time spent in each subsystem in POSTGRES (e.g. parser, optimizer) was also calculated. By looking at the breakdown in execution time, one can tell easily whether the system has spent too much time in a single subsystem. The reports were distributed among the development team members and suggestions for modifications were made corresponding to the results obtained. This process was repeated several times. The results and the interpretations of the profiling work will be presented later in this paper (Section 3.1).

2.3 Benchmarking POSTGRES

To have a clearer idea of how the performance of POSTGRES compares to that of university INGRES, and to provide a reference for future work in improving the performance of POSTGRES, we decided to benchmark POSTGRES. The Wisconsin Benchmark [2] was chosen as a common ground for comparing POSTGRES with university INGRES for the following reasons:

1) Although the Wisconsin Benchmark is not a comprehensive tool for measuring performances of DBMS's, it tests basic functionalities that are provided in most DBMS's.
2) The Wisconsin Benchmark is widely used in benchmarking DBMS's, so the results can be used for quick comparisons with DBMS's other than university INGRES.

NOTE: The original strategy was to benchmark POSTGRES and compare the results with published Wisconsin Benchmark results for university INGRES [2]. Since the published results were obtained by running university INGRES on a VAX 11/750, and the measurements for POSTGRES were done on a DECstation 3100, a conversion factor accounting for hardware and software differences between the two machines would have been needed for meaningful comparison. However there is no single conversion factor that would apply to all kinds of queries. For example, differences between the two machines in CPU speed as well as for I/O processes would have to be reckoned for different types of queries. Due to the difficulty and uncertainty in estimating accurate conversion factors, the idea of using published benchmarking results for university INGRES was abandoned.

To have an accurate comparison between the DBMS's, both university INGRES and POSTGRES were set up on the same SUN workstation. A SUN workstation was used because it is the only machine on which both DBMSs can run. The version of university INGRES used was released in 1987, and included enhancements not found in the system used to obtain previously published benchmarking results. Hence, performance results for this version of INGRES were expected to be slightly better than the published results. The benchmarking results that reflect all the optimizations done in this project will be given later

in this paper.

POSTGRES was also benchmarked using the TP1 Benchmark [1]. Reasons for using the TP1 Benchmark in addition to the Wisconsin Benchmark are as follows:

1) The queries in the Wisconsin Benchmark are data-intensive. Without the TP1 Benchmark, the performance of POSTGRES on overhead-intensive queries may not be obtained.
2) TP1 rating is also widely used as an index of the speed of a DBMS.
3) Lock and transaction managers can be tested by the TP1 Benchmark, and problems not detected with the Wisconsin Benchmark may be discovered when running the TP1 Benchmark.

The database and the queries used are the same as those described in Section 2.2.

The execution time was measured by the UNIX command "time," which gives the user time, system time, elapsed time, CPU usage, number of I/O accesses, and number of page faults for a command.

During the benchmarking work, a number of problems not discovered in the profiling stage were found. The system has been modified and re-benchmarked many times to improve performance. The nature of the changes and the results will be described later in this paper (Section 3.2).

2.4 Testing the Optimizer in POSTGRES

Besides optimizing the frequently called routines and trying to reduce the number of calls to those routines, a DBMS can also be speeded up by eliminating faults in the optimizer. The work is not aimed at producing a faster optimizer since it was found from the profile data that the time spent in the optimizer is not significant compared to the execution time. This part of the project focuses more on the quality of the execution plan produced by the optimizer, e.g. whether the right join method or scan method is planned. It prevents the performance of the system from being degraded for particular kinds of queries. For example, the execution time may be a lot longer if a nested loop join (also called iterative substitution), rather than a merge join or a hash join, is planned for joining two large relations having no indices.

```
retrieve into temp1 (tenk1.all) where (tenk1.unique2 < X002) and (tenk1.unique2 > 1)
destroy temp1
retrieve into temp2 (tenk2.all) where (tenk2.unique2 < X002) and (tenk2.unique2 > 1)
destroy temp2
retrieve into temp3 (tenk1.all) where (tenk1.unique2 < X003) and (tenk1.unique2 > 2)
destroy temp3
retrieve into temp4 (tenk2.all) where (tenk2.unique2 < X003) and (tenk2.unique2 > 2)
destroy temp4
retrieve into temp5 (tenk1.all) where (tenk1.unique2 < X004) and (tenk1.unique2 > 3)
destroy temp5
retrieve into temp6 (tenk2.all) where (tenk2.unique2 < X004) and (tenk2.unique2 > 3)
destroy temp6
```

Figure 2.4 The First Set of Optimizer Tests

It was expected that some of the faults in the optimizer, if any existed, could be exposed during the benchmarking. Three additional sets of tests were designed to verify the decisions made by the optimizer. The first set of tests runs on a database with two 10-K tables of 208-byte records. Each record is identical to those described in Section 2.2. The

queries are listed in Figure 2.4. The two relations were alternated to minimize the effect of buffering. The value of **X** was varied from 1 to 9 so that the selectivity changed from 0.1 to 0.9. Clustered B-tree indices were defined on the field "unique2" in all relations for all queries. This first set of tests was designed to examine the decisions made by the optimizer in choosing between sequential scan and index scan. It checks if the optimizer correctly switches from index scan to sequential scan when the selectivity of the queries increases gradually.

The second set of tests runs on a database with two 1-K tables of 208-byte records (same records as above). The queries are listed in Figure 2.5. This second set of queries was designed to examine the decisions made by the optimizer in choosing between the three join methods supported by POSTGRES: nested loop join, merge join, and hash join. Queries 1 and 2 test the quality of equal-join plans. Queries 3 and 4 check how restrictions affect the plans. Queries 5 and 6 test to see if only nested loop join is planned for non-equal-joins. Clustered B-tree indices were defined on "unique2" in all relations for queries 2, 4, and 6, but not queries 1, 3, and 5. By running the same queries both with and without B-tree indices, one can check if the optimizer can give plans that make good use of available indices.

