

Decision-Support Workload Characteristics on a Clustered Database Server from the OS Perspective

Yanyong Zhang[§], Jianyong Zhang[†], Anand Sivasubramaniam[†], Chun Liu[†], Hubertus Franke[‡]

[§] Department of Electrical & Computer Engg.
Rutgers, The State University of New Jersey
Piscataway NJ 08854
{yyzhang}@ece.rutgers.edu

[†] Department of Computer Science & Engg.
The Pennsylvania State University
University Park PA 16802
{jzhang, anand, chliu}@cse.psu.edu

[‡] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{frankeh}@us.ibm.com

Abstract

A range of database services are being offered on clusters of workstations today to meet the demanding needs of applications with voluminous datasets, high computational and I/O requirements and a large number of users. The underlying database engine runs on cost-effective off-the-shelf hardware and software components that may not really be tailored/tuned for these applications. At the same time, many of these databases have legacy codes that may not be easy to modulate based on the evolving capabilities and limitations of clusters. An in-depth understanding of the interaction between these database engines and the underlying operating system (OS) can identify a set of characteristics that would be extremely valuable for future research on systems support for these environments. To our knowledge, there is no prior work that has embarked on such a characterization for a clustered database server.

Using IBM DB2 Universal Database (UDB) Extended Enterprise Edition (EEE) V7.2 Trial version and TPC-H like¹ decision support queries, this paper studies numerous issues by evaluating performance on an off-the-shelf Pentium/Linux cluster connected by Myrinet. These include detailed performance profiles of all kernel activities, as well as qualitative and quantitative insights on the interaction between the database engine and the operating system.

1 Introductions

Clusters of workstations built with commodity processing engines, networks and operating systems are becoming the platform of choice for numerous high performance computing environments. Commodity hardware and software com-

ponents make the price, upgradeability and accessibility of clusters very attractive, facilitating their widespread deployment in several application domains. It is no longer just the traditional scientific or engineering community (for computational physics, chemistry, computer-aided design, etc.) that are using such systems. An increasing number of commercial applications (web servers, database engines, search engines, e-commerce, image servers, media bases, etc.) are being hosted on clusters, that are usually back-ends for a web-based interface. Vendors have come out with medium to high end database engines that can exploit a cluster's capabilities [4, 10, 5, 7, 8].

Most commodity off-the-shelf software (including the operating system) are not specifically tuned for cluster environments, and it is not clear if gluing together individual operating systems, that do not know the presence of each other, is the best approach to handle such loads [2]. Further, off-the-shelf operating systems are meant to be for general purpose usage, with most of them really tuned for desktop applications or uniprocessor/SMP class server applications. Their suitability for cluster applications is not well understood.

At the same time, these applications are in turn not extensively tuned for these operating systems. An important reason is the fact that the database engines have legacy codes that have evolved over several revisions/optimizations over the years, and it is difficult to fundamentally change their design overnight in light of these new systems (regardless of how modular they may be), which are still evolving.

Our goal in this paper is not to develop application-specific operating systems, nor is it to find out what OS mechanisms/capabilities are needed for extensibility/customization as other researchers have done [10]. Rather, coming from the applications viewpoint, we would like to make a list of recommendations based on the execution characteristics that can benefit future developments. We have also taken the liberty of suggesting possible mechanisms and their implementation (specifically in Linux) for optimizing the execution based on these gleaned characteristics.

A detailed characterization of the execution of applications on a cluster from the OS perspective can contribute to the knowledge-base of information that can be used for guiding future developments in systems software and applica-

¹These results have not been audited by the Transaction Processing Performance Council and should be denoted as "TPC-H like" workload.

tions for these environments. It would also be invaluable for fine tuning the execution for better performance and scalability, since each of these applications/environments has high commercial impact. With this motivation, this paper embarks on characterizing the complete execution of the IBM DB2 Universal Database (UDB) Extended Enterprise Edition Version 7.2 (which can be downloaded from [1] for 90 day free trial)² on a Linux cluster from the OS perspective. A comprehensive study of several database workloads is overly ambitious and is on our future research agenda. In this paper we focus specifically on TPC-H queries [9], a decision-support database workload.

It is well understood that I/O is the biggest challenge faced by database engines on uniprocessors/SMPs and there is a large body of prior work proposing hardware and software enhancements to address this problem. It is not clear if I/O becomes any less important when we move the engine to a cluster environment, since there is another factor to consider, which is the network communication. System scalability with cluster size is dependent on how parallel is the computation division across the cluster nodes, how balanced are the I/O activities on different nodes, and how does the communication traffic change with data set and cluster sizes. All this requires a careful profiling and analysis of the execution of the queries on the database engine. To our knowledge, there has been no prior investigation of completely characterizing the execution of TPC-H on a clustered database engine, and studying these characteristics for optimization at the application-OS boundary.

