INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

PERFORMANCE EVALUATION OF
CONCURRENT OLTP AND DSS
WORKLOADS IN A SINGLE DATABASE
SYSTEM

by

James R. Paulin, B.A., M.B.A.

A thesis submitted to the Faculty of Graduate
Studies and Research in partial fulfilment of the
requirements for the degree of

Master of Science

Information and Systems Science

School of Computer Science

Carleton University

Ottawa, Ontario

November 1, 1997

© Copyright

1997, James R. Paulin

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27065-3

Canada

The undersigned hereby recommend to the Faculty of
Graduate Studies and Research acceptance of the thesis,

PERFORMANCE EVALUATION OF CONCURRENT
OLTP AND DSS WORKLOADS IN A SINGLE
DATABASE SYSTEM

submitted by

James R. Paulin, B.A., M.B.A.

in partial fulfilment of the requirements for the degree of

Master of Science,

Information and System Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

Ottawa, Ontario

Date

ABSTRACT

PERFORMANCE EVALUATION OF CONCURRENT OLTP AND DSS WORKLOADS IN A SINGLE DATABASE SYSTEM

by James R. Paulin, B.A., M.B.A.

Supervisor:
Professor S. Dandamudi
School of Computer Science
Carleton University

To date most organisations have been unwilling to allow decision support (DSS) and online transaction processing (OLTP) applications to execute concurrently against the same database. The main concern has been that the performance impact of demanding DSS queries on short OLTP transactions would be unacceptable. This thesis reports the results of a set of experiments designed to evaluate the performance of mixed OLTP and DSS workloads. The experiments were conducted on a commercial relational database management system using generated test data and embedded SQL applications that were designed to be representative of real-world environments. The thesis also examines alternative strategies for mixed workload execution, particularly with respect to locking issues, and reviews proposals in the literature for new database capabilities intended to support mixed workloads. We conclude that in the right circumstances performance issues should not deter users from executing mixed workloads against a single database system even with today's technology.

ACKNOWLEDGEMENTS

I am indebted to my supervisor, Professor S. Dandamudi, for his constant and timely support throughout the research for and preparation of this thesis. I would also like to thank Professor E. Otoo of the School of Computer Science for his assistance installing the database management system used for the performance experiments.

This thesis would never have seen the light of day without the encouragement and patience of my wife, Elspeth, and my children, Hilary and Clare.

CONTENTS

ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
CONTENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	ix
INTRODUCTION	1
1.1 The Problem	1
1.2 OLTP and DSS	3
1.3 The Batch Reporting Solution	5
1.4 The Data Warehouse Solution	6
1.5 The Mixed Workload Solution	8
1.6 Thesis Approach	10
1.7 Contribution	12
1.8 Issues not Addressed and Final Remarks	13
MIXED WORKLOAD STRATEGIES	16
2.1 Business Issues	16
2.2 Query Management	18
2.3 Unacceptable Solutions	21
2.4 The Read Uncommitted Isolation Level	22
2.5 Partitioned Data	26
2.6 Enhanced Concurrency Control Schemes	32
2.7 Strategies Tested	34
METHODOLOGY	36
3.1 Overview	36
3.2 Test Databases and Hardware	37
3.3 Test Data	38
3.4 Test Transactions, Queries and Programs	47
3.5 Workloads and Test Execution	55
3.6 Metrics	67
RESULTS AND ANALYSIS	75
4.1 Preliminary Results	75

4.1.1 DBMS Issues	75
4.1.2 Hot Spot Queries in the Unified Environment	78
4.1.3 Similarity of Different Mixed Workloads	80
4.2 Sar Results	81
4.2.1 Standalone Workloads	82
4.2.2 Hot Spot Queries	87
4.2.3 Mixed Workloads	88
4.3 Performance Results	91
4.3.1 Heavy Transaction Workloads and Execution Priorities	92
4.3.2 Impact of Queries on Transaction Response Times	97
4.3.3 Impact of Queries on Transaction Throughput	108
4.3.4 Impact of Transactions on Queries	110
4.3.5 Hot Spot Query Stream in the Split Environment	115
4.3.6 Response Time Variability	120
4.3.7 Results by Transaction Type	123
4.4 Mixed Workload Efficiency	126
4.5 Split versus Unified Data	133
4.6 Concluding Comments on the Results	138
RELATED WORK	143
5.1 Mixed Workload Proposals	143
5.2 Stable Version Scheme	144
5.3 Dynamic Finite Versioning	152
5.4 Version per Query Scheme	157
5.5 Comparison of the Transient Versioning Schemes	163
5.6 Enhanced Serialisability	165
5.7 Other Related Work	171
5.7.1 Oracle's Read Consistency	171
5.7.2 Temporal Databases	173
5.7.3 Cursor Stability	175
CONCLUSIONS AND FUTURE WORK	177
6.1 Summary of Results and Conclusions	177
6.2 Key Issues for Mixed Workloads	183
6.2.1 Business Issues	183
6.2.2 Technical Issues	184
6.3 Future Work	185
6.4 Final Words	187
APPENDIX A: SAR RESULTS	189
APPENDIX B: PERFORMANCE RESULTS	191
APPENDIX C: TABLE DEFINITIONS	202

APPENDIX D: TRANSACTION AND QUERY SQL	206
D.1 Transaction SQL	206
D.2 Query SQL	211
APPENDIX E: SAMPLE PROGRAMS	216
E.1 t1.sc: Sample Transaction Program	216
E.2 qset3.sc: Sample Query Program	226
REFERENCES	232

LIST OF FIGURES

Number	Page
Figure 1: Database model of a health insurance agency	40
Figure 2: Driver program output from a typical test execution	67
Figure 3: CPU utilisation for standalone workloads	83
Figure 4: Data disk utilisation for standalone workloads	85
Figure 5: Memory utilisation for standalone workloads	86
Figure 6: Response time increases for medium and light transaction workloads	98
Figure 7: Absolute transaction response times	100
Figure 8: Unified versus split data response time increases	102
Figure 9: Absolute transaction response times in the split and unified environment	103
Figure 10: Impact of transactions on query elapsed time	111
Figure 11: Absolute execution times for 3-up queries	112
Figure 12: Absolute execution times for a single query stream	113
Figure 13: Response time increase caused by generic and hot spot queries	116
Figure 14: Absolute response times with generic and hot spot queries	117
Figure 15: Impact of transactions on generic versus hot spot query execution times	118
Figure 16: Impact of queries on response time variability	121
Figure 17: Version record for a primary key value	146
Figure 18: An example of timing relationships in the stable version scheme	147
Figure 19: An example of timing relationships with Dynamic Finite Versioning	155
Figure 20: Bober and Carey's version per query scheme	160

LIST OF TABLES

Number	Page
Table 1: Row lengths and number of rows in the test tables	42
Table 2: Unified data: Impact of queries on transactions	92
Table 3: Split data: Impact of queries on transactions	93
Table 4: Impact of transactions on queries	94
Table 5: Selected performance results by transaction type for unified data	124
Table 6: Selected performance results by transaction type for split data	125
Table 7: Mixed workload efficiency	130

Chapter 1

INTRODUCTION

“In sum, I think the OLTP/decision support split is a good one and a permanent one.”

Robert Epstein, Executive Vice President and cofounder of Sybase, Inc.,
quoted in *Database Programming and Design*, Vol. 10, No. 7, July, 1997,
p.54.

1.1 The Problem

Databases were first invented so that organisations could use one copy of their data to satisfy the needs of different applications rather than having each application maintain a separate copy for its own purposes. At first, databases were primarily used for online transaction processing (OLTP) systems since transactions were the lifeblood of most businesses. More recently, users have increasingly come to expect databases to answer arbitrarily demanding and complex queries in support of decision-making functions. Such query applications are known as decision support systems (DSS).

By and large, however, organisations have been reluctant to attempt using a single database system to concurrently process both transactions and queries. The chief concern has been that mixed workloads would not provide acceptable performance, particularly for the OLTP applications. Would not the addition of DSS queries render the closely guarded transaction processing environments unmanageable?

Before proceeding any further, our use of the terms ‘transaction’ and ‘query’ must be clarified. The word ‘transaction’ has a specific, technical meaning in database theory. It refers to a logical unit of work which respects the ACID properties: atomicity, consistency, isolation and durability. In this technical sense a query is also a transaction, however long it may run. In particular, databases must execute queries at the requested isolation level. In this thesis, however, we normally use the term ‘transaction’ to refer to OLTP-style transactions, which is the meaning most users of commercial information systems attach to the word. When we occasionally use ‘transaction’ in its technical sense that will be made clear either explicitly or from the context. Similarly, the term ‘query’ is normally used here to refer specifically to a DSS query, even though some of the OLTP transactions in the performance experiments were also queries.

Given this particular interpretation of the terminology, our goal is to understand what in fact does happen when a mixed workload of transactions and queries is processed in the same database system at the same time. Are mixed workloads feasible at all? Or should organisations continue to segregate their transaction and query processing applications?

In the course of examining the feasibility of mixed workloads, a number of subsidiary issues will be addressed. Among them are the following:

- What strategies are available for executing mixed workloads in a single database system? How do they compare from a performance point of view?