The third set of tests runs on the same database used for the second set of tests. The queries are listed in Figure 2.6. They were designed to determine if the cost given by the optimizer is sensitive only to the selectivity of a range qualification and not to the absolute value of the range.

query1:
  retrieve into temp1 (u11 = onek1.unique1, u21 = onek1.unique2, two1 = onek1.two,
    four1 = onek1.four, ten1 = onek1.ten, twenty1 = onek1.twenty, h1 = onek1.hundred,
    t1 = onek1.thousand, tt1 = onek1.twothousand, ft1 = onek1.fivethousand, tent1 =
    onek1.tenthousand, odd1 = onek1.odd, even1 = onek1.even, st11 = onek1.stringu1,
    st21 = onek1.stringu2, st41 = onek1.string4, u12 = onek2.unique1, u22 = onek2.unique2,
    two2 = onek2.two, four2 = onek2.four, ten2 = onek2.ten, twenty2 = onek2.twenty,
    h2 = onek2.hundred, t2 = onek2.thousand, tt2 = onek2.twothousand, ft2 = onek2.fivethousand,
    tent2 = onek2.tenthousand, odd2 = onek2.odd, even2 = onek2.even, st12 = onek2.stringu1,
    st22 = onek2.stringu2, st42 = onek2.string4) where (onek1.unique2 = onek2.unique2)
query2:
  same as query1, except that Btree indices were defined on onek1, and onek2
query3:
  retrieve into temp2 (u11 = onek1.unique1, u21 = onek1.unique2, two1 = onek1.two,
    four1 = onek1.four, ten1 = onek1.ten, twenty1 = onek1.twenty, h1 = onek1.hundred,
    t1 = onek1.thousand, tt1 = onek1.twothousand, ft1 = onek1.fivethousand, tent1 =
    onek1.tenthousand, odd1 = onek1.odd, even1 = onek1.even, st11 = onek1.stringu1,
    st21 = onek1.stringu2, st41 = onek1.string4, u12 = onek2.unique1, u22 = onek2.unique2,
    two2 = onek2.two, four2 = onek2.four, ten2 = onek2.ten, twenty2 = onek2.twenty,
    h2 = onek2.hundred, t2 = onek2.thousand, tt2 = onek2.twothousand, ft2 = onek2.fivethousand,
    tent2 = onek2.tenthousand, odd2 = onek2.odd, even2 = onek2.even, st12 = onek2.stringu1,
    st22 = onek2.stringu2, st42 = onek2.string4) where (onek1.unique2 = onek2.unique2)
    and (onek2.unique2 < 100)
query4:
  same as query3, except that Btree indices were defined on onek1, and onek2
query5:
  retrieve into temp3 (u11 = onek1.unique1, u21 = onek1.unique2, two1 = onek1.two,
    four1 = onek1.four, ten1 = onek1.ten, twenty1 = onek1.twenty, h1 = onek1.hundred,
    t1 = onek1.thousand, tt1 = onek1.twothousand, ft1 = onek1.fivethousand, tent1 =
    onek1.tenthousand, odd1 = onek1.odd, even1 = onek1.even, st11 = onek1.stringu1,
    st21 = onek1.stringu2, st41 = onek1.string4, u12 = onek2.unique1, u22 = onek2.unique2,
    two2 = onek2.two, four2 = onek2.four, ten2 = onek2.ten, twenty2 = onek2.twenty,
    h2 = onek2.hundred, t2 = onek2.thousand, tt2 = onek2.twothousand, ft2 = onek2.fivethousand,
    tent2 = onek2.tenthousand, odd2 = onek2.odd, even2 = onek2.even, st12 = onek2.stringu1,
    st22 = onek2.stringu2, st42 = onek2.string4) where (onek1.unique2 < onek2.unique2)
    and (onek1.unique2 < 10) and (onek2.unique2 < 100)
query6:
  same as query5, except that Btree indices were defined on onek1, and onek2

Figure 2.5 The Second Set of Optimizer Tests

```
query 1:
  retrieve (tenk1.all) where tenk1.unique2 < 502 and tenk1.unique2 > 500
query 2:
  retrieve (tenk1.all) where tenk1.unique2 < 502 and tenk1.unique2 < 500
query 3:
  retrieve (tenk1.all) where tenk1.unique2 < 102 and tenk1.unique2 > 100
query 4:
  retrieve (tenk1.all) where tenk1.unique2 < 102 and tenk1.unique2 < 100
```

Figure 2.6 The Third Set of Optimizer Tests

The procedure for conducting the above tests was as follows. First, the optimizer was allowed to choose its own plan for each of the queries. Then internal flags were set to force the system to use other alternatives for each of those same queries. The plans and timing were compared to see if the optimizer was correct. Again, the UNIX command "time" was used to do the measurement. The results will be presented later in Section 3.3.

3. Results

This section presents a description and interpretation of the results. Section 3.1 gives the profiling data at different stages of the speed up process. It describes the changes that account for the speedup factor of three and presents the incremental improvement resulting from each change. Section 3.2 lists POSTGRES's performances on the Wisconsin and TP1 Benchmarks. It also includes the problems discovered through out the benchmarking phase and the corresponding solutions. Section 3.3 shows the results, with their implications, of the optimizer tests described in Section 2.4.

3.1.1 The First Set of Profiling Data

Figure 3.1 shows the first set of profile data obtained by running the queries listed in Figure 2.1. "Fastgetattr" appeared first in the list, occupying 18.14% of the total execution time. It is a routine in the access methods which returns a certain field in a tuple when the tuple and the attribute number of the requested field are given. It accounts for almost one-fifth of the total execution time because it is called by nearly all the modules in the system, and it is called whenever the value of a field in a tuple is accessed.

Total number of cycles = 2618063022

cycles	% cycles	cum %	cycles/call	procedure (file)
474836539	18.14	18.14	262	fastgetattr (tuple.c)
349159913	13.34	31.47	45	NodeIsType (inh.c)
126453252	4.83	36.30	1252	prs2Retrieve (prs2retrieve.c)
107532633	4.11	40.41	34	BufferIsValid (bufmgr.c)
87267648	3.33	43.74	55	prs2ActivateBackwardChainingRules (prs2bkwd.c)
67328668	2.57	46.32	38	amgetattr (tuple.c)
59392116	2.27	48.58	588	attributeValuesCreate (prs2attr.c)
54332183	2.08	50.66	55	malloc (malloc.c)
50773180	1.94	52.60	335	fmgr (fmgr.c)
46864586	1.79	54.39	30	prs2ActivateForwardChainingRules (prs2bkwd.c)

Figure 3.1 The First Set of Profile Data

In POSTGRES, an execution plan produced by the optimizer is in the form of a tree. During execution, the executor traverses the tree, performing actions according to the types and the attributes stored in the nodes of the tree. Each node may have one of over 80 different types, such as "Sort", "MergeJoin", "HashJoin", "Scan", etc. However, nodes cannot have arbitrary types. For example, only "Material" nodes, "Sort" nodes or "Unique" nodes can be

children of a "Temp" node. The rules stating the valid types the children of a certain node can have form a type inheritance tree. The second routine in the list shown in Figure 3.1, "NodeIsType", is a routine which determines if a given node has a type which is a descendent of another given type in the type inheritance tree.