The next section reviews literature related to this work, and Section 3 gives details on the experimental setup. Section 4 gives the overall system execution profile. Based on the system profile, the I/O and network characteristics and optimizations are discussed in sections 5 and 6. Finally, Section 7 summarizes the results and contributions of this study.

2 Related Work

Operating systems support for databases has long been considered an important issue for performance [18, 13, 11]. In fact, using the research prototype INGRES system, Stonebraker [18] argues that the buffer manager, file system, scheduling and ipc of Unix are not necessarily those that are suitable for a database engine. Several other studies over the years [20, 21] have addressed specific database engine and OS interaction issues for buffer and virtual memory management, file system support, and network support. Most of these studies have been on systems where the database engine runs on a single node. While there has been some investigation [10] for cases when the database engine runs on multiple nodes, these have been limited to specific issues and did not examine the complete picture. There have been recommendations from developers in the database industry

²Our goal in this paper is NOT to benchmark the database engine on a cluster, or commend/criticize its implementation. We understand that there are several issues in configuring and tuning the database engine for boosting performance, which we may have not attempted. In order to prevent any misconceptions about its absolute performance, which may or may not match the numbers available from the Transaction Processing Performance Council, we do not give absolute execution times or mention the clock speed of the processors used in this study.

[3] suggesting features to incorporate in Linux for database support.

Databases on clusters have drawn little attention from the systems angle (note that there is a large body of work on parallel databases which are not closely related to this work), except for certain specific issues. It has been pointed out that one could harness the physical memory pool across the cluster to create a larger buffer space for reducing I/O [10].

On the characterization front, there has been prior work [14, 15] in I/O characterization for the TPC-C, TPC-D and some production workloads using simulation on traces obtained from a Windows NT SMP multiprocessor. There is also a recent study [16] examining the I/O characteristics of TPC-H on an Intel Xeon server. Message exchange characterizations between processing nodes has not been investigated previously.

This is the first study to look at the complete execution picture of a commercial database engine on a cluster environment with TPC-H.

3 Experimental Setup

The clustered version of DB2 that we use is based on a shared-nothing architecture, wherein each cluster node has its own processor, memory and disk, and the nodes are connected by a communication medium that supports sockets.

Our experiments use an 8 dual node Linux/Pentium cluster, that has 256 MB RAM and 18 GB disk on each node. The nodes (both the server nodes and the client node) are connected by both switched Myrinet and Ethernet. We use Linux 2.4.8, which was the latest release at the time of conducting the experiments. This kernel has been instrumented in detail to glean different statistics, and also modified to provide insight on the database engine execution since we are treating it as a black box. We have also considered the overheads of instrumentation by comparing the results with those provided by the proc file system to ensure validity.

TPC-H [9] is a standard benchmark for decision-support workloads provided by Transaction Processing Performance Council (TPC). It contains a sequence of 22 queries (Q1 to Q22), which examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. We downloaded the utility from TPC site, and generated the dataset of 30 GBytes and the queries. We fired the queries one after another to the database engine³. There are several measures that are used to determine their performance as specified in [9]. In this paper, our consideration is the response time for each query, i.e. the time interval between submitting the query and getting back the results.

4 Operating System Profile

We first present a set of results that depict the overall system behavior at a glance. The following results have been obtained by both sampling the statistics exposed by the Linux proc file system (`stat`, `net/dev`, `process/stat`) as well as by instrumenting the kernel. In fact, these two approaches complement each other, with each helping to vali-

³In this paper, we only present the results of the first 20 queries Q1 to Q20.

date the results obtained from the other approach. The kernel instrumentation was done by inserting code in the Linux system call jump mechanism, as well as in the scheduler and points where there is pre-emption (such as blocking) or resumption. For instance, in a read system call, there is time spent executing the system call operations on the CPU following which the process is blocked. At this instant, time accounting for this process has to be stopped and needs to be resumed when the CPU schedules this again. The instrumentation should again account for time just before the process leaves the system call. A high resolution timer available on the Pentium was used for measuring time, and we observed that the overheads and intrusiveness of the instrumentation was negligible (overall results matched those in the proc file system). The proc file system information is used to present the percentage utilization of the system in different modes, the rates/frequency of I/O, page fault and network activities. The profile of different system calls is presented from the kernel instrumentation.

The first set of results is shown in Table 1, which gives system statistics for each query in terms of: the percentage of time that the query spent executing on the CPUs in user mode (relative to its overall execution time), the percentage of time that the query spent executing on the CPUs in system mode (relative to its overall execution time), the average number of page faults incurred in its execution per jiffy (10 milliseconds in Linux), average number of file blocks read per jiffy, average number of file blocks written per jiffy, average number of packets sent over the network per jiffy, the average number of packets received from the network per jiffy, and the percentage utilization of the CPU(s) by the database engine during I/O operations (captures the overlap of work with I/O operations). The file block size is 4096 bytes, and the Maximum Transfer Unit (MTU) for network packets is 3752 bytes. In addition to these, the table also shows the top four system calls (in terms of time) exercised by each query during its execution, and the percentage of system time that is spent in each of these calls. We have not included system calls that take less than 1% of system time. These statistics help us understand what components of the OS are really being exercised, and the relative importance of these components.