- What is the impact of concurrent queries on transaction response time and response time variability?
- How do concurrent transactions impact query execution?
- How efficiently do mixed workloads make use of system resources?
- What happens if many of the queries and transactions target the same subset of the data? (The 'hot spot' question.)
- What are the key issues that will determine the success or failure of mixed workloads?

1.2 OLTP and DSS

Online transaction processing applications are characterised by high volumes of pre-defined transactions which typically execute in a few seconds and exhibit low response time variability. Normally the transactions are submitted by human users who wait for the system to respond. Hence the importance of fast and consistent responses from the database system. Each transaction typically executes a business function: hires a new employee, makes a bank account deposit, reserves a hotel room, and so on. OLTP is the traditional, essential function of commercial information systems.

Compared to OLTP applications, DSS queries represent a very different type of workload for database systems. An example DSS query for a health insurance agency would be to find the 100 most costly beneficiaries over the preceding 12 month period out

of, say, the 7 million beneficiaries, which means scanning, summing and sorting perhaps hundreds of millions of payment records.

Where transactions are pre-defined, queries are often if not primarily ad hoc: users do not know what queries they will submit in advance. New queries are being asked all the time. Many queries are only submitted once. While transactions normally complete in a matter of seconds, queries can take hours, even in massively parallel processing systems. Transactions usually access a small number of data blocks, and use indexes to retrieve the precise data they need, while queries routinely read entire tables, and process complex joins that transactions can rarely afford. How could any database management system (DBMS) be expected to execute two such disparate workloads concurrently and provide acceptable performance to both? Surely it would be better to segregate them and tune the segregated databases for their respective workloads.

Such in fact has been the traditional view. Most system managers are extremely protective of their OLTP systems, warding off any possible interference that could adversely affect the length or the variability of transaction response time. The author is not aware of any sizeable database systems that execute concurrent OLTP and DSS workloads. The quote at the beginning of this chapter from a leading authority on database systems echoes the conventional opinion that OLTP and DSS applications mix like oil and water.

It should also be noted that OLTP and DSS systems tend to serve different user communities. On the whole, OLTP systems have automated 'clerical' functions, the

automated teller machine being a classic example. DSS systems, on the other hand, are primarily the domain of the 'knowledge worker' or decision maker. Here, an example is the buyer for a retail store chain who uses a variety of DSS reports and ad hoc queries to track product sales and movement and to decide what to purchase, in what region, and at what price it should be sold.

1.3 The Batch Reporting Solution

For many years, organisations have avoided mixed workloads by restricting the OLTP and DSS applications to different times of the day, and possibly days of the week. Normally, the OLTP system would run during business hours while batch reports would be executed during the night and possibly on weekends. In this way, the database would never be faced with a mixed workload and, in particular, transaction performance would not be adversely affected by concurrent queries.

Batch reporting overnight has satisfied many needs for information. But it has some significant limitations. The most important is that it presumes that a batch processing 'window' is available: a time slot when the OLTP system is inactive and reports can be executed. Unfortunately, such batch windows are being squeezed more severely every year as organisations expect their OLTP systems to be available for longer hours. Applications such as automated banking machine systems never shut down. And even if a small batch window is available there are frequently other tasks that must be performed in that time slot such as database backups or reorganisations.

The other major problem with batch reporting is that it is a far cry from the promise of Decision Support Systems. Reports tend to be outdated, static, difficult to develop, generic, and even irrelevant to the issues of the day. Knowledge workers do not want to wait until the next day or next week for information they need right away. They want to be able to create queries on the fly in response to specific issues they face. Responsiveness and flexibility are not characteristic of cumbersome batch reporting systems. Queries often need to be refined through several iterations, a process which can take weeks using batch reporting.

In effect, restricting knowledge workers to batch reporting severely limits their access to the corporate information resource. They require much more immediate and flexible access in order to truly take advantage of the data.

1.4 The Data Warehouse Solution

In the last few years, data warehouses have become a popular solution for Decision Support Systems. Organisations have taken to copying some or all of the data in their OLTP databases into data warehouse platforms of various types, and then providing a number of tools and applications to enable their employees to run both ad hoc and pre-defined queries against the data warehouse.

The essential notion here is that a second copy of the data is created for query purposes, and so once again the OLTP database is spared the trauma of having to concurrently execute queries and transactions. The data warehouse is typically a relational

database, and large warehouses normally employ parallel processing hardware and software. But there are also a number of vendors offering data warehouse products that are not databases at all in that they do not support the notion of transactions, and in particular concurrency control via locking. The assumption is that the data is read-only and so locking is unnecessary overhead.

There have been many reasons for building data warehouses. Many OLTP databases are still non-relational and hence difficult to use for query purposes. Often a data warehouse integrates data from a number of separate databases and so makes it relatively easy to obtain a global view of the organisation's activities. Data Warehouses will often accumulate several years of data to provide users with a historical perspective, whereas most OLTP systems keep only relatively recent data.

Of course, data warehouses also overcome the limitations of batch reporting. Users no longer need to be restricted to a shrinking batch window since OLTP applications do not run on data warehouses. Moreover, knowledge workers can submit ad hoc queries against the data warehouse directly and, depending on the size of the query, can usually expect an answer in the course of the business day, or at worst overnight.

However, data warehouses come at a price. In almost all cases they run on a separate hardware platform from the OLTP system precisely so as not to interfere with OLTP performance. There are additional software licence costs. But the biggest challenge is developing mechanisms for extracting data from the OLTP systems and

loading it into the data warehouse. The author has considerable experience with extracts and loads and can attest to the fact that they are non-trivial, expensive, and subject to various types of errors. The infamous batch window problem raises its head once again: will the OLTP system shut down long enough to permit the extracts to run? If the extract is run against the log instead so as to capture just the most recent database changes, how does one cope with rollbacks in the OLTP system?

Since the data in the data warehouse is a copy of the 'live' OLTP data, there is always the issue of its timeliness. Some users populate their data warehouses monthly, which clearly means that the data can be up to a month out of date. It is rare for a Data Warehouse to be updated more frequently than once a day.

1.5 The Mixed Workload Solution

Given the drawbacks of batch reporting and data warehouses, the obvious question is: why not run both OLTP and DSS applications on the same database at the same time? In this thesis, we refer to this scenario as a 'mixed workload'. The traditional answer to the question has been that OLTP applications cannot afford the adverse performance impact of concurrent DSS queries.

Of course, there are other reasons for not mixing OLTP and DSS workloads. If the OLTP system is running on a navigational DBMS then ad hoc queries are generally not feasible. If it is essential to integrate data from diverse OLTP databases then it may be necessary to build a data warehouse. This will be the case, for example, when the OLTP

data is located in hundreds or thousands of retail store systems, and must be centralised to make head-office queries of sales activity possible.

Another major reason for avoiding mixed workloads is that businesses do not want to run queries against data that is changing. If the query 'locks' the data being read then transactions are blocked for the duration of the query which is unacceptable. If the query reads data that is being modified as it executes then the validity of the answer may be in doubt. Or again, some organisations need to be able to run a number of queries against data that does not change in order to be able to properly analyse it; they need a snapshot of the data at a point in time rather than a live database.

However, there is a large class of users that do not fall into any of these categories. They have built numerous OLTP systems based on relational databases, some of them quite sizeable. The data resource is there and they would like to use it to satisfy at least some of the demand for DSS queries as well as the traditional OLTP function. In a large number of cases, queries can in fact be executed against the data without locks because approximate answers are acceptable. And, of course, organisations are building new OLTP databases all the time. Should they be designed for the OLTP workload only? Or should they be expected to support DSS-style queries as well?

This brings us back to the traditional objection to mixed workloads which is the central issue of this thesis: what is the impact of mixed workloads on the performance of OLTP transactions and DSS queries? That is, should one contemplate mixed workloads at all or will they have such an adverse impact on the transactions in particular that they

are simply not feasible? If mixed workloads cause transaction response times to lengthen unacceptably or to become highly variable then users will insist that the queries be removed.

1.6 Thesis Approach

The heart of this thesis is a set of experiments with mixed workloads for which we report and analyse the results. However, before discussing the experiments, it is first necessary to address the theoretical issues mentioned above related to locking and the integrity of the data. Thus, in Chapter 2 a number of strategies for processing mixed workloads are examined.

Two of those strategies, referred to here as unified data and split data, constituted the basis for the performance experiments. The experimental methodology employed is described in Chapter 3, while the results and analysis are presented in Chapter 4.

In brief, test data was generated and loaded into a number of tables in a database system using a commercial DBMS. Transaction and query application programs were then written and run against the data both standalone and concurrently so as to measure the impact of concurrent execution on performance. The experiments are thus a 'simulation' in the sense that they attempt to emulate real world workloads and data structures. But whereas other simulations randomly generate transaction and query response times based on some assumed processing model, the experiments conducted in

this thesis were ‘real’ in that an actual DBMS processed actual queries and transactions against actual data.

In our view the experimental approach is the only satisfactory way to settle the performance issue posed by mixed workloads, short of actually implementing mixed workloads in real systems. That is, it is not feasible to predict with any confidence how mixed workloads will perform based on analytical models alone. Database hardware and software systems have become very complex, with more features and more intelligent optimisers being developed every year. In order to be able to say with confidence how a particular database system will respond to a particular workload the only viable approach is to try it.