"BufferIsValid," which was ranked fourth, checks to see if a given buffer descriptor points to a valid buffer page managed by the POSTGRES buffer manager.

"Prs2ActivateBackwardChainingRules," "prs2ActiveForwardChainingRules" and "prs2Retrieve" are routines within the rule manager. "Prs2Retrieve" is called for each tuple whenever a retrieve operation is performed on a relation by the executor. It determines whether the current tuple is affected by any rules defined, and if so, creates the resulting tuple. "Prs2ActiveBackwardChainingRules" and "Prs2ActiveForwardChainingRules" are two of those routines called by "prs2Retrieve". They check to see if any backward chaining rules and forward chaining rules defined will affect the current tuple, and if so, return the correct result.

It is obvious that "fastgetattr" is called numerous times through out the execution and that POSTGRES spends too much time there for each call. It is certainly not doing what its name suggests! Some local optimizations were made in "fastgetattr." For example, the value of an expression tested many times within "fastgetattr" is saved in a variable to prevent the expression from being evaluated more than once. Moreover, the executor and the rule manager were modified to save results from "fastgetattr" whenever possible so that it might be called less frequently. The changes reduced the number of CPU cycles per call to

"fastgetattr" from 262 to 167 and the number of calls by 88.81%, bringing it down from the top to fifth in the list.

% time	subsystem
19.51	access methods
19.09	system functions
18.83	lib
15.64	rule manager
11.26	storage manager
10.14	utilities
4.66	executor
0.88	optimizer (planner)

Figure 3.2 The Timing Break Down for the First Set of Profile Data

Figure 3.2 gives the breakdown of the timing grouped by subsystems in POSTGRES. Routines under the lib and utilities categories are shared by all subsystems in POSTGRES. A significant portion of the execution time was spent in low level subsystems in the DBMS (e.g. storage manager, access methods, etc.) because the queries are data-intensive. However, the time taken by the rule manager was unexpectedly long. The whole rule subsystem took up 15.64% of the total execution time even when no rule was involved in the queries, and that did not include the time spent in the library, utility and access method routines called by the rule manager. Such a slow rule manager is certainly undesirable. Most of that 15% of execution time is used in setting up the rule environment and determining whether data being manipulated are governed by any rules. The overhead is so great because "prs2Retrieve" does not check as to whether there is a rule lock on the current relation. It defers the check until individual rule activation routines are called for each related attribute in each tuple. Not

only is greater overhead incurred by doing that; checking is also repeated. The routine was modified so that the check is now made at the very beginning. The rule manager returns immediately if no rule is defined on the current relation.

3.1.2 The Second Set of Profile Data

Total number of cycles = 1424178040

cycles	% cycles	cum %	cycles/call	procedure (file)
361237275	25.36	25.36	47	NodeIsType (inh.c)
83880093	5.89	31.25	36	BufferIsValid (bufmgr.c)
55489865	3.90	35.15	339	fmgr (fmgr.c)
33860943	2.38	37.53	167	fastgetattr (tuple.c)
30075291	2.11	39.64	53	malloc (malloc.c)
22713410	1.59	41.23	197	FindLocalBuffer (bufmgr.c)
20964390	1.47	42.71	15	BM_debug (buf_sync.c)
19796392	1.39	44.10	196	ExecMakeBogusScanAttributes (eutils.c)
19052880	1.34	45.43	33	MemoryContextAlloc (mcxt.c)
17320800	1.22	46.65	30	AllocSetAlloc (aset.c)

Figure 3.3 The Second Set of Profile Data

% time	subsystem
38.12	lib
18.80	storage manager
16.77	utilities
6.71	executor
5.84	access methods
5.07	system functions
1.03	rule manager
0.58	optimizer (planner)

Figure 3.4 The Timing Break Down for the Second Set of Profile Data

After the rule manager and the access method had been changed, the profiling was repeated for the queries shown in Figure 2.1. Results are shown in Figure 3.3 and Figure 3.4. Note from the decrease in total number of cycles that POSTGRES was speeded up by 83.83% due to the modifications described above. "Fastgetattr" dropped from top of the list to fifth, and the number of CPU cycles used per call dropped from 262 to 167. No routines in the rule subsystem remained among the top ten time-consumers. The most heavily called rule routine is in the 43rd position. The great improvement verifies that the approach used to speed up the system is effective.

3.1.3 Profile Data for Transactions

Total number of cycles = 399556840

cycles	% cycles	cum %	cycles/call	procedure (file)
65448041	16.38	16.38	280	_doprnt (doprnt.c)
38194770	9.56	25.94	52	NodeIsType (inh.c)
30160232	7.55	33.49	363	fmgr (fmgr.c)
14040260	3.51	37.00	54	malloc (malloc.c)
10557409	2.64	39.64	33	BufferIsValid (bufmgr.c)
8791922	2.20	41.84	311	yylook (scan.c)
8298378	2.08	43.92	33	MemoryContextAlloc (mcxt.c)
6789582	1.70	45.62	27	AllocSetAlloc (aset.c)
6269200	1.57	47.19	28	fprintf (fprintf.c)
6005856	1.50	48.69	131	fastgetattr (tuple.c)

Figure 3.5 The Profile Data for Transactions

% time	subsystem

25.72	system functions
22.08	lib
18.84	utilities
12.18	storage manager
11.73	access methods
2.66	executor
1.37	parser
1.00	optimizer (planner)
0.05	rule manager