4.1 Observations

From these results, we make the following observations:

- As is to be expected with database applications, a large portion of execution time is spent in I/O. Disk operations are so dominating in some queries (Q1, Q8, Q12, Q17) that the CPU utilization does not cross 50% in these queries. I/O costs not only result in poor CPU utilization overall but also in significantly increasing the system call overhead itself. In some cases (such as Q12), the system CPU time (overheads) even exceeds the amount of time spent executing the useful work in the query at the user-level. The bulk of the execution time in the system mode is taken up by file system operations (pread/pwrite). Note that this system call overhead (system CPU time) does not include the disk latencies. Rather, this high overhead is due to memory copy-

ing, buffer space management and other book-keeping activities.

- Though the numbers are not explicitly given here, we would like to point out that the high read overheads are not only because of the higher number of file system read calls, but are also due to the higher cost per invocation of this call. We noticed that a pread call can run to nearly a millisecond in some queries. Of all the system calls considered, we found the per pread invocation taking the maximum amount of time.
- When we examine the CPU utilization during I/O (last column of Table 1), we find that there is some overlap of work with disk activity in many queries. However, there is still much room for improvement - only 5 queries out of 22 have more than 50% CPU utilization during I/O.
- After the file system calls, we found socket calls (select, socketcall) to be the next dominant system overhead. Sockets are used to transmit queries and results between the client machine and the server nodes. In addition, sockets are also used to exchange information between the server nodes during the processing of a query.
- Despite the dominance of I/O in many queries, queries like Q11 have a high CPU fraction (particularly in the user mode). Even though there are I/O operations in these queries, their costs are overshadowed by useful work (CPU utilization is around 66% even during periods of disk activity, and the bulk of it is in the user mode).

In summary, from the OS designer's viewpoint, file system I/O and message transfers using sockets are the two crucial services for optimization for this database engine (with the former issue being much more critical).

5 I/O Subsystem: Characterization and Possible Optimizations

We find that I/O is still the dominant component in many queries of TPC-H for clustered database servers. We now set out to look at the I/O subsystem more closely, trying to characterize its execution and look for possible optimizations. In the interest of space we specifically present results and suggestions for the read component of I/O (pread), which is usually much more dominant at least in decision support workloads, such as TPC-H.

5.1 Characteristics

Query	Q6	Q14	Q19	Q12	Q15	Q7	Q17
%	20.0	19.0	16.9	15.4	13.4	12.1	10.8
Query	Q8	Q10	Q1	Q13	Q3	Q4	Q18
%	10.5	10.3	10.0	10.0	9.6	9.1	9.0
Query	Q20	Q2	Q9	Q5	Q16	Q11	
%	7.9	5.2	5.2	4.6	4.1	3.5	

Table 2: pread as a percentage of total execution time

query	user CPU (%)	system CPU (%)	system CPU breakup (%)				page faults per jiffy	blocks read per jiffy	blocks written per jiffy	packets sent per jiffy	packets received per jiffy	CPU utilization during IO (%)
			pread	pwrite	select	ipc						
Q1	27.58	21.44	pread	pwrite	select	ipc	1.50	51.01	23.1151	0.0012	0.0015	26.87
			46.7	46.7	3.3	2.9						
Q2	56.22	15.67	socketcall	pread	select	pwrite	0.39	21.97	1.7718	1.9077	1.9248	53.73
			35.4	32.9	20.1	7.3						
Q3	40.76	17.76	pread	socketcall	select	pwrite	0.99	55.47	4.9591	0.9778	0.9938	55.40
			54.1	15.1	13.1	12.8						
Q4	51.48	15.19	pread	select	socketcall	pwrite	0.00	22.97	1.0478	0.3517	0.3652	68.41
			60.0	17.6	11.2	6.9						
Q5	58.96	15.68	socketcall	pread	select	pwrite	0.05	16.73	1.1369	2.0779	2.0849	42.81
			43.3	29.2	21.7	3.7						
Q6	40.65	22.20	pread	ipc	socketcall		1.73	90.72	0.0020	0.0012	0.0012	33.49
			90.1	4.9	4.3							
Q7	52.44	16.82	pread	pwrite	ipc	select	0.00	16.89	1.9467	2.3880	2.3521	31.04
			72.1	14.8	6.1	6.0						
Q8	20.65	17.71	pread	pwrite	select	ipc	0.01	27.63	4.8078	0.0261	0.0228	12.91
			59.5	23.7	9.4	5.8						
Q9	51.41	13.52	pwrite	pread	select	ipc	0.00	6.69	1.9276	0.0133	0.0136	23.21
			40.8	38.7	13.9	2.5						
Q10	41.79	17.87	pread	socketcall	select	pwrite	0.17	41.99	1.8880	0.8774	0.8859	18.71
			57.7	17.9	13.4	7.0						
Q11	81.00	13.28	socketcall	pread	select	ipc	0.49	18.87	0.0020	2.3794	2.4011	66.46
			43.6	27.0	24.8	3.9						
Q12	14.91	19.73	pread	selet	ipc		0.25	40.79	0.0197	0.0102	0.0101	4.8
			78.3	15.2	5.2							
Q13	53.23	21.86	pread	socketcall	select	ipc	1.62	45.86	0.0034	2.0966	2.0935	32.71
			45.7	30.3	18.8	4.6						
Q14	33.57	22.19	pread	select	ipc		1.01	84.78	0.0025	0.1156	0.1175	29.75
			85.6	8.3	5.1							
Q15	55.37	18.69	pread	select	ipc	nanosleep	0.60	75.26	0.0033	0.6260	0.7703	51.68
			71.7	14.0	10.9	1.9						
Q16	51.84	15.30	socketcall	pread	select	ipc	2.32	15.36	0.8454	2.4836	2.4993	46.24
			43.5	27.0	22.2	5.7						
Q17	23.71	18.26	pread	pwrite	select	ipc	0.00	32.76	5.2532	0.0036	0.0037	13.94
			59.4	26.1	8.7	4.9						
Q18	52.64	14.77	pread	socketcall	select	pwrite	0.04	17.16	0.6261	0.3890	0.3803	35.11
			61.1	17.3	13.3	5.2						
Q19	35.48	21.16	pread	select	ipc	socketcall	0.29	78.76	0.0032	0.3343	0.3329	23.38
			80.1	7.3	5.9	5.9						
Q20	56.98	14.72	pread	select	socketcall	ipc	0.00	15.74	0.1601	0.3456	0.2973	47.36
			53.4	25.6	15.3	3.6						