An implication of course is that the conclusions of this thesis are to some degree specific to the DBMS, application and platform tested. To what degree that is true can only be decided by future research and real-life experience. Nonetheless, we believe that the results reported from our experiments provide a useful indication of how mixed workloads can be expected to perform, and a reference point against which future results can be compared. At the very least, a number of key issues that will determine mixed workload performance in any environment are highlighted.

Most theses review the related work in the field in an early chapter before going on to discuss their new research. But we have been unable to find any other published experimental data in the area of mixed workload performance. Instead, the articles in the literature that address the issue of mixed workloads are theoretical in nature: they propose

new database capabilities such as record versioning and enhanced locking techniques that are intended to support concurrent transaction and query execution. We do feel these proposals are sufficiently germane to the thesis topic that they merit discussion, and so they are reviewed in *Chapter 5, Related Work*. But the main focus of the thesis is the performance experiments conducted on the basis of existing technology. And so we have chosen to explain the experiments and report on their results before discussing the new and as yet largely unimplemented proposals that have been made in the literature.

Chapter 6 summarises the results and conclusions of our research.

1.7 Contribution

The key contribution of the thesis is that it reports and analyses the results of a set of mixed workload performance tests based on generated data and a commercial DBMS. Published experimental performance studies are all too rare. Of course, vendors of database management systems are keenly interested in the mixed-workload issue and conduct their own tests in-house. But they jealously guard the results of such studies so that the user community is mostly left in the dark until they buy the product.

The Transaction Processing Council, an industry consortium, has defined a number of standard database performance benchmarks known as TPC-B, TPC-C and TPC-D. These benchmarks usually cost over half a million dollars to conduct and are still the subject of intense debate as to their validity. But in any case none of them address the

issue of mixed workloads. TPC-B and TPC-C are for OLTP-style transactions only, while TPC-D is limited to DSS queries.

Again, it is difficult to say how representative the performance characteristics of the particular platform, DBMS and applications used in this thesis will be of mixed workloads in general. Future research will be needed against other configurations to determine patterns of mixed workload behaviour. We believe the results reported here constitute a useful starting point.

Other contributions of the thesis include the following:

- a discussion of mixed workload strategies.
- an assessment of the relative performance of the split and unified data strategies.
- an analysis of the efficiency of mixed workloads compared to standalone workloads in terms of system resource utilisation.
- a review of other mixed workload techniques proposed in the literature.

1.8 Issues not Addressed and Final Remarks

In order to clarify the subject of this thesis it is useful to observe that there are some closely related issues that it does not address. First, the question of ‘long-lived’ transactions (LLTs) has received considerable attention in the literature, specifically in the context of software management. If a developer ‘checks out’ a software module from a software database in order to make modifications to it over a period of days, then in some

sense that module is 'locked' for the entire period it is checked out. This is a 'long-lived transaction' which poses specific challenges for database management systems.

DSS queries are usually long-lived compared to OLTP transactions. But they are long-lived because it takes the DBMS a long time to process them, not because some data has been 'checked out' of the database by a user. DSS queries are active as opposed to passive long transactions. They impose a major workload on the DBMS while checking out a software module plainly does not. So while this thesis is looking at the execution of long-lived transactions, that is queries, concurrently with OLTP applications, it is not considering the issues related to passive LLTs.

A second issue that is not addressed in this thesis is long-lived updates, where 'updates' can mean insertions, deletions or updates of existing data. Bulk updates are a common occurrence in information systems: monthly interest calculations for all of a bank's accounts; deleting all the transactions more than a year old; modifying all the organisation codes to implement a new coding structure; and so on. Such bulk updates again represent a very different type of workload compared to traditional OLTP systems, and usually cannot be processed concurrently with OLTP transactions since the data affected by the bulk update must be locked for the duration of the update.

Bulk updates are a thorny problem for database systems, but this thesis does not address it. Instead, the focus here is on concurrent queries and transactions, where it is understood that the query does not modify the data it is querying in any way. The query

may create intermediate tables in the process of its execution, which tables are normally dropped when the query completes, but the base data itself is unaffected.

We would like to conclude this introductory chapter with two remarks. First, the reader will notice that the name of the actual DBMS used is not revealed in the thesis. There are two reasons for this. We have no authorisation from the DBMS vendor (who was not a participant in the research) to publish the company name, and the vendor could potentially object to some of the findings. Second, the author is an employee of the vendor of a competing database management system, and therefore in a potential conflict of interest situation. We wish to stress, however, that the aim of the research was not to evaluate the DBMS but rather mixed workloads, and that every effort was made to ensure that the DBMS was performing optimally.

Second, whatever the theory of database management systems may be the fact is that to date the only access that most users have had to their data most of the time has been via pre-defined OLTP-style transactions. On the whole, the user community has exhibited considerable trepidation in attempting to actually use a database system to support multiple concurrent applications, doubtless because of painful experiences. Progress has been slow and incremental. The basic question of this thesis is: have DBMSs got to the point where the envelope can be pushed a bit further, and mixed workloads can be seriously considered?

Chapter 2

MIXED WORKLOAD STRATEGIES

2.1 Business Issues

In this chapter we make the assumption that some organisation wishes to execute transactions and queries against the same database at the same time. In other words, it has decided to attempt a mixed OLTP and DSS workload. Leaving aside performance issues for the time being, what problems do mixed workloads present, and what strategies are available to the organisation for executing them?

The most important issue is data integrity. Normally, when a DBMS executes a query it places read locks on the data accessed by the query. Provided that the DBMS follows a proper locking discipline such as two-phase locking, the person who submits the query can be confident that the results will not be corrupted by any concurrently executing transactions. Either all of the changes or none of the changes effected by any given transaction will be visible to the query, and since a transaction always takes the database from one consistent state to another, so too the data seen by the query will be consistent.

While such locking or concurrency control schemes work well for a mixture of OLTP transactions and very short queries, they become problematic when DSS queries are introduced, because DSS queries can run for minutes or hours and lock entire tables. During that period of time no OLTP transaction can process inserts, updates or deletes

against the locked data. Obviously, this is unacceptable from an OLTP user point of view.

So how are mixed workloads to be executed at all?

Another key issue that mixed workload strategies must take into consideration is the nature of the business and its data. Some businesses or applications are naturally 'periodic'. They operate on a strict time cycle, say by month. An accounting system is a good example in which there are strict rules regarding monthly cut-offs, prior-period adjustments, and so on. In this case, there is normally a business rule that no data older than a specified date may be modified; all modifications must be processed against the current data. Moreover, it is critical to the business that the prior period data remain static so that it can be properly analysed and compared to the data for other periods.

For other businesses or applications the data is relatively dynamic. It is always possible to go back and modify data for an earlier time period. It is not essential to be able to create snapshots that are known to be unchanging and against which many queries can be run with consistent results. Certainly, if the only data of interest is the current data then it will constantly be subject to change.

In response to these issues, there are three broad strategies available to an organisations that wishes to execute concurrent OLTP and DSS applications in a single database system. The first is to relax the locking discipline and allow queries to read data that is subject to change and hence possibly inconsistent. The second is to partition the data on the basis of time periods. Transactions are restricted to current data so that queries can be confident that historic data is static. The third strategy is to implement an

enhanced concurrency control scheme at either the application or DBMS level, such as data versioning.

Whatever mixed workload strategy is adopted, it will need to meet the following requirements:

- maintain OLTP transaction integrity; i.e., support the ACID properties.
- not allow long queries to block transactions from processing.
- ensure that transaction response times are acceptably short and predictable.
- enable DSS queries, including lengthy and complex queries, to receive some reasonable level of service.
- fit with the business model, whether it can be characterised as periodic with static historical data or relatively dynamic.

2.2 Query Management

Before examining the possible mixed workload strategies more closely, we first turn to an issue that is common to them all: how to manage the query workload? DSS applications, again, are characterised by a relatively small number of queries that are often ad hoc and one-time in nature and can range from short and simple to long and complex. Hence, it is not possible to predict what queries will be submitted, and the DSS workload can be ‘lumpy’ since there are relatively few queries of widely varying size. This contrasts

to OLTP workloads where many, small, pre-defined and well-known transactions behave in a predictable, stochastic fashion.

In early data warehouses, it was customary for organisations to allow their knowledge workers to simply submit queries at will, and the DBMS would immediately process them concurrently with any other outstanding queries, just as an OLTP system attempts to respond immediately to any submitted transaction. More recently, data warehouse users have begun insisting on tools that will enable them to manage the query workload so that, for example, very demanding queries are deferred from peak hours to overnight execution, and to enable queries to be assigned priorities. The intention is to ensure that the DBMS does not become clogged with too many concurrent queries, and that there is some predictability in response time for at least shorter, high priority queries.

Query management systems are usually based upon queues. When a user submits a query it is not sent directly to the DBMS for processing, but rather to the query management system which places it in the appropriate queue, perhaps based on a preliminary estimate of the required execution time. The query manager releases queries from the queues to the DBMS in accordance with administrator-defined policies, and normally routes the output as requested by the user at query submission time.