Figure 3.6 The Timing Break Down for Profile Data for Transactions

The improved system was then profiled using the database described in Section 2.2 and the queries listed in Figure 2.3. The results are shown in Figure 3.5 and Figure 3.6. "Doprnt" is a routine that performs the output for "sprintf", which prints strings to a buffer with a specified size. "Fmgr" is the function manager in POSTGRES, and it dispatches function calls through table look-ups. "Malloc" is a library function for dynamic memory allocation. Both "MemoryContextAlloc" and "AllocSetAlloc" are routines within the memory manager in POSTGRES. The former is responsible for dynamic memory allocation in a given context, and the latter allocates memory for a set of specified items. When comparing the results shown in Figure 3.4 and Figure 3.6, it was noticed that the percentage of time spent in the parser, the planner, and system functions increased while the percentage of time spent in the storage manager and the executor decreased. The results confirm that the queries are overhead-intensive. However, the change in execution time distribution is not dramatic. Moreover, the time spent in parsing and choosing an execution plan is still low compared to the data processing time. Routines like "NodeIsType," "fmgr," "BufferIsValid" remain high in the list. In subsequent runs, the profile data obtained by running the first set of queries (shown in Figure 2.1) were chosen to be the common reference in comparing performances

in different stages of the speed up process.

3.1.4 The Third Set of Profile Data

After the improvements in "fastgetattr" and the rule manager were made, "NodeIsType" became the top time-consumer. The next step was to seek improvements there. It was discovered that "NodeIsType" is sometimes called to determine whether a node has a certain inherited type although the answer is known beforehand. To provide a way to bypass those tests, a compilation directive (NO_NODE_CHECKING) was added. With NO_NODE_CHECKING defined, the system was profiled again. The results are shown in Figure 3.7. As indicated by the total number of cycles, POSTGRES has increased in speed by another 26.18%.

Total number of cycles = 1128683572

cycles	% cycles	cum %	cycles/call	procedure (file)
119778377	10.61	10.61	53	NodeIsType (inh.c)
83880093	7.43	18.04	36	BufferIsValid (bufmgr.c)
55491325	4.92	22.96	339	fmgr (fmgr.c)
33861375	3.00	25.96	167	fastgetattr (tuple.c)
30075282	2.66	28.63	53	malloc (malloc.c)
22713410	2.01	30.64	197	FindLocalBuffer (bufmgr.c)
20964390	1.86	32.49	15	BM_debug (buf_sync.c)
19796392	1.75	34.25	196	ExecMakeBogusScanAttributes (eutils.c)
19052880	1.69	35.94	33	MemoryContextAlloc (mcxt.c)
17320800	1.53	37.47	30	AllocSetAlloc (aset.c)

Figure 3.7 The Third Set of Profile Data

3.1.5 The Fourth Set of Profile Data

Examination of the source code of POSTGRES revealed that there are many validity checks in the program. For example, there are numerous places where a pointer is checked to see if it points to valid data; if not, the system gives an error message and then either continues or stops. In POSTGRES, most of these checks are implemented by the following macro functions: Assert(), AssertState(), AssertArg(), LogAssert(), LogAssertState(), and LogAssertArg(). These checks are quite useful in the development phase when the system is incomplete and not fully tested. However, checking cases that could not occur when the system functions normally represents an overhead and degrades the performance unnecessarily. Accordingly, a compilation directive (NO_ASSERT_CHECKING) was added to the code so that those checks could be easily turned on or off by defining or undefining NO_ASSERT_CHECKING. The profile data for the system with NO_ASSERT_CHECKING defined are given in Figure 3.8. Another speedup of 23.42% was attained, as indicated by comparing the new total number of cycles with the old total.

Total number of cycles = 914535407

cycles	% cycles	cum %	cycles/call	procedure (file)
60438523	6.61	6.61	47	NodeIsType (inh.c)
55493874	6.07	12.68	339	fmgr (fmgr.c)
53619029	5.86	18.54	34	BufferIsValid (bufmgr.c)
32231012	3.52	22.06	159	fastgetattr (tuple.c)
30075282	3.29	25.35	53	malloc (malloc.c)
22713410	2.48	27.84	197	FindLocalBuffer (bufmgr.c)
19796392	2.16	30.00	196	ExecMakeBogusScanAttributes (eutils.c)
19232925	2.10	32.10	15	BM_debug (buf_sync.c)

15588720	1.70	33.81	27	AllocSetAlloc (aset.c)
14461436	1.58	35.39	142	ExecMakeFunctionResult (qual.c)

Figure 3.8 The Fourth Set of Profile Data

3.1.6 The Fifth Set of Profile Data

It is interesting that "BM_debug," a routine used to print debugging information for the buffer manager, appeared 8th in the profile shown in Figure 3.8 although most of the calls to it had already been suppressed. It was found that "BM_debug" was called by another debugging routine, "flag_print", which was called by "RelationGetBuffer," itself called numerous times during normal execution. Although "BM_debug" returned without doing anything when it was not in debug mode, the two levels of useless calls caused an overhead of 2.1%. The inefficient code was corrected and profile data showing the improvement are given in Figure 3.9.

Total number of cycles = 876602510

cycles	% cycles	cum %	cycles/call	procedure (file)
60438523	6.89	6.89	47	NodeIsType (inh.c)
55494963	6.33	13.23	339	fmgr (fmgr.c)
53616101	6.12	19.34	34	BufferIsValid (bufmgr.c)
32231801	3.68	23.02	159	fastgetattr (tuple.c)
30075282	3.43	26.45	53	malloc (malloc.c)
22713410	2.59	29.04	197	FindLocalBuffer (bufmgr.c)
19796392	2.26	31.30	196	ExecMakeBogusScanAttributes (eutils.c)
15588720	1.78	33.08	27	AllocSetAlloc (aset.c)
14461436	1.65	34.73	142	ExecMakeFunctionResult (qual.c)
14315412	1.63	36.36	141	heapgettup (access.c)

Figure 3.9 The Fifth Set of Profile Data

3.1.7 The Sixth Set of Profile Data

After getting fruitful results from eliminating some of the validity checks, it was thought that similar optimization could be made to "BufferIsValid", which checks whether a given buffer descriptor points to a valid buffer page. The routine was modified so that it simply returned true whenever called. The system crashed after the modification. The source code was then carefully examined, and it was found that some of the calls to "BufferIsValid" are meaningful. It is sometimes used to see if a certain buffer page exists. If not, the buffer page is fetched. So blindly returning true from "BufferIsValid" will undermine the normal execution. As a result, every call to "BufferIsValid" was examined. A compilation directive (NO_BUFFERISVALID) was added to each call to "BufferIsValid" that serves only as an error check. Those calls that affect decision making were left intact. In this way, checks on buffer pointers can be turned off when they are not necessary. The profile data shown in Figure 3.10 indicate a further speedup of 2.75% can be obtained by setting NO_BUFFERISVALID. Altogether, POSTGRES's speed has been raised by 206.86% (three times as fast as before).