Table 1: System Profile (statistics are collected from node 1)

Table 2 sorts the queries in decreasing order based on the fraction of total query execution time spent in the pread system call obtained from earlier profile results. We can see that pread is a significant portion of the execution time in many queries. It takes over 10% of the execution time in 11 of the queries. It should be noted that this is the time spent in the system call (i.e. in buffer management, book-keeping, copying across protections domains, etc.), and does not include the disk costs itself. This implies that it is *not only important to lower or hide disk access costs, but to optimize the pread system call itself*. In the interest of space, we focus on queries Q6 and Q14 which incur the maximum pread overhead in the rest of this section (the trends/arguments are similar for the others).

Before presenting the characterization results for these queries, we would like to briefly explain how we believe the reads are invoked in the query execution. The database engine has several agent processes that actually perform the work in the queries, and prefetcher processes that make pread calls to bring in data from disk ahead of when the agent may need it. In addition to the prefetcher reads that are usually invoked 32 blocks (i.e. $32 * 4K = 128K$ bytes) at a time, the agent also occasionally invokes pread calls, and we find these requests are usually for individual blocks (4096

bytes). Figures 1 (a), (b) pictorially show these observations for the two queries when one looks at the Cumulative Density Function (CDF) of read request sizes that are issued by the prefetcher and the agents individually. The other graphs (c), (d) in the same figure show the CDF of the temporal separation of the pread requests. We observe that the agent reads are more bursty, coming in closer proximity, than the reads issued by the prefetcher.

We found that the preads issued by the agent are usually for a block that has been recently read by the prefetcher just before that invocation. It may come as a surprise as to why this block could not have been serviced by the prefetcher directly (if it was only recently read), instead of going to the kernel. One possible explanation is that the agent is doing a write on this block, and it may not want to write into a page that is residing in the prefetcher. Instead of making ipc calls to remove the page from the prefetcher, it would be better to create a copy within the agent by using a pread call directly. For further credibility on this hypothesis, before returning from pread calls, we modified the kernel to set the corresponding data pages to be read-only mode, and we found the agent to incur (write) segmentation faults (indicated as copy-on-write in Table 4) on nearly all those pages (compare the copy-to-user and copy-on-write columns for the agent in Ta-

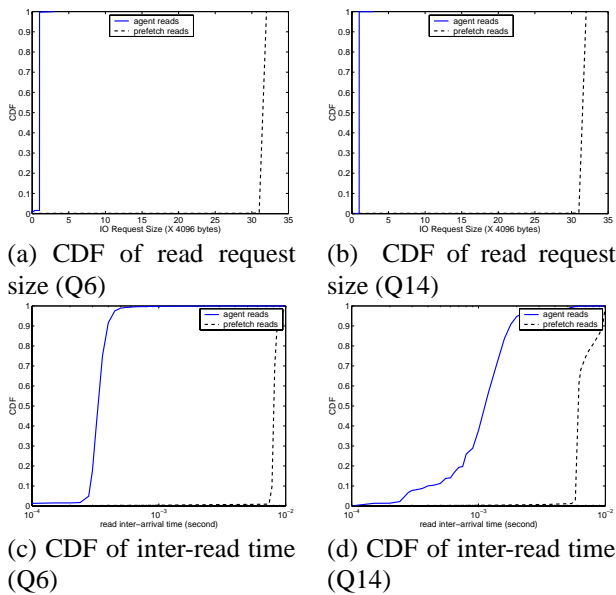


Figure 1: IO characterization results in terms of read request size and time between successive reads.

ble 4). Finally, it should be noted that the agent pread calls are much lower (both in terms of the number of calls and in terms of the number of blocks read) than those for the prefetcher.