If users find they need query management systems in the query-only environment of data warehouses, then they will be even more in need of such tools when executing queries concurrently with OLTP applications. It simply will not be acceptable to have a 'lumpy' query workload with its consequent variable impact on transaction performance,

at least during peak business hours. Nor will users be willing to risk having the DBMS become overloaded with queries to the point where transactions receive few system resources. That is, a query management system will be a pre-requisite for any mixed workload environment.

Needless to say, a queue-based query management system implies that the DSS ideal is compromised to some degree. We are somewhere in between the frustrations of traditional batch reporting systems and instant answers to the most complex, ad hoc queries. Nonetheless, an effective query management system will be viewed by most DSS users as a major advance over batch reporting. Particularly if batch reporting has become impossible because there is no batch window left.

Even if some batch reporting window had still been available, a query management system will permit at least shorter queries to be executed during the day, in a nearly interactive fashion. Depending upon system capacity, it will normally be possible to execute some longer queries at a lower priority level during business hours. Long queries will no longer be forced to fit inside the batch window since they can continue to execute concurrently with transactions. Finally, allowing concurrent execution of queries and transactions gives organisations the option of configuring their hardware and software platforms to provide the desired level of transaction and query service during business hours.

For purposes of the research conducted in this thesis, it was assumed that all queries would be submitted through either one or three queues, referred to here as

'streams'. That is, experimental mixed workloads consisted of either transactions plus one query stream (queue) or transactions plus three query streams. Each query stream consisted of one or more queries which were executed serially with no intervening sleep time since, of course, it was assumed that the queries were queued. No priorities were attached to the queues, so that any priority differentiation was determined by the DBMS or operating system and not the user.

By means of these query queues or streams, we have assumed a simple query management discipline. Queries are not submitted directly to the system in an unpredictable fashion, but rather through queues which thus control the number of concurrently executing queries in the DBMS. Hence the DBMS query workload becomes manageable and predictable.

In our view, some such query management system will be required in any mixed workload environment and is a reasonable imposition on DSS users which enables them to benefit from concurrent query and transaction processing.

2.3 Unacceptable Solutions

Two potential solutions to the mixed workload problem can be ruled out immediately. First, the normal locking mechanism implemented by all commercial, relational database management systems cannot be used. If it were, lengthy queries would lock large volumes of data and all transaction processing against such data (inserts, updates and deletes) would be blocked until the queries released their locks.

Second, it is not acceptable to simply turn off locking altogether unless some alternative means of maintaining transaction integrity is substituted for it. OLTP transactions that are modifying the data normally operate under an appropriate locking discipline in order to ensure transaction integrity. In particular, locking enforces the isolation property: transactions can execute as if there were no other concurrent transactions in the system. Thus, with a locking discipline there is no potential for transactions to interfere with each other and so possibly corrupt the data.

We can conclude that in a DBMS that uses locking, queries cannot be allowed to lock data needed by transactions, and that transactions must use locks. The only possible alternative would be to implement some concurrency control scheme not based upon locks that guaranteed transaction integrity while supporting concurrent query execution.

2.4 The Read Uncommitted Isolation Level

One possible mixed workload strategy is to execute the DSS queries without locks, while the OLTP transactions run under the control of the usual locking system. In this way the two workloads can be executed concurrently without having the queries lock large volumes of data, thus blocking transactions against the same data for the duration of query execution.

In this scheme, transaction integrity is maintained, but the validity of results returned by the DSS queries is compromised. While the queries are executing the data they read could be modified.

Suppose, for example, that a query was executed to find the total payments made to some doctor in the current month to date. While the query was running a transaction was executed to insert some new payments for the doctor and delete some old payments, thus correcting a bookkeeping error. If the query mistakenly read both the newly inserted payments and the old, incorrect payments before they were deleted then the answer the query returned would have been incorrect as of any point in time. If the query had executed with locking, then it would either have reported the total payments to the doctor before the transaction was processed at all, or after it was processed in its entirety, either of which would have represented a 'consistent' database state.

However, in many organisations it is acceptable for the answers to DSS queries to be imprecise. Most DSS queries read many rows of data, and normally only a small percentage of them will change because of transaction activity during query execution. Thus, the answer returned by the query will be at least approximately correct in aggregate, even if it does not represent a completely consistent database state.

It is also debatable whether there is a 'correct' answer for some DSS queries. Again, suppose the query was to find total payments made month to date, only in this case to all doctors. While the query executes new payments are inserted, some of which the query reads and some of which are missed. What is the 'correct' answer in this case? Typically, the 'correct' answer is considered to be consistent data at the time the query began executing. But if the intent is to retrieve the total payments made month to date, and new payments are inserted as the query executes, then presumably those new

payments should be included in the total. In other words, the correct answer is a consistent database state as of the time the query completes execution.

In real life, the user who submits the query will likely just want a reasonably good estimate of the month to date payments, and will not be concerned whether it includes payments inserted while the query executes. From his point of view, there is no precise, 'correct' answer, just a reasonably good one. In fact, he would probably be pleased if the query picked up some of the new payments.

There are other application environments, of course, in which it is essential that the data remain unchanged while the query runs. For example, reports against a prior accounting period must all execute against the same data if they are to be consistent and hence reconcilable. No transactions can modify the data while the report queries are executing. In such an environment either the query must lock the data or some other mechanism must guarantee that the data is not being updated.

There will also be short queries which are more transactional in nature that must run under the control of the locking system to guarantee valid results. For example, if a user wishes to know her bank account balance the appropriate locks will be placed as the data is read. Such 'transactional' queries, however, are operational in nature and do not fall into the decision support category.

As for DSS queries, there are many business situations in which approximate answers will be perfectly acceptable. Indeed, in the author's seven years of experience with DSS systems most users were prepared to tolerate approximate answers.

In technical terms, executing queries without locks is known as the 'read uncommitted' isolation level. This is the lowest level of transaction isolation which amounts to no isolation: the transaction is not protected at all from the actions of other concurrent transactions. If the query happens to read a block which has been modified by some other transaction then it will use that data in arriving at its answer even if the transaction had not yet committed and could even, potentially, roll back its changes. The transaction may have locked the block, but since the query is running in read uncommitted mode it will read the block anyway.

In summary, one possible mixed workload strategy is to execute queries without locks, at the read uncommitted isolation level, and to execute transactions with locks. In this way, transactions and queries can be executed concurrently. The price that is paid is that answers to queries will be approximate to the extent that transactions modify the data being queried while the queries run. Whether or not this price is acceptable will need to be decided on a case by case basis, but it will in fact be acceptable for many DSS applications.

This particular strategy is one of the two that were tested in the performance experiments conducted for this thesis. It is referred to here as the 'Unified Data' strategy. As discussed in the next chapter on methodology, a single set of tables was defined containing all the data used by both transactions and queries. Then performance tests were carried out in which the transactions placed locks while the queries ran without locks.

2.5 Partitioned Data

Another mixed workload strategy is to partition the data against which queries and transactions execute. In most if not all applications partitioning will be based on time: the transactions will run against the 'current' data partition while the queries target the 'historical' data partition.

A second assumption needs to be made for the partitioned data strategy to be viable: the historical data is not allowed to change. That is, there is a cut off point in time after which no further modifications may be made to prior period data.

The benefit of a partitioned data approach is that queries against the historical partition read stable data: it is not changing while the queries execute. Consequently, users can be confident that the answers returned by such queries are 'correct', at least for the period of time covered by the historical data. What may be more important to some businesses is that many queries can be run against the historical data and their answers will be consistent since the data is static.

If there were no hard and fast rule against modifications to historical data then the benefit of partitioned data is either lost altogether or becomes one of degree. Perhaps relatively few transactions would execute against historical data, but even so the query responses could not be relied upon to be completely correct as of some point in time, nor consistent with respect to other queries against the same data.

A partitioned data strategy will fit well with certain business environments and applications. Financial systems in particular tend to be periodic in nature and subject to

rules that disallow modifications to historical data, precisely so that prior periods can be analysed and reported upon without the risk of inconsistent results. All transactions would then be restricted to the current or most recent periods. Any queries against historical data would produce valid results, even if the queries were executed concurrently with transactions.

It can be observed immediately that the queries against unchanging, historical data can be run without locks. Any number of queries can be run concurrently against the historical partition without compromising the integrity of their answers since queries are 'read only' in nature, at least with respect to the base data. (It is assumed that any temporary tables created in the course of query execution are invisible to other queries.) Since transactions that modify historical data are forbidden there is no need to isolate the queries from such transactions and so the queries can run without locks.

In addition to ensuring that the answers to DSS queries are correct and consistent, the partitioned data strategy offers some potential performance benefits versus the unified data strategy discussed earlier in which queries run at the read uncommitted isolation level. Both the queries and the transactions will execute against a subset of the data, either the historical or the current portion. Assuming that the query only needed the historical data, partitioning saves it from unnecessary reads. Similarly, transaction processing will be more efficient against smaller tables since, for example, the number of index levels will be smaller and the cache hit ratio will likely improve.

The partitioned data strategy can be implemented in two ways: physically or logically. In a physical implementation, the base tables themselves are partitioned into separate tables of current and historical data. The queries then use the historical tables while transactions run against the current tables.