Total number of cycles = 853181733

cycles	% cycles	cum %	cycles/call	procedure (file)
60438523	7.08	7.08	47	NodeIsType (inh.c)
55492049	6.50	13.59	339	fmgr (fmgr.c)
36907901	4.33	17.91	31	BufferIsValid (bufmgr.c)
32230512	3.78	21.69	159	fastgetattr (tuple.c)
30075291	3.53	25.22	53	malloc (malloc.c)
22408604	2.63	27.84	195	FindLocalBuffer (bufmgr.c)

19796392	2.32	30.16	196	ExecMakeBogusScanAttributes (eutils.c)
15588720	1.83	31.99	27	AllocSetAlloc (aset.c)
14461436	1.70	33.69	142	ExecMakeFunctionResult (qual.c)
13679448	1.60	35.29	135	heapgettup (access.c)

Figure 3.10 The Sixth Set of Profile Data

3.2.1 Performance on the Wisconsin Benchmark

Figure 3.11 shows the results for running the Wisconsin Benchmark on both POSTGRES and university INGRES. The benchmarking was done near the end of the project when all optimizations described above had been made.

query	POSTGRES (sec.)	INGRES (sec.)	ratio (POSTGRES/INGRES)
1: select 1% into temp, no index	47.7	11.8	4.04
2: select 10% into temp, no index	59.0	21.1	2.80
3: select 1% into temp, clust. index	5.1	2.3	2.22
4: select 10% into temp, clust. index	39.4	13.7	2.88
5: select 1% into temp, non-clust. index	7.5	4.5	1.67
6: select 10% into temp, non-clust. index	61.4	26.3	2.33
7: select 1 to screen, clust. index	1.2	1.0	1.20
8: select(1%) to screen, clust. index	5.2	1.0	5.20
9: joinAselB, no index	228.0	816.0	0.28
10: joinABprime, no index	234.0	790.0	0.30
11: joinCselAselB, no index	244.0	175.0	1.39
12: joinAselB, clust. index	266.0	51.0	5.22
13: joinABprime, clust. index	279.0	48.0	5.81
14: joinCselAselB, clust. index	298.0	96.0	3.10
15: joinAselB, non-clust. index	261.0	77.0	3.39
16: joinABprime, non-clust. index	201.0	55.0	3.65
17: joinCselAselB, non-clust. index	277.0	116.0	2.39
18: project 1% into temp	169.0	64.0	2.64
19: project 100% into temp	55.0	14.0	3.93
20: min scalar aggr., no index	----	9.0	----
21: min aggr. func., no index	----	47.0	----
22: sum aggr. func., no index	----	43.0	----
23: min scalar aggr., with index	----	9.0	----

24: min aggr. func., with index	----	60.0	----
25: sum aggr. func., with index	----	44.0	----
26: insert, no index	8.0	0.8	10.00
27: delete, no index	45.8	8.3	5.52
28: update key, no index	52.3	7.8	6.71
29: insert, with index	5.6	1.2	4.67
30: delete, with index	2.3	0.4	5.75
31: update key, with index	3.8	0.5	7.60
32: update non-key, with index	58.0	7.9	7.34

Figure 3.11 Comparisons between POSTGRES and INGRES on the Wisconsin Benchmark

In general, POSTGRES is about three times slower than university INGRES (after all modifications mentioned previously have been made). It does retrievals faster than insertions and deletions. When no index is defined on the relations, it performs joins around four times faster than university INGRES does. The reason is that university INGRES can do joins only by nested loop join, while POSTGRES is equipped with hash join and merge join as well. When no index is defined on the relations, nested loop join has to scan the relations many more times than either hash join or merge join. This is the only circumstance in which POSTGRES bests university INGRES at the present time. It should be noted that comparisons were not made for queries 20 to 25 in the Wisconsin Benchmark because no aggregate function is currently supported by POSTGRES.

There are several reasons why POSTGRES is slower than university INGRES. First, university INGRES is a mature research project, while POSTGRES is an active one. So various parts of the POSTGRES design and implementation are still under development, and further optimization is possible. Second, POSTGRES is a much larger DBMS, including many extended features not found in university INGRES such as rules, user defined types, history queries, transitive closure queries, and attributes of type relation. Although a query may not

involve them directly, overhead is required to support these additional features.

3.2.2 Performance on the TP1 Benchmark

The TP1 benchmarking work was started after modifications to "fastgetattr" and the rule manager were made. In the first stage, POSTGRES could do 0.699 transactions per second when a single backend was run. The CPU usage was 42%, and the numbers of input and output requests to the disk were 11.86 and 13.79 per transaction respectively. Since the CPU usage was rather low, the measurement was repeated using two backends at the same time. With the help of parallelism, POSTGRES could do 0.893 TP1. The CPU usage was found to be 58%, and the numbers of input and output requests were 9.98 and 13.35 per transaction respectively. Note that in the above tests, POSTGRES had a buffer pool of sixteen 8K pages (default value). Several strange phenomena were observed. First, theoretically, if the CPU cycles and the I/O requests of the two parallel backends overlap perfectly, POSTGRES should be able to perform 1.21 TP1, and the CPU usage should be 72.41%. However, the actual results were not close to the theoretical ones.

Second, the number of I/O requests per transaction was too high. In the worst case, when tuples accessed by consecutive transactions fall on different pages, there should not be more than six reads and six writes per transaction in steady state. The explanation is as follows. It is assumed that in steady state, all system catalogs are in the buffer. Since the database used is large (218 8K pages per relation), each B-tree has two levels. It is assumed that the root page is in the buffer pool all the time, but the leaf pages are not due to the

random accesses. Tuples accessed by consecutive transactions are assumed to be on different pages since the database is very large and the accesses are random. For each transaction, a tuple from each of the relations "account," "teller," and "branch" has to be read. For each tuple, two reads are generated, one for the B-tree leaf page and one for the page containing the tuple. Therefore a total of six reads is needed. POSTGRES does no over-write storage scheme [6], so whenever there is a replacement, the old tuple is marked invalid instead of being updated in-place, and a new tuple is inserted into the relation. Assuming the new tuples are always appended at the end of a relation, that page will remain in the buffer over transactions and a write is not generated for each appending. However, invalidated tuples for consecutive transactions are assumed to fall on random pages. Similar to the above case, then, six writes are needed to invalidate the B-tree leaf page and the old tuple page for each of the three relations.