Query	prefetcher hit ratio	agent hit ratio	Query	prefetcher hit ratio	agent hit ratio
Q1	0.5711	1.0000	Q11	0.2788	1.0000
Q2	0.5321	1.0000	Q12	0.4735	1.0000
Q3	0.5164	1.0000	Q13	0.5261	1.0000
Q4	0.4873	1.0000	Q14	0.5344	1.0000
Q5	0.5991	1.0000	Q15	0.4227	1.0000
Q6	0.4729	1.0000	Q16	0.4409	1.0000
Q7	0.5182	1.0000	Q17	0.7365	1.0000
Q8	0.5300	1.0000	Q18	0.4226	1.0000
Q9	0.4683	1.0000	Q19	0.5625	1.0000
Q10	0.5082	1.0000	Q20	0.5453	1.0000

Table 3: Fraction of block requests that hit in Linux file cache for prefetcher and agent requests

We also include in Table 3 the fraction of pread block requests that hit in the Linux file cache for the prefetcher and the agents. As was pointed out, the agent requests come very soon after the prefetcher request for the same block, and thus nearly always hit in the Linux file cache. With the prefetcher requests on the other hand, we find the file cache hits range between 40-60%. We mentioned earlier that the prefetcher requests are usually for 32 blocks at a time. The Linux file cache manager itself does some read ahead optimizations based on application behavior and brings in 64 blocks (twice this size). With a lot of regularity (sequentiality) in I/O request behavior for this workload, this read ahead tends to cut down the number of disk accesses by around 50%, which actually indicates that Linux file caching is quite useful in supporting the database application.

5.2 Recommendations and Possible Optimizations

A significant portion of pread cost is expended in copying data (that is in a block in the file cache, either already or brought in upon disk I/O completion), from the kernel file cache to a user page, which needs to cross a protection-domain boundary (using the *copy-to-user()* mechanism). In the current 2.4.8 Linux implementation, a copy is actually made at this time (a patch addressing this is available [6]).

This problem of reducing copying overheads for I/O has been looked at by several previous studies [19, 12, 17]. There are different techniques one could use, and a common one (which is used by Linux itself in several other situations to reduce copying) is to simply set the user page table pointer to the buffer in the file cache. This could affect the semantics of the pread operation in some cases, particularly when more than one user process reads the same block. In the normal semantic, once the copy is done, a process can make updates to it without another seeing it, while the updates would be visible without copies. This is usually addressed (as is by Linux in several other situations) by the *copy-on-write* mechanism, i.e. map the frame into user space, but make it read-only. If the user process does write into it, a segmentation fault is incurred and at this time we actually copy the data from the file cache buffer into another frame and update the process page table accordingly. Some studies [12, 19, 17] suggest that even this may not be very efficient since updating virtual address mappings can become as expensive as copying. Instead, sharing of buffers between user and kernel domains is advocated. However, this may require extensions to the kernel interface, and application exploitation of this extension, which as we pointed out early on can become cumbersome for legacy applications.

To examine the potential benefits of such an implementation, we track the total number of *copy-to-user()* calls that are made (actually one for each page) and the number of these calls that cannot be avoided (you cannot avoid it when there is a write segment violation and we need to do a *copy-to-user* at that time), during the execution of these queries after setting these pages to read-only mode. These numbers are shown in Table 4. As we can observe, the number of *copy-on-writes* that are actually needed is much lower than the number of *copy-to-user* invocations, as was suspected initially. In general, we get no less than 65% savings in the number of copies, with actual savings greater than 80% for most queries (see the last column of this table). Most of these savings are due to the prefetcher reads. Our measurements of *copy-to-user* routine for a single block using the high resolution timer takes around 30 microseconds for one page. For 32 block reads that the prefetcher issues, avoiding this cost can be a significant savings. By avoiding these copies, we can use file system to achieve the performance of using raw device, without all the disadvantages of using raw device. This is particularly true when the blocks hit in the file cache (and there is no disk I/O) since this cost is a significant portion of the overall time required to return back to the application. Table 3 shows that this happens nearly 50% of the time. Even with disk activity, Table 1 shows CPU utilization higher than 50% in most queries, suggesting that removing this burden of copying by the CPU would help query execution.