A logical implementation of partitioning would be based upon views. That is, there would be one set of unified base tables against which historical and current views of the data would be defined. Then again the queries would access the historical views while transactions would be restricted to the current views (assuming that the DBMS supports updateable views, which is still problematic).

We first note that if partitioning is based upon views then the queries *must* run without locks since otherwise they risk locking the entire base table(s). DSS queries often scan entire tables, and so even if a view restricted the query to, say, the 80% of the data that was historical, the DBMS would lock the entire table rather just the historical data. As a result, all transactions against the table would be blocked while the query executed, if the query were allowed to place locks.

A logical implementation of partitioning based on views would ensure that concurrently executing transactions and queries did not interfere with each other in the sense that transactions could not modify data needed by the queries. But it does not offer the potential performance benefits of physical partitioning. Queries against the historical views will still need to scan the entire, unified base tables in order to find the historical

data. Transactions against the current views will in fact be running against tables that contain all the data.

Since unified tables would underlie logical partitions or views, the performance characteristics of mixed workloads against a logically partitioned database can be expected to be very similar to those of a unified database. There is little likelihood that performance experiments based on logical partitioning would yield results that differed significantly from the unified strategy. Hence the logical partitioning approach was not in fact tested. However, the performance of the split data strategy in which the base tables were physically partitioned was thoroughly tested and compared to the unified data results.

The physical partitioning of data has potential benefits in the areas of performance and the integrity of responses to historical queries. But it is not without its drawbacks as a mixed workload strategy. First, partitioning the data into separate tables based on time periods creates complexity and administrative overhead. Application developers need to be aware of the split and target the correct tables. Data base administrators will need to manage more tables, and to develop mechanisms for periodically migrating current data into historical tables. Such periodic data movements are typically batch processes that require the database to be inactive, which is a major challenge when batch windows are unavailable.

The other difficulty with the partitioned data strategy is that it is difficult to achieve a completely clean split of all of the organisation's data into current and historical time periods. While transactional data is usually temporal, most organisations maintain

many, typically hundreds, of tables of 'indicative' data which will usually not have associated time periods. For example, a list of valid medical service codes in a health insurance application is normally independent of time in that they are neither strictly historical nor current in nature. The data is valid in both time periods.

Queries and transactions will both need to access these other tables which cannot be classified or split into current and historical partitions. So now we are back to the original difficulties posed by mixed workloads. If the DSS queries lock such tables then transactions will be blocked which is unacceptable. If the queries run without locks on these non-partitioned tables then there is the risk that answers will be neither completely correct nor consistent, even if the indicative tables are less frequently updated than the transaction data tables.

A possible strategy would be to partition the tables whose data can be divided into current and historical periods, and then to replicate all of the other tables. Half of the replicated tables would be associated with the historical data and used by the queries, while the other half would be used and potentially updated by transactions. On a periodic basis, say monthly, the updated indicative tables could be migrated to the historical partition. The migration of the transactional and indicative data from the current to the historical partition would need to be synchronised.

The main problem with this strategy is that we are well down the road towards building a complete data warehouse. That is, for most organisations a large majority of tables would need to be replicated and relatively few could be split 'horizontally' based on

a date criterion. But the issue we wish to address in this thesis is: given that we do not want to run a data warehouse but rather use a single database for both transactions and queries, what is the best mixed workload strategy to adopt and what are its performance impacts? When the majority of tables have been replicated, one is no longer in a true mixed workload environment.

Replicating most of the tables has obvious costs in terms of hardware requirements, migration processing, and administrative overhead. These are costs that mixed workloads are intended to avoid. So the 'partition/replication' strategy was not tested in the performance experiments. Instead, we tested the physical partitioning strategy in which the transaction data was partitioned, but both queries and transactions accessed the same indicative tables.

It must be observed that this partitioned (or 'split') data strategy used in the performance tests was not 'pure' because both transactions and queries accessed certain tables such as the beneficiary table. The queries ran without locks so as not to block transactions against common tables. Consequently, even though the queries ran against static transaction data, the integrity and consistency of their results could not be guaranteed. Compared to the unified data strategy, the query results against the partially partitioned data would be relatively stable, but not completely so. Organisations considering a partitioned data approach will have to decide on a case by case basis whether the degree of improvement in query result integrity and consistency, as well as

any performance benefits, warrant the costs of dividing the transaction data into separate tables by time period.

2.6 Enhanced Concurrency Control Schemes

The third broad strategy for processing mixed query and transaction workloads is to use an enhanced concurrency control scheme as compared to traditional database locking policies applied to a single copy of the data. A number of possibilities are available here, including data versioning and enhanced serialisability techniques.

In general, enhanced concurrency control techniques have not been implemented in commercial database management systems. None of them were available in the DBMS used for our experiments and so we were unable to test their performance characteristics. However, enhanced concurrency control schemes have received considerable attention in the literature, and so are the main subject of *Chapter 5, Related Work*.

Since they are fully described in Chapter 5, we will only provide a very brief description of enhanced concurrency control schemes here. The most promising, in our view, is transient versioning in which the DBMS automatically maintains multiple copies of rows so that queries can access row versions that existed at some previous point in time while transactions continue to modify the most recent versions of the same rows. Old versions are merely transient because they are automatically deleted by the DBMS when there is no query still running that might need them.

Some other schemes fall into the category of enhanced serialisability techniques. They are based on the observation that a short transaction may be able to serialise before or after a long query as long as either all of the data the transaction modifies or none of it has already been read by the query. Transactions and queries can thus be allowed to execute concurrently against the same data without corrupting each other as long as the DBMS determines that they can indeed be serialised.

As for the performance of these sophisticated concurrency control techniques, it is clear that none of them will be faster than the unified data strategy in which queries run without locks concurrently with transactions. Consider the transient versioning scheme for example. Maintaining multiple versions of the same row plainly implies overhead for the DBMS. When a query executes it must be provided with the correct version of each row. When a transaction creates a new row version the structures that track the different versions of the same row must be updated.

Consequently, we claim that the performance results from the mixed workload experiments using the unified data strategy represent a 'best case' outcome for any of the enhanced concurrency control techniques. Our results effectively constitute a set of baseline performance numbers against which the cost of enhanced concurrency control techniques can be compared when they are eventually implemented.

There is one enhanced concurrency control scheme that is not discussed in the literature but which should nonetheless be mentioned for completeness. In principle, it is possible to implement permanent versioning at the application level. Transaction logic

could be modified to permanently maintain every version of every row ever created along with appropriate date and time stamps to indicate when each version was valid. Query logic could then make use of the date and time stamps to access consistent row versions at some previous point in time while transactions continued to update the latest versions of the same data.

While application-maintained versioning may be a theoretical possibility, it plainly is of little practical interest. Asking transactions to maintain appropriate versions of every row would be onerous and error prone. Similarly, query logic would become unwieldy if all row access needed to be qualified with date and time constraints.

Although application-maintained permanent versioning is impractical, DBMS-maintained permanent versioning has received serious attention under the heading of temporal databases. Please see *Chapter 5. Related Work* for a discussion of temporal databases as well as the other enhanced concurrency control schemes.

2.7 Strategies Tested

Of the various mixed workload strategies discussed above there are two that are most likely to be used by organisations in the near future. One is maintaining unified tables and executing DSS queries at the read uncommitted isolation level, which means without locks. The second is physical partitioning of at least the large, transaction data tables into current and historical sub-tables, with the transactions being restricted to the

current data. The names given to these two strategies in the balance of the thesis are 'unified data' and 'split data'.

Most of the performance tests conducted for this thesis were thus executed twice: once using the unified data strategy and a second time against partitioned or split data. The intention was to measure the impact of mixed workloads on transaction and query performance using both strategies, and so to be able to assess their relative merits.

None of the other possible strategies were subjected to performance testing. As argued above, logical partitioning based upon views would not likely perform differently from the unified data strategy. The DBMS used did not support any of the versioning or enhanced serialisability techniques discussed in the literature, and so none of them were tested either. Again, however, none of the advanced techniques will perform better than the unified data strategy since they all imply additional DBMS overhead.

Chapter 3

METHODOLOGY

3.1 Overview

The purpose of the experiments conducted for this research was to evaluate the performance of two mixed workload strategies: first, the unified data strategy in which all of the data was in one set of tables and the queries ran without locks; and, second, the split data strategy in which the large tables were physically partitioned based on a date criterion and the transactions were restricted to the current data.

The experiments were executed on a commercial, relational DBMS that is in widespread use. Two databases were created, one with the unified tables and a second with the split tables. A database model was developed based on a fictitious health insurance agency, and test data was generated to populate the tables. Then a number of OLTP transactions and DSS queries were developed using SQL and embedded in C-language programs.

An experiment consisted of invoking a driver program that would in turn fork processes to run selected transactions and queries a specified number of times. The experiments tracked a variety of performance measures such as response time. Data regarding system utilisation was also logged using the System Activity Reporting (sar) utility.

In all, over 800 tests were executed. The general strategy was to run a given workload standalone, and then rerun it in a mixed environment and measure the change in the workload's performance. For example, how much does the mean response time for a given set of transactions increase in, say, the unified environment when the transactions are run concurrently with a particular query workload?