To examine the first problem, we wrote a small C program to test the environment. The program alternated among read, computation, and write for 5000 iterations. Parameters were adjusted so that the program did I/O around 50% of the time. When the program was run in single user mode, it took 7:44 minutes to complete and the CPU usage was 57%. When two copies of it were run in parallel, it took 10:46 minutes to complete both, and the CPU usage was 80% as compared to 100% in case of perfect overlapping between I/O and computations. The results confirmed that the problem is not unique to POSTGRES, and perfect overlapping between I/O and CPU time does not exist in the system. There are several reasons for this imperfection. The primary reason is that there is time when both processes are reading, so both have to be blocked and the CPU is wasted until one of the reads has

been completed. Moreover, heavier paging activities are generated with two parallel processes. That lengthens the elapsed time, and thus lowers the CPU usage.

To examine the second problem more closely, we traced the read-write activities and buffer page replacements. The buffer manager generated over eighty read and ten write requests to the disk in the first transaction. The reason for the extremely high number of reads is that, for every execution of a query, several system catalogs like "pg_attribute," "pg_operator," and "pg_relation" were scanned sequentially. The table "pg_attribute" alone consumed nine buffer pages, and the total size of all system relations was found to be twenty-five. With only sixteen buffers, the system catalogs were paged out in between queries within a transaction; hence, unnecessary I/O requests occurred. The buffer pool was enlarged, and the system was traced again. Forty-eight buffer pages were used since it was determined that the working set for a transaction should be less than that number.

With a larger buffer pool, the situation improved slightly. >From the tracing, it was found that data pages were touched unnecessarily. The reason was found to be related to a design within the replacement routine. As described above, a replacement is actually implemented as an invalidation and an insertion in POSTGRES. When trying to insert the new tuple, the routine randomly selects a page in the relation and determines whether space is available for the insertion. If insufficient space is found, the process is repeated. After the random checks fail three times, the new tuple is appended to the end of the relation. It should be noted that the pages selected may not be in the buffer pool, and if so, extra disk reads are needed. The main objective of the design is to make use of the space in existing pages of a relation before allocating new ones. However, the trace data showed that those

random checks failed most of the time. Moreover, the scheme does not completely remove the need for garbage collection. It still relies on the vacuum daemon to reclaim unused memory. It was decided that trading real time performance for little improvement in memory usage is not worthwhile. As a result, the replacement routine was changed to bypass the random checks and by default append the new tuple to the end immediately. (One still can get the old behavior by setting a flag during compilation.) With this change, POSTGRES was able to do 0.99 TP1, and the average number of reads per transaction was reduced to 4.77.

The above modification helps to reduce only the number of reads, not writes, per transaction. There are two reasons for the high number of writes per transaction. First, since POSTGRES does no over-write storage management, the logging scheme for crash recovery can be simplified [6]. Instead of logging the the operations of a transaction (logical logging) or saving the before and after images (physical logging), a log entry can record only the status of a transaction, i.e., whether it is committed, aborted, or running. Since the old tuples have never been over-written, no undo is necessary. The right image of a tuple can be retrieved by consulting the log to find out which image has the latest committed transaction identifier. The only requirement for this scheme is that all changes within a transaction have to be written to disk before it is committed; otherwise, those changes are subject to being lost if a crash occurs. This is one reason that the number of writes per transaction is larger than six.

In addition to the flushing of dirty buffers at the end of a transaction, a bug in the append routine also contributes to the high number of writes per transaction. When a new tuple is appended to the end of a relation during a replacement, irrespective of whether the

random checks have been executed or bypassed, the new tuple is put in a new page and the page is appended to the last page of the relation. Memory is thus wasted, and consecutive appends could not be written to disk by flushing a single page since the changes are on different pages.

To fix these problems, several modifications were made. First, a command line argument was added to POSTGRES so that if a user has stable main memory, dirty buffers are written to disk only when they are paged out, but not at the end of each transaction. Second, the append routine was changed to determine if there is space in the last page for the append before allocating new pages. After the modifications, POSTGRES is able to do 1.25 TP1 while stable main memory is assumed, and the numbers of reads and writes to disk per transaction are reduced to 4.43 and 5.1 respectively.

In the following three sections, the results for all the optimizer tests and their implications will be presented.

3.3.1 Results for the First Set of Optimizer Tests

X	plan	planned/forced	time	cal. cost
1	index scan	planned	2:21	9.19e-05
	seq. scan	forced	5:32	1021
2	index scan	planned	3:22	9.19e-05
	seq. scan	forced	5:45	1021
3	index scan	planned	4:26	9.19e-05
	seq. scan	forced	6:13	1021
4	index scan	planned	5:24	9.19e-05

	seq. scan	forced	6:48	1021
5	index scan	planned	6:11	9.19e-05
	seq. scan	forced	7:16	1021
6	index scan	planned	7:13	9.19e-05
	seq. scan	forced	7:49	1021
7	index scan	planned	8:03	9.19e-05
	seq. scan	forced	8:05	1021
8	index scan	planned	9:03	9.19e-05
	seq. scan	forced	8:06	1021
9	index scan	planned	10:11	9.19e-05
	seq. scan	forced	8:55	1021

Figure 3.12 Results for the First Set of Optimizer Tests

The results for running the queries shown in Figure 2.4 are given in Figure 3.12. If sequential scan is used, the whole relation has to be scanned once and only once to get the correct answers no matter what selectivity a query has. When the selectivity of a query is small, only a small portion of the relation has to be scanned if B-tree indices exist, and so a big performance gain can be obtained by using an index scan. When the selectivity of a query is close to 1, most of the pages of a relation have to be scanned anyway. Using an index scan requires the scanning of the B-tree pages also, and thus may be slower than using a sequential scan in this case. The data given in Figure 3.12 show that the optimal switch-over point occurs when the selectivity is about 0.7. However, POSTGRES always chose to use index scan. Moreover, the cost for doing an index scan should vary according to the selectivity of a query, but the cost computed by the optimizer in POSTGRES did not change throughout the tests. Also, the optimizer's cost estimate was unreasonably small.