Query	prefetcher			agent			total
	copy-on-write	copy-to-user	% Reduction of copies	copy-on-write	copy-to-user	% Reduction of copies	% Reduction of copies
Q1	0	1040228	100	11551	11565	1.2	98.9
Q2	0	383334	100	63145	63145	0	85.7
Q3	0	1253107	100	52155	52157	0.003	96.0
Q4	0	997507	100	235758	235759	0.0004	80.9
Q5	0	1007919	100	307689	307689	0	100.0
Q6	0	914482	100	47	47	0	100.0
Q7	0	1084454	100	276790	276791	0.0003	79.7
Q8	0	978134	100	255057	255060	0.001	79.3
Q9	0	2478154	100	316500	316502	0.0006	88.7
Q10	0	974062	100	278213	278215	0.0007	77.8
Q11	0	170643	100	6833	6834	0.01	96.1
Q12	0	911544	100	134933	134933	0	87.1
Q13	0	184491	100	42	43	2.3	100.0
Q14	0	945619	100	38429	38430	0.003	96.1
Q15	0	1175166	100	38394	38395	0.003	96.8
Q16	0	36137	100	15001	15003	0.01	70.7
Q17	0	1968122	100	113962	113963	0.0008	94.5
Q18	0	1945777	100	502	503	0.19	100.0
Q19	0	865529	100	38429	38429	0	95.7
Q20	0	847755	100	50442	50444	0.003	94.4

Table 4: % of copy-to-user calls that can be avoided. Of the given copy-to-user calls, only the number shown under the copy-on-write are actually needed. The statistics are given for the prefetcher and agents separately, as well as the overall savings.

6 Network Subsystem: Characterization and Possible Optimizations

6.1 Characteristics

We next move on to the other exercised system service, namely TCP socket communication. As in the earlier section, we first attempt to characterize this service based on certain metrics that we feel are important for optimization. We examine the message exchanges based on the following characteristics: the message sizes, the inter-injection time, and the destination for a message. We present these characteristics using density functions.

In the interest of space, we show these characteristics pictorially in Figure 2 for two queries (Q7 and Q11), which have the highest message injection rates shown in Table 1. These results have been obtained by instrumenting the kernel and logging all the socket events, their timestamps and arguments at the system call interface. As will be pointed out later on, for some characteristics we also needed to log messages themselves or at least their checksums.

From the density function graphs, we observe the following:

- The message length CDF graph shows that just a handful of message sizes are used by DB2. In fact, we observed messages were usually either 56 bytes or 4000 bytes. We hypothesize that the shorter size (56 bytes) is used for control messages, and the larger size (4000 bytes) is used for actual data packets.
- There are many messages that are sent out in close proximity (temporally). In fact, Figures 2 (c),(d) show that nearly 60% of the messages are separated by less than 1 millisecond from each other temporally. In fact, the temporal separations are much lower for queries Q2, Q7, Q10, Q13, Q14, Q15, Q16, Q18 and Q19. In Q7, which is shown in Figure 2(c), while there are a few messages that are farther apart, nearly 90% of the mes-

sages are within 1 millisecond. We found similar observations for most of the other queries as well. Communication usually occurs after processing some data from disk, which takes some time during which there are periods of network inactivity.

- The destination PDF graph is considerably influenced by the nature of database operations. In DB2, joins usually involve all-to-all communication of their corresponding portions of the table, and thus queries that are join intensive (such as Q11 that is shown here) have the PDF evenly distributed across the nodes. There are a few queries, such as Q7, that are not really join intensive, but perform more specific operations that are based on values of certain primary keys. With such executions, there is a slight bias in communication towards nodes that have those values.

6.2 Recommendations and Possible Optimizations

The above message characteristics say that messages are clustered together, often coming in close temporal proximity. Further, database operations such as joins use all-to-all communication of messages which have a high probability of being the same size. These observations suggest an hypothesis that many of these messages may actually be point-to-point implementations of a multicast/broadcast that the database engine would like to perform. It should be noted that a multicast can send the same information to several nodes at a much lower cost than sending individual point-to-point messages. This saves several overheads at a node (copies, packetization, protocol header compositions, buffer management, etc.) and can also reduce network traffic/congestion if the hardware supported it. Possible implementations of multicast are discussed later in this discussion.

To find out how many of the message exchanges can be modeled as multicasts, we investigated several approaches. During the execution, in addition to the above events, we also logged the messages themselves. These logs were then