The remainder of this chapter describes the experimental process in more detail.

3.2 Test Databases and Hardware

As mentioned above two independent databases were created for the performance experiments. In one the tables were unified. In the second database the two large transaction tables, `Payment_Header` and `Payment_Detail`, were split in two based on a date field in the data. It should be noted that in the split environment the smaller tables were neither replicated nor split, but were in fact exactly the same as the corresponding tables in the unified environment.

While two databases were created at most one was in use at any point in time. That is, tests against the split and unified environments were never run concurrently.

The hardware platform that was used for the experiments was a Sun Ultra1 with a 143 MHz UltraSparc processor, 32 MB of memory, and two SCSI disk drives. The operating system was Solaris 2.5.

A significant effort was made to set up the database environment so as to optimise performance given the DBMS and platform used. In particular, one disk was dedicated

exclusively to the tables and indexes of the two databases. Everything else went to the second disk, including the database log and journal files, DBMS software, test programs, test results, *sar* data, and other Unix file systems. In addition, the log file was defined as a raw I/O device managed by the DBMS so as to speed up log writes.

As will be seen from the test results, the disk that contained the data and indexes (referred to as 'disk 1' in *Chapter 4: Results and Analysis*) proved to be the main performance bottleneck, and in hindsight performance could perhaps have been further improved by splitting some of the data between the two disks as well. Nonetheless, the manuals for the DBMS employed stated that at least using a raw log file on a separate disk from the data disk would yield performance benefits.

3.3 Test Data

One of the major challenges in conducting a database performance study is coming up with a good test database design and accompanying data. Creating a test database and data that are reasonably representative of real world applications and data demographics requires time and thought. Otherwise, the test applications and results are too restrictive in nature and lack plausibility.

For purposes of the research done for this thesis a database model was developed based on a health insurance organisation, which could be public or private, that pays medical practitioners, mainly doctors, for services rendered to insured patients, known as 'beneficiaries'. Please see the database model depicted in Figure 1. In the database model

diagram each box or diamond represents, respectively, an entity or relationship which was implemented as a table in the test database. Above the line in each shape is the table name while the primary key attributes are shown below the line. The table definitions are in Appendix C, while the number of rows per table are shown in Table 1.

The health insurance agency receives requests from health professionals for payment for services provided to beneficiaries. A single payment request will include a header with the identity of the professional and the beneficiary, and one to four detail lines with each line representing a service provided to the beneficiary by the professional. The bulk of the transaction data, including amounts eventually paid if any, is found in these two payment data tables. As seen in Table 1, the `Payment_Header` and `Payment_Detail` tables contained approximately 250,000 and 500,000 rows respectively in the unified data database.

The other tables contained supporting information needed to determine whether a payment should be made and for what amount, as well as to enable analysis of the payments by various criteria such as the region in which the beneficiary lives. For example, the `Norm` table represents a relationship between professional categories and medical service codes that specifies which professional categories are permitted to bill for which service codes. The `Region_Postal_Code` table maps the first three characters of a postal code to a geographic region, making it possible to classify medical establishments (hospitals), beneficiaries and professionals by region.

Figure 1: Database model of a health insurance agency

For the split data tests, two additional tables were defined: Current_Detail and Current_Header. The Payment_Detail and Payment_Header tables were used for the historical data. None of the other tables changed. In fact, the definitions of the Current_Header and Detail tables exactly mirrored those of the Payment_Header and Detail tables.

Payment data was divided between current and historical periods in the split database based upon the payment date attribute in the Payment_Detail table. If the payment date was null, meaning that no payment had yet been made for the service, or if it was in the current month then the payment detail line and its associated header were placed in the current tables when the test data was loaded. Payments for older dates were placed in the historical tables. In fact, the 'historical' data covered a 12 month period, while the current data was for the first 15 days of the current (13th) month, plus rows with null dates. As a result, the Payment Header and Detail tables in the split environment had about 94% of the data volume of the same tables in the unified environment. The other 6% of the data was in the current tables.

Since up to four detail lines could be associated with a payment header, and since payment dates could vary for the different detail lines belonging to one header, care had to be taken in the test data generation process to ensure that the different payment dates on a given header's detail lines did not cross the current/historical date boundary. Otherwise the payment header data would either have to have been replicated or shared by transactions and queries, thus defeating the intent of the split data approach. In our view,

Table Name	Row Length (bytes)	Number of Rows
Payment_Header (unified)	57	250,822
Current_Header (split)	57	14,927
Payment_Header (split)	57	235,895
Payment_Detail (unified)	61	501,642
Current_Detail (split)	61	29,760
Payment_Detail (split)	61	471,882
Beneficiary	178	30,000
Category_of_Prof	31	1100
Diagnosis	32	250
Establishment	142	150
Norm	33	1500
Professional	126	1000
Prof_Admissibility	55	1100
Prof_Address	102	2000
Prof_Category	31	20
Region	31	8
Region_Postal_Code	31	160
Service	43	500
Service_Type	32	50

Table 1: Row lengths and number of rows in the test tables

this constraint on the legitimate payment dates within a single payment is typical of business rules in many organisations.

Once the data was loaded and indexes were created both the unified and split databases occupied about 126 MB of disk space. Of this, about 93.5% or 118 MB were

occupied by the payment tables in both the unified and split databases. The only other table of any size was the beneficiary table which took about 7.3 MB of space. This type of space distribution with a small number of large transaction tables and a large number of small indicative tables is typical of many business environments.

Considerable attention was paid to the demographics of the test data: the range of values of each attribute, the number of distinct values, the distribution of the number of rows per value, nulls, and the foreign key to primary key relationships. As an example, the following rules were used in the program that generated test data for the beneficiary table:

- the beneficiary_id ranged from 500,001 to 530,000 for the 30,000 rows, since it was the unique primary key.
- the birthdate distribution was as follows: 3%: 1900-9, 7%: 1910-19, 10%: 1920-29, 8%: 1930-39, 8%: 1940-49, 15%: 1950-59, 13%: 1960-69, 13%: 1970-79, 14%: 1980-89, 9%: 1990-96.
- civil_status: 30%: S(ingle), 40%: M(arried), 25%: C(ommon law), 5%: W(idow/er).
- admissibility_code: 'A' for admissible (for payment of services received) unless the beneficiary_id ended in '888', in which case it was 'I' for inadmissible.
- admissibility_start: 19930601.
- admissibility_end: null, except for the '888' beneficiaries: 19970510.
- sex: 52%: F, 48%: M.
- family_name: 'Beneficiary 500,001', etc.

- other_names: 'Names 500,001', etc.
- address_1 and _2: 'Beneficiary 500,001 address line 1 (or 2)'
- address_3: as in address_1 and _2 for 25% of the beneficiaries. Null for the other 75%.
- postal_code: First character: G. Second: half from 1 to 3, and the other half from 4 to 8. Third: random letter from A to T. Fourth: blank. Fifth and seventh: random digit from 0 to 9. Sixth: random letter from A to Z.
- social_security_ind: 90%: N, 10%: Y.

Beneficiary_id was also an attribute of the Payment_Header and Current_Header tables, thus implementing a foreign key/primary key relationship. The payment header data was actually generated by beneficiary. The number of headers per beneficiary was exponentially distributed with a mean of 8.333 rows per beneficiary, thus obtaining approximately 250,000 headers as desired.

Rules of this type specified the data demographics for all tables, and thus also governed the generation of test data whether manual, for the small tables, or by means of programs for the larger tables. In order to create meaningful data the rules were occasionally somewhat involved. For example, to choose a service code to use in a Payment_Detail row, the test data generation program first needed to determine the identity of the professional on the corresponding header. That was used to select one of the professional categories to which the professional belonged. Then, normally, one of the service codes that was valid for the professional category according to the Norm table was

chosen for the payment detail line. However, for a randomly chosen subset of detail records a service code was also chosen randomly, meaning that some payments would in fact violate the rules articulated by the Norm table. Hence queries could be written looking for such violations which, plainly, would involve a number of joins.

Because data demographics rules such as these were used in generating the test data, test databases were developed which, in our opinion, realistically emulated real-world environments, and were also rich enough to support the creation of many legitimate transactions and queries.

The large tables and the indicative tables that were subject to growth were implemented as btrees, while the smaller, static tables used an isam data structure. The DBMS manual recommended btrees for any tables that were expected to grow in order to avoid overflow chains. In any case, all of the tables could be accessed directly based on key (cluster index) values, whether unique or non-unique.

The Payment_Header and Current_Header tables used the control_no attribute as a unique key, essentially a system-generated transaction identifier. The Payment_Detail and Current_Detail tables were clustered by payment date and control_no, a non-unique key since there could be multiple detail lines for a single control_no. Since there could be at most four detail lines per header, adding the line number to the key would have been unlikely to save disk I/O even if only a single detail line needed to be retrieved.