qualification	equation
key < constant	(constant - min. key value) / (max. key value - min. key value)
key > constant	(max. key value - constant) / (max. key value - min. key value)

Figure 3.13 Equations for Calculating Selectivities

These problems happened for the following reasons. First, the wrong function for calculating the selectivity of a qualification was registered for the "greater than" operator. Second, the way of calculating the selectivity for a "less than" or "greater than" qualification was also wrong. The optimizer set the selectivity to 3 / (number of distinct values appear in that field in the whole relation). The number of distinct values for "unique2" in either "tenk1" or "tenk2" was 10,000, so the selectivity calculated was very close to zero. As a result, the optimizer thought that just very few tuples had to be examined to get the correct answer by doing an index scan. That explained why the calculated cost was unreasonably small. Since the number of distinct key values stayed at 10,000 throughout the tests, the calculated cost did not change over different selectivities, as noted above.

X	plan	planned/forced	time	cal. cost
1	index scan	planned	1:13	235.434
2	index scan	planned	2:17	408.534
3	index scan	planned	3:28	581.634
4	index scan	planned	4:36	754.734
5	index scan	planned	5:44	927.834
6	seq. scan	planned	6:22	1021
7	seq. scan	planned	6:58	1021
8	seq. scan	planned	7:35	1021
9	seq. scan	planned	8:13	1021

Figure 3.14 Revised Results for the First Set of Optimizer Tests

The first bug was fixed by entering the right selectivity function into the appropriate system catalog ("pg_operator"). To solve the second problem, the optimizer was changed to calculate the selectivity according to the equations shown in Figure 3.13 [5]. Note that the high and low key values have to be inserted into the system catalog "pg_statistic" before the optimizer can calculate the right selectivity. Furthermore, the equations assumed an even distribution for the values in a field. After the changes were made, the tests were run again. The results are shown in Figure 3.14. The system was found to switch over to sequential scan from index scan when the selectivity was raised over 0.6. Although the switch-over point can be moved closer to the optimal point mentioned above (0.7) by tuning the I/O-to-CPU factor in the cost function, the current results are considered acceptable. Further tuning may not be meaningful because the switch-over point is already close to the optimal point. Moreover, there is uncertainty in estimating the optimal point due to experimental variations (varying test parameters as well as error). In addition, it was (necessarily) determined in an ad hoc fashion.

3.3.2 Results for the Second Set of Optimizer Tests

query	plan	planned/forced	time
1	hash join	planned	0:23
	merge join	forced	1:06
	nested loop	forced	too long
2	nested loop	planned	0:32

	hash join	forced	0:32
	merge join	forced	1:06
3	nested loop	planned	> 22:21
	hash join	forced	0:18
	merge join	forced	0:34
4	nested loop	planned	0:13
	hash join	forced	0:23
	merge join	forced	0:46
5	nested loop	planned	2:35
	hash join	not appropriate	
	merge join	not appropriate	
6	nested loop	planned	0:25
	hash join	not appropriate	
	merge join	not appropriate	

Figure 3.15 Results for the Second Set of Optimizer Tests

The results for running the optimizer tests given in Figure 2.5 are shown in Figure 3.15. The results verify that the optimizer can pick the right plan for equal-joins. Furthermore, the optimizer does not plan any merge join or hash join for non-equal-joins at all. However, when there was a "less than" restriction in an equal-join and no B-tree index was defined, the optimizer made the wrong choice. It chose nested loop join (i.e. iterative substitution) instead of hash join or merge join. This is partly because of the bugs in the selectivity function (mentioned above). Moreover, the the optimizer did not set the expected size of the results. After correcting the errors, the optimizer was able to pick the right choice. For non-equal-joins, the optimizer made the correct decision, i.e. choosing nested loop join.

3.3.3 Results for the Third Set of Optimizer Tests

query	cal. cost
1: retrieve (tenk1.all) where tenk1.unique2 < 502 and tenk1.unique2 > 500	141.69800
2: retrieve (tenk1.all) where tenk1.unique2 < 502 and tenk1.unique2 < 500	12.19500
3: retrieve (tenk1.all) where tenk1.unique2 < 102 and tenk1.unique2 > 100	78.87540
4: retrieve (tenk1.all) where tenk1.unique2 < 102 and tenk1.unique2 < 100	3.17048

Figure 3.16 Results for the Third Set of Optimizer Tests

The results for running the optimizer tests given in Figure 2.6 are shown in Figure 3.16. They are totally unexpected. First, the cost for query 1 is much greater than the cost for query 2, although the opposite should happen because more tuples are retrieved in query 2 (500) than in query 1 (1). The same error can be observed in query 3 and query 4. Second, the cost for query 1 and query 3 should be more or less the same since they retrieve the same number of tuples (1). However, the resulting costs differ by a factor of 2. The reason for these problems is that the optimizer does not recognize that the two qualifications joined by the "and" are correlated, and it merely multiplies the selectivities of the two qualifications to get the overall selectivity. The selectivity of (unique2 < 502) is 0.05, and the selectivity of (unique2 > 500) is 0.95, and so the overall selectivity is set to 0.0475 (should actually be 0.0001). By the same token, the overall selectivity of query 2 is set to 0.0025 (should actually be 0.05). This explains the poor cost estimates in the tests. Fixing the optimizer so that it can recognize correlated range qualifications requires significant modification. As a result, the development team was notified of the problem, and changes were left for the team members to make.

4. Suggestions

Throughout the project, different problems have been discovered and most were solved. However, it is possible to improve the performance of POSTGRES further. Some suggestions are made here. First, only sequential scan is currently available to access the system catalogs. That may be appropriate when the system catalogs are small. If the database gets very big, some of the system catalogs like "pg_attribute" may become too large for a sequential scan to be efficient. Currently, one can define a B-tree index on any field of any system catalog. However, it will not be used because the normal optimizer is not used to plan the access to system catalogs. Instead, plans are hard-wired to reduce the planning overhead. As shown in the TP1 benchmarking, that may degrade the performance and make the set of working buffers unnecessarily large. Therefore, the code should be modified to make good use of indices defined on system catalogs. An even better but more involved solution is to have specific hash access method for system catalogs. For example, access to "pg_attribute" is normally achieved by retrieving a certain attribute tuple given a relation identifier and the attribute number. If the entries are hashed on the two fields, the fetch can be done in one access and there is no need to bring in all the pages for a complete scan.