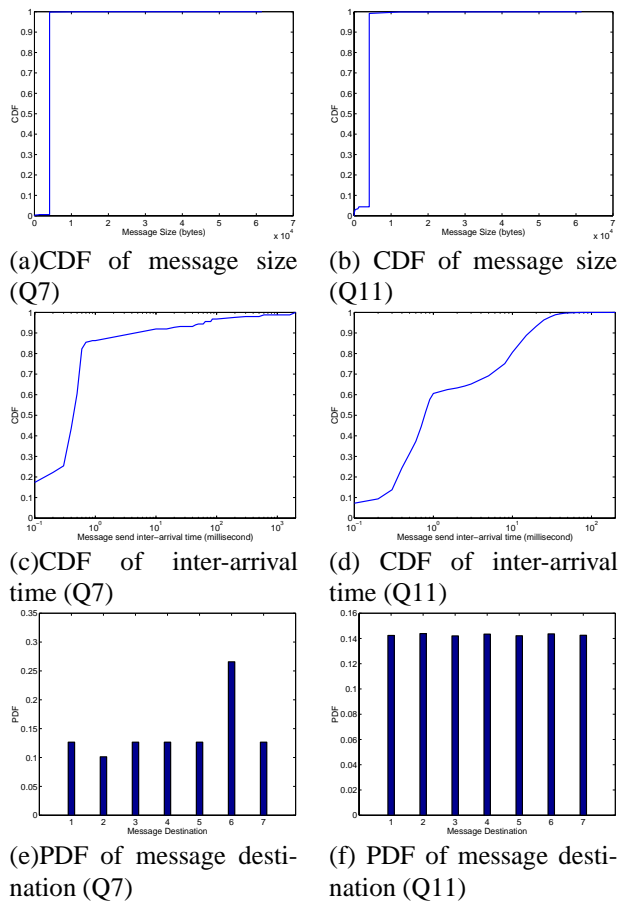


Figure 2: Characterizing Message Sends

subsequently processed to compare whether successive messages were identical and addressed to different destinations. Another approach that we tried (which is actually a possible one to use within the OS during the course of execution itself to detect multicasts) is to compare checksums of successive messages. We found that both the approaches - actually comparing the messages or comparing the checksums - gave us similar results, and Table 5 gives the percentage of reduction in the number of messages that would be sent if the underlying infrastructure supported multicasts. This information is given for both the short and long messages to verify if multicasts are beneficial to any one class of messages or for both.

query	% save of total msg	% save of small msg	% save of large msg	query	% save of total msg	% save of small msg	% save of large msg
Q1	44.7	71.4	38.7	Q11	9.6	28.6	0.1
Q2	20.4	58.7	0.2	Q12	8.3	7.8	2.9
Q3	48.2	64.3	38.0	Q13	24.5	75.2	0.1
Q4	22.6	58.6	0.1	Q14	27.9	80.4	0.7
Q5	8.0	7.1	8.4	Q15	46.6	56.5	0.7
Q6	76.4	78.6	45.5	Q16	59.1	63.0	56.9
Q7	57.5	71.4	56.2	Q17	41.5	66.7	27.3
Q8	29.1	75.5	4.8	Q18	11.4	32.3	0.00
Q9	66.8	78.5	61.1	Q19	26.7	79.4	0.2
Q10	25.0	73.6	0.1	Q20	21.1	62.8	0.1

Table 5: The potential impact of multicast on queries

We find that there is a substantial multicast potential in these queries. There is a reduction in the total number of messages ranging from 8% to as high as 76%. This potential can be realized only if the underlying network supports multicasts (incidentally, we found a large number of these multicasts are in fact broadcasts, which Ethernet can support). Even assuming that the underlying infrastructure (either at the network interface level, or in the physical network implementation) supports multicast, the message exchanges should be injected into this infrastructure as multicast messages. This can be done at two levels. First, the application (i.e. the database engine) can itself inject multicast messages into the system. The other approach, which we investigate, is to automatically detect multicast messages within the operating system (or middleware before going to sockets) and perform the optimizations accordingly. We next describe an online mechanism for such automatic detection.

The online algorithm in the OS or middleware can make the system wait for a certain time window while collecting messages detected as multicasts, without actually sending these out. At the end of this window, we send a single multicast message for all the corresponding destinations of the saved messages. The advantage with this approach is that we do not send a message to a destination that the application does not send to. The drawback is that the time between successive messages and time window may be too long a wait that it may be better off just sending them as point-to-point messages. Further, if the window is not long enough, we may not detect some multicasts, and end up sending point-to-point messages.

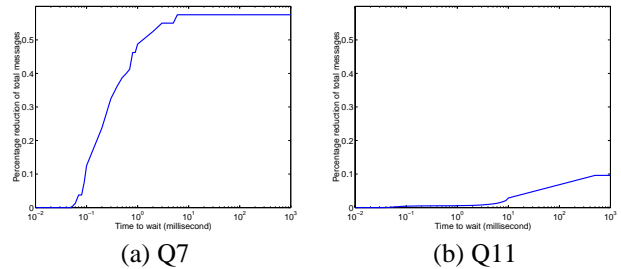


Figure 3: The impact of wait time on multicast message detection

Consequently, it is important to understand the impact of window size on multicast potential with this online algorithm. If we use such an algorithm within the OS/middleware, then the percentage reduction in the number of messages that need to be sent out with this approach is given in Figure 3 as a function of the time window that it waits for Q7, Q11, and Q16. As is to be expected, expanding the window captures a large fraction of multicasts until the benefits taper off. However, one cannot keep expanding a window arbitrarily since this can slow down the application's forward progress in case this message is needed immediately at the destination. We noticed that the TCP socket implementation on the underlying platform had one-way end-to-end latencies of around 100 microseconds. So it is not unreasonable to wait for comparable time windows since the message would anyway take a large fraction of that time to

leave that node. If we consider, window wait times of say 500 microseconds, then we can see reduction of around 40% of the messages for Q7. On the other hand, Q11 does not benefit much from such an online algorithm (nor from the offline algorithm). We found that queries Q6, Q7, Q9, Q15, Q16, Q17 (the graphs are not explicitly given here) had at least 15% message reduction with a wait time of 500 microseconds.