By making payment date the major clustering index of the detail tables, the intention was to support efficient execution of queries against a specific date range,

particularly queries against the current month data in the unified database. As will be seen in the results discussion in Chapter 4, however, the optimiser did not make good use of the date key when date criteria were specified in the queries. It is possible, then, that given the particular DBMS used it would have improved performance if the Payment_Detail table had been clustered on control_no just like the Payment_Header table. Then the frequent joins between the two tables would have been faster. However, time did not permit redoing the experiments with a different Payment_Detail table structure, and in any event such a change is unlikely to have affected the relative performance of different mixed workload tests.

Only four secondary indices were defined in the unified database. The Payment_Header table had secondary indices on prof_id and beneficiary_id to support rapid retrieval of the payments for a particular professional or beneficiary. The Payment_Detail table had secondary indices on control_no and service_code. This supported efficient queries against particular service codes, as well as the rapid look-up of the payment details associated with the control_no of a particular header. In the split data environment there were corresponding indices for both the current and historical payment tables.

It would certainly be possible to define more indices to support faster queries, say on the establishment_no in the Payment_Header table, or on postal_code in the Beneficiary table. But while DSS applications tend to favour numerous indices so as to facilitate queries, OLTP databases tend to minimise the number of indices in order to

speed up transactions. The more indices there are the more work a transaction needs to do with every insert, update and delete, since corresponding changes must be made to the affected indices. In our opinion, given the database model and data demographics used, the decision to use just four indices in the unified database (eight in the split) was a reasonable compromise between the needs of OLTP and DSS applications. Such compromises will be necessary in any mixed workload environment.

All modern DBMSs have sophisticated optimisers which make heavy use of statistics describing the data demographics, and the DBMS used for our experiments was no exception. Statistics were therefore collected on all of the cluster index columns and all of the secondary index columns. As a result, the optimiser normally had statistics on columns used to specify selection and join criteria. Since the database was restored to its original state prior to each test run, the statistics were always precisely valid when a given test began executing. The statistics could not be and therefore were not updated dynamically as transactions executed.

3.4 Test Transactions, Queries and Programs

Based on the health insurance agency database model, a number of test OLTP transactions and DSS queries were defined for use in performance testing. Various combinations and volumes of queries and transactions, together with inter-transaction sleep time, constituted 'workloads' that the DBMS was asked to execute.

The most demanding transactions were payment inserts. Each payment insert transaction inserted a new header row plus one to four detail rows in the Payment_Header and Payment_Detail tables respectively. Two DBMS procedures were used to perform the transaction: one to insert a header and the other to insert a detail line. Both procedures took as arguments the relevant data to be inserted. The SQL used in the procedures as well as all of the other transaction and query SQL can be found in Appendix D.

Both procedures executed a number of integrity checks prior to inserting a row. The integrity checks could have been executed by the transaction application program instead, but that would have required many more interactions between the application and the DBMS. It is much more efficient to implement such validations in a procedure since a single call from the transaction application can then automatically process all of the necessary checks and perform the insert if the checks are successful.

The header insert procedure verified that the beneficiary and professional for whom the payment was being processed existed in the Beneficiary and Professional tables. A similar validation was performed for the establishment number if it was not null. The detail line insert procedure first verified that a corresponding header existed. Then it checked that both the professional and beneficiary were 'admissible' for payments according to the Prof_Admissibility and Beneficiary tables. Finally, it was ensured that the service code existed in the Service table. It should be noted that the data demographics

rules deliberately created some integrity violations so that a small number of transactions would be and were in fact rejected.

The transaction programs themselves were written in C with embedded SQL. An example of such a program, `t1.sc`, may be found in Appendix E. Another set of programs was written to generate input test data files for the payment inserts and other transactions. The transaction test data generation programs borrowed heavily from the logic used to generate the table data, and so remained faithful to the data demographics rules.

The C transaction programs read the input test data files and then made the necessary calls to the SQL procedures to perform the header and detail line inserts. Since the input data was occasionally null for some fields, the transaction programs needed to be able to make the appropriate procedure calls with both null arguments and legitimate attribute values. The transaction programs also measured the response time for each transaction executed and stored the time in an array so that summary performance data could be calculated.

A number of other relatively simple transactions were also defined and executed in all of the transaction workloads. For these additional transactions, the necessary SQL statements were embedded directly in C-language programs rather than in DBMS procedures. Three such transactions were used in the performance tests: payment updates, beneficiary inserts and beneficiary queries.

The payment update transaction was actually an update to a `Payment_Detail` row. A test data generation program read the original `Payment_Detail` table and picked out

control_no/line_no combinations for rows whose payment date was either in the current month or null, indicating an unpaid service. The test data generation program then generated transaction input records with the control_no, line_no, a new payment amount, and payment decision and reason codes.

The payment update transaction program read the file of input data and processed the updates against the Payment_Detail table. Prior to executing the update, it would first select the old amount from the original row.

The beneficiary insert transaction emulated the addition of new beneficiaries to the Beneficiary table. Like the other transactions, it was based on an input file of new beneficiary rows to be added to the database. The only verification performed by the beneficiary insert was to select some of the data in the new row back after it had been inserted.

Finally, the beneficiary query retrieved all of the payment header and detail rows in the database pertaining to a particular beneficiary, as identified by an input beneficiary id number. Beneficiary queries were relatively time consuming, even though they were treated as OLTP rather than DSS transactions. That is, they were executed with locks so that any concurrent transaction activity against the same beneficiary's data would have to be serialised. However, there was a secondary index on beneficiary_id in the Payment_Header table, and given a header record the corresponding detail lines could be found quickly using the secondary index in Payment_Detail on control_no.

The beneficiary query transaction, then, used two cursors. The first retrieved the header records for the beneficiary in an outer loop. The second selected the corresponding detail lines in an inner loop.

In the split data environment the beneficiary query transaction became even more complex since the intention, again, was to retrieve complete payment data for a given beneficiary. The SQL for both cursors used a UNION statement to retrieve the headers and detail lines from both the current and history payment tables. Unlike beneficiary queries, the payment insert and update transactions in the split environment were restricted to the current payment tables.

A payment delete transaction was also defined for performance testing purposes to round out the transaction types. However, as is described in Chapter 4, the DBMS would sometimes abort queries if the data being queried was deleted, and so the delete transaction was dropped from the test mix.

Nonetheless, we believe that the transaction types used reasonably emulated real-life workloads. Most business environments process a large number of inserts into the transaction tables, and those inserts are required to pass a number of integrity checks (payment inserts). Existing transaction data will be modified (payment updates). The indicative tables will also have some update activity against them (beneficiary inserts). Finally, some relatively short queries, as compared to DSS queries, will be processed with locks as part of the OLTP application (beneficiary queries).

In order to simulate DSS workloads, a variety of queries were also developed. Again, the SQL for the queries can be found in Appendix D. Some queries took several minutes to execute standalone, while others took advantage of index structures and specific selection criteria to execute in about 10 seconds standalone. When developing test queries, the execution plans produced by the optimiser were studied to ensure that they were reasonable and that a good mix of DBMS functions was being exercised: different types of join strategies, scans, indexed access, and so on.

Some of the queries used in the experiments were the following:

- Count the number of services for which each professional has been paid in the last three months. Report the name, category and id of the professional, and the total amount paid to him or her for the services as well as the service count.
- For a particular date, find the total costs by establishment within each region. (A 'short' query.)
- Find the 100 most costly beneficiaries.

In the split environment most of the queries ran exclusively against the historical payment data, as well as the indicative tables. They did not query the current payment tables. An exception, however, was the 'hot spot' stream. In that case, a number of queries were written that targeted the current payment tables exclusively (as well as the indicative tables). The intention was to investigate mixed workload performance when both queries and transactions focused on a relatively small subset of the overall database.

A set of hot spot queries that selected only current month payment data was also written for the unified data environment but, as reported in Chapter 4, was not successful in creating a bona fide hot spot.

The function of the C application programs was to emulate users submitting transactions and a query manager submitting queued DSS queries. Normally, transaction input data would be obtained from users by means of a user interface application running on client systems separate from the DBMS server platform. Once the necessary transaction input data had been assembled and submitted for execution by the user, it would be forwarded over the network to the DBMS server for processing. While it is possible that the query manager would run on the same platform as the DBMS, it too could be located on a separate server.

The goal of the experiments conducted for this research was to examine the performance of mixed workloads in a single database system. Since it can be assumed that the DBMS server platform will typically not also be responsible for the user interface programs, nor perhaps even for the query manager, it was important to minimise the load placed by the query and transaction programs on the test platform.

To this end the role of the transaction and query application programs was kept to a minimum. The transaction programs read a file of input data and submitted the transactions to the DBMS. When the transaction was complete the response time was recorded in a program array. After all the transactions for a workload were processed the application calculated summary statistics such as mean response time and the standard

deviation of response time. The fact that DBMS procedures were used to execute the most complex transactions, payment inserts, was another technique that helped to minimise the interactions with and processing done by the application programs.

Similarly, the only function of the query application programs was to submit a relatively small number of queries, fetch the responses, and record execution time. A sample query program, `qset3.sc`, is included in Appendix E. During query development the results of all queries were examined and validated. But during performance testing the query programs did little or nothing with the query result rows after they had been fetched. At most some simple control totals were output to ensure that the query had succeeded. All I/O done by both the transaction and query programs was to the second disk in the test system as opposed to the data disk.

As a result and as reported in Chapter 4, the load placed upon the server by the application programs was negligible. Virtually all of the processing done during performance testing was carried out by the DBMS in response to SQL requests, whether to process transactions or queries. The test platform thus behaved very much like a database server.

Two other C programs were instrumental in executing performance tests. One simply invoked the `sar` utility to capture system resource consumption while a test was running. The second was a driver program which forked a number of processes to execute the `sar` program, and the transaction and query application programs involved in a performance test. The driver read a simple control file that specified which application

programs to execute and how many transactions and queries they should process. The driver then waited until all the application programs had completed and read their various summary result files to consolidate them into a single result file for the overall test. An example of typical performance test output produced by the driver is shown in Figure 2 later in this chapter.

3.5 Workloads and Test Execution

Based on the individual transactions and queries described above, a number of query and transaction ‘workloads’ were defined. Standalone as opposed to mixed workloads ranged from a single, lengthy query to thousands of transactions of different types.

Three distinct transaction workloads were defined, referred to as heavy, medium and light. It is critical to observe right away that exactly the same transactions were executed in each of these three workloads. The only factor that varied was the average sleep time between transactions. Normally in this thesis we try to use the terms heavy, medium and light *workloads* rather than *transactions*, since it was the workloads and not the transactions that varied. Nonetheless, the reader needs to be aware that when we occasionally refer to, say, ‘light transactions’ what is really intended is the ‘light transaction workload’.

For each transaction workload, four client application programs were invoked concurrently by the driver: two executed payment insert transactions from separate input

files, while the other two executed payment updates, beneficiary inserts and beneficiary queries. Since each transaction client program logged on to the database and created its own session, this means that there were effectively four concurrent users in all of the transaction workloads. For each transaction workload, the sleep times between transactions was exponentially distributed. What varied between heavy, medium and light workloads was the mean of the exponential sleep time distribution.

In the case of the heavy transaction workload, both of the payment insert programs used a mean sleep time of 10 ms between inserts. For the two other transaction programs the mean sleep time between transactions was 20 ms. By giving payment inserts a shorter average sleep time the intention was to emulate the fact that they would be the most intensive transaction workload in a real application. For the medium transaction workload the mean sleep times were 200 ms for the payment inserts and 400 ms for the other transactions, while the corresponding figures for the light transaction workload were 400 ms and 800 ms. With these mean sleep time values, we found that the three transaction workloads, when run standalone, placed a significant and yet varied load on the test system as reported in the sar results in Chapter 4.

Of course, the actual time a human user would normally take to submit another transaction after receiving a response from a previous one would be much greater than even 800 ms. But in order to simulate the sleep time between transactions that is typical of actual users, far more 'client' application programs would need to have been executed in order to achieve significant arrival rates for the DBMS. Since we were constrained to

executing our performance tests on a single platform that acted as both client and server, and since the focus was on database server performance, it was important, again, to minimise the workload imposed on the test system by the client applications.

For this reason, exactly four client transaction programs were executed in each test that involved a transaction workload. Each of the four programs introduced exponentially distributed sleep times between submitted transactions with the mean sleep times given above so as to simulate the stochastic nature of transaction arrivals. But by keeping the sleep times short we were able to achieve significant transaction workloads with a small number of client programs, thus constraining the load imposed on the test platform by the client side of the application.

In a given performance test involving transactions, each of the client transaction programs was passed an argument that dictated how many 'repetitions' it was to execute. Typical values were, say, 1520 repetitions for each of the two payment insert programs, and 200 repetitions for the two other transaction programs. In the case of payment inserts, a 'repetition' consisted of exactly one payment insert as described above: adding a header and its associated detail lines to the appropriate tables by means of procedure calls. But for the other two transaction programs a repetition or 'cycle' consisted of four payment updates, two beneficiary queries and one beneficiary insert. Thus 200 repetitions equated to the execution of 1400 transactions. In all, then, $1520 * 2 + 1400 * 2 = 5840$ transactions were executed in the course of this particular performance test.

The minimum number of repetitions in any test was 304 for the payment insert programs and 40 for the other transaction programs, which equated to 1168 transactions in total. This small number of transactions was needed when testing the impact of the hot spot query stream on the light transaction workload in the split environment. The hot spot queries would complete so quickly that only a small number of light workload transactions could be completed before the queries.

The maximum number of repetitions was 6840 for the payment inserts and 900 for the other transactions which equated to 26,280 transactions in total. The large volume was needed to ensure that the transaction programs all ran longer than the queries when testing the impact of concurrent transactions on the 3-up query stream in the unified environment. In the detailed result tables in Appendix B the number of repetitions used in each test of transaction performance is shown. Again, typical values were 1140 and 150, or 1520 and 200 for the payment insert and other transaction programs respectively.

Only minor changes were needed to the transaction programs in order to execute them in the split data environment. Specifically, they needed to insert data into, update and query the current header and detail tables rather than the historical payment tables. In all, 20 different transaction client programs were needed, four for each of the five transaction workloads tested: heavy, medium and light workloads against unified tables, and medium and light workloads against split tables. Corresponding to each of the 20 transaction programs was a separate input file containing the necessary transaction data plus a sleep time to be used before the next transaction. That is, random sleep times were

input rather than generated during test execution, again to minimise client application resource consumption.

Three types of query workloads were used in the performance tests: ‘3-up’ streams, single query streams, and, in the split environment, a ‘hot spot’ query stream. 3-up query streams consisted of three concurrently executing sets of queries. Within a set of queries one to 90 queries were executed serially. Single query stream workloads consisted of the serial execution of one set of queries. In the split environment, a hot spot query stream was also developed which consisted of six serially executed queries against the current payment header and detail tables, as well as the indicative data.

Certainly, it would have been desirable to conduct performance tests with two concurrent query streams, four concurrent query streams, and so on. However, given the time available to carry out the experiments it was decided to restrict the query workloads to either a single query stream or three concurrent query streams. It would be feasible to execute more than three concurrent query streams in a multi-processor system, but it is unlikely that executing more than three streams would provide any additional performance benefits in a uniprocessor environment such as the one used in this research. As the sar data in Appendix A shows, even the single DSS query executed in the qset5a query stream used 50% of the CPU cycles and 59% of the data disk capacity while it was executing. Thus, by testing mixed workloads with either one or three query streams, the likely range of the number of concurrent query streams that would be executed on uniprocessor platforms was covered.

Leaving aside the hot spot stream, three ‘generic’ query streams were defined and used in both the 3-up and single stream workloads. In the unified environment they were known as qset1a, qset3 and qset5a, while the corresponding split data query streams were qspl1a, qspl3 and qspl5a. qset1a/qspl1a executed two long queries, while qset5a/qspl5a executed a single long query. qset3/qspl3 executed two short queries 45 times for a total of 90 queries, with each execution using a different input parameter and hence accessing different data.

In this way, at least two different types of queues that one would expect to be implemented in a query management system were emulated: queues for long queries and queues for short queries. However, no attempt was made to give the shorter queries greater priority than the long queries. Since the scenario we were trying to emulate was a queued query management system there was no sleep time between the queries submitted in a given stream.

The queries used in the three generic, as opposed to the hot spot, query streams in the unified and split environments were very nearly identical. In both environments they retrieved data from the Payment_Header and Payment_Detail tables, even though in the split databases these tables were slightly smaller. The only changes made to the queries for the split environment were to modify the date selection criteria as required in order to ensure that approximately the same amount of data would be retrieved by corresponding queries. Thus, for example, in the unified environment (qset1a) one query retrieved payment data for professionals for the 90 day period prior to May 20, 1997, where May,

1997 was the current (13th) month. Since the historical payment tables in the split environment only covered the 12 months up to the end of April, the same query was modified to retrieve payment data for the 90 day period prior to April 30, 1997.

In the split environment, there were actually two distinct 3-up query streams tested, as compared to the single 3-up stream used against the unified tables. One of the 3-up streams executed `qspl1a`, `qspl3` and `qspl5a`, which is to say two streams of long queries and one of short queries against the historical data. The other 3-up stream consisted of `qspl1a`, `qspl3` and `qspl6`, where `qspl6` was the hot spot query stream against the current tables. In this way, the performance impact of hot spot queries could be measured when they were executed along with other historical queries as well as when they were the only queries running. In the performance results, the two 3-up query streams in the split environment are identified as the 'generic' 3-up queries and the '3-up queries including the hot spot stream'.

The concept of 'repetitions' for query workloads was only meaningful for `qset3/qspl3`: the short query streams. In their case, they were always asked to execute 45 repetitions of their two short queries, although again the query parameters changed with each repetition. The long query streams were only executed once whenever they were part of a performance test.

In a given performance test, the driver program would fork and launch one or more of the query and transaction application programs. If a transaction workload was included in the test then four client transaction programs were always executed

concurrently: two for the payment inserts and two for the other transactions. By varying the actual transaction programs launched, the driver could determine whether the test was against the split or unified environment, and whether a heavy, medium or light transaction workload was to be processed. If queries were part of the test, then the driver would launch one or three query programs which would target either the unified or split database, and which could also potentially be the hot spot query stream in the split environment.

All of the transaction and