Within the source code of POSTGRES, there are numerous error checks. They verify that that the system is in the expected state, or that the data structure has not been damaged. Note that they are intended to detect bugs within the system, not to check for mistakes made by users at run time. Results presented in Section 3.1 showed that great performance gains can be obtained by commenting out those checks. However, in most subsystems of POST-GRES, the routines that do internal error checks are not used just for bug detecting; they are

used also for making decisions and detecting user mistakes in run time. Note that these tests or checks are essential for the normal functioning of the system. For example, "BufferIsValid" is used as an validity check throughout the system, and is also used in the buffer manager to test if a given tuple is already in the buffer pool. Since the routines are used for more than one purpose, there is no easy way to separate calls for different purposes. It is suggested that error-checking routines used only in the debugging phase be coded separately from those needed in normal operation so that the debugging checks can be suppressed easily when performance becomes important. It would be even easier to suppress them if they were macros.

The functions that calculate selectivities are now smart enough to make use of the maximum and minimum values of a certain key. However, those values are not provided by the system. They have to be inserted into the system catalog "pg_statistic" by the user. There are several drawbacks. First, the users have to know their data sets. Furthermore, the identifiers of the relation, the attribute, and the operator involved have to be entered together with the maximum and minimum values. A user has to make several queries to various system catalogs to find out those values before actually inserting the tuple to "pg_statistic". This is an unreasonable burden. It is good, however, if those high and low values of an attribute can be inserted into the system catalog automatically when an index is created for it, and when the vacuum daemon wakes up.

Currently, the optimizer does not know whether an index is clustered or not. A clustered index can only be created by loading the tuples of a relation in the sort order of the indexed field. By default, the optimizer assumes that the tuples are non-clustered. When

doing tests with clustered indices, the cost function has to be changed manually to assume clustering. The customization is something that may be unknown to most users, and is certainly not flexible. As seen in Figure 3.11, the time to perform a retrieve on a relation with clustered indices is much shorter than the time for doing the same operation on a relation with non-clustered indices. Performance may suffer greatly if the optimizer calculates the cost function using only the default assumption about clustering (either clustered or non-clustered) without knowing the real circumstance. It would certainly be useful if the system were to have the capability to cluster any relation over any attribute, and if the optimizer were to know whether a relation is clustered over a certain field. Clustering should not be omitted from a DBMS if performance is important. Before clustering is supported by POSTGRES (and especially if it will never be supported), there should be a command to allow a user to declare that a certain relation is loaded in the sort order of a certain field. With such a command, the optimizer could rely on this information in calculating costs, and there would be no need for a user to change the cost function in POSTGRES to customize with its data.

5. Summary

The main goal of this project was to improve the performance of POSTGRES. The project was divided into three subtasks. First, we profiled the DBMS, and based on the observations, we revised key modules within the DBMS. This phase increased the system's speed by a factor of three.

Second, POSTGRES was benchmarked with the Wisconsin Benchmark and the TP1 Benchmark. Various problems were discovered in doing the benchmarking. They were 1) the buffer pool was too small, 2) the replacement routine did useless random inspections before appending a tuple to the end of a relation, 3) the buffer manager flushed all dirty buffers before committing, and 4) the append routine put the appended tuple in a new page and appended the page to the end of a relation instead of putting the appended tuple into the last page (if space was indeed available) of the relation. By fixing these problems, the system's TP1 rating was doubled. After all modifications mentioned previously have been made, we found that POSTGRES is about three times slower than university INGRES.

Third, three sets of tests were designed to verify the decisions made by the optimizer. The first set determined whether the optimizer chose correctly between sequential scan and index scan when the selectivity of a query changed. By running the tests, bugs in the functions that calculate selectivities were discovered. After those bugs were fixed, the optimizer switched from index scan to sequential scan at the correct time when the selectivity of a query increased gradually. The second set tested whether the optimizer made the correct choice among the various join methods, namely nested loop join, merge join, and hash join. It also examined how restrictions in a query and indices affected the decisions. It was discovered that the optimizer gave the wrong plan when there was a range qualification and no index had been defined. The third set determined whether the cost calculated by the optimizer was sensitive only to the selectivity of a range qualification and not to the absolute value of the range. It was discovered that the optimizer did not handle correlated qualifications correctly.

Through out the project, various suggestions have been made to the development team concerning the performance of POSTGRES. Potentially useful suggestions were 1) to have more efficient methods for accessing system catalogs, 2) to separate error checks that can be suppressed after debugging from those that are essential for the normal execution of the system, 3) to insert maximum and minimum values of an attribute when an index is created on that attribute and also when the vacuum daemon wakes up, and 4) to have the capability to cluster any relation, or at least let the user declare that a certain relation has been clustered.

References

[1] Anon Et Al, "A Measure of Transaction Processing Power," Datamation, 1985.

[2] Dina Bitton, David J. DeWitt, Carolyn Turbyfill, "Benchmarking Database Systems, a Systematic Approach," Proc. 9th International Conference on Very Large Data Bases, November 1983.

[3] Dina Bitton, Carolyn Turbyfill, "A Retrospective on the Wisconsin Benchmark".

[4] Paula Hawthorn, Michael Stonebraker, "The use of Technological Advances to Enhance Data Management System Performance," Technical Report, University of California, Berkeley.

[5] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, "Access Path Selection in a Relational Database Management System," Proceedings of the 1979 SIGMOD Conference, 1979.

[6] Michael Stonebraker, "The Design of the POSTGRES Storage System," Readings 13th International Conference on Very Large Data Bases, Brighton, England, 1987.

[7] Michael Stonebraker, Lawrence A. Rowe, "The Design of Postgres," Proceedings 1986 ACM-SIGMOD Conference on Management of Data, Washington, D. C., May, 1986.

[8] M. Stonebraker, E. Wong, P. Kreps, "The Design and Implementation of INGRES," ACM-TODS, September 1976.