7 Summary of Results and Concluding Remarks

Clustered database services that are being offered to host applications are becoming increasingly popular in medium and large scale enterprises, to meet the needs of demanding queries, voluminous datasets and the large number of concurrent users. The understanding of the interactions between these applications and the underlying OS is critical to provide better system and architectural support for such system software. This is the first study to embark on such a characterization and to present a range of performance statistics for the execution of TPC-H queries, an important decision support workload for enterprises, on a medium sized Linux cluster of SMP nodes (a popular configuration in today's commercial market) connected by Myrinet and Ethernet.

Moving from a uniprocessor/SMP to a cluster does not make I/O any less important for a database engine. We find that disk activity can push CPU utilization as low as 30% in some queries. The overhead of I/O is not just because of the disk latencies, and a significant portion is in the pread system call itself (copying costs mainly). We find that it is extremely important to optimize the pread system call itself, by reducing the amount of copying. There are several known techniques for reducing copying costs, and in this study we examined the virtual address remapping scheme to show how many copies can actually be avoided. By employing such a scheme, we can utilize the benefits given by file system (such as ease to manage, structured dataset, etc), while avoiding the disadvantages of it (file caching).

The other system service that is also exercised is the socket communication to exchange control and data messages amongst the nodes. We find messages are often bunched together, and there are two main message sizes (56 or 4000 bytes). Further, many of these messages are identical, suggesting potential for multicasts/broadcasts, which the database engine implements as point-to-point messages. We find that a significant number of messages (particularly the shorter ones) can be reduced by multicasts.

It should be noted that our goal in this paper is not to recommend specific implementations or designs for improving performance. Rather, we are trying to identify characteristics of application-OS interactions and to suggest issues that can help improve performance for this workload.

References

- [1] DB2 Product Family. <http://www-3.ibm.com/software/data/db2/udb/downloads.html#eeelinix>.
- [2] GLUnix: A Global Layer Unix for a Network of Workstations. <http://now.cs.berkeley.edu/Glunix/glunix.html>.
- [3] High-Performance Database Requirements. <http://lwn.net/2001/features/KernelSummit/>.
- [4] IBM DB2. <http://www.ibm.com/db2>.
- [5] IBM Informix Extended Parallel Server (XPS). <http://www-4.ibm.com/software/data/informix/xps/>.
- [6] Kernel Patch kiobuf. <http://www-124.ibm.com/pipermail/evms/2001-January/000039.html>.
- [7] Microsoft SQL server 2000. <http://www.microsoft.com/sql/default.asp>.
- [8] Oracle 9i Real Application Clusters. <http://www.oracle.com/ip/index.html>.
- [9] TPC-H Benchmark. <http://www.tpc.org/tpch>.
- [10] J. Catozzi and S. Rabinovici. Operating System Extensions for the Teradata Parallel VLDB. In *Proceedings of Very Large Databases Conference*, pages 679–682, 2001.
- [11] P. Christmann, T. Harder, K. Meyer-Wegener, and A. Sikeler. Which Kinds of OS Mechanisms Should Be Provided for Database Management. In *Experiences with Distributed Systems*, pages 213–251. J. Nehmer (ed.), Springer-Verlag, 1987.
- [12] P. Druschel and L. L. Peterson. Fbufs: A highbandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, 1993.
- [13] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*. In *Lecture Notes in Computer Science 60*, pages 393–481. Springer-Verlag, 1978.
- [14] W. W. Hsu, A. J. Smith, and H. C. Young. Analysis of the Characteristics of Production Database Workloads and Comparison with the TPC Benchmarks. *IBM Systems Journal*, 40(3), 2001.
- [15] W. W. Hsu, A. J. Smith, and H. C. Young. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks - An Analysis at the Logical Level. *To appear in ACM Transactions on Database Systems*, 2001.
- [16] M. A. Kandaswamy and R. L. Knighten. I/O Phase Characterization of TPC-H Query Operations. In *Proceedings of the 4th International Computer Performance and Dependability Symposium*, March 2000.
- [17] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1), 2000.
- [18] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.
- [19] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.
- [20] I. L. Traiger. Virtual memory management for database systems. *Operating Systems Review*, 16(4):26–48, 1982.
- [21] S. Venkatarman. Global memory management for multi-server database systems. Technical Report CS-TR-1996-1325, Univ. of Wisconsin, 1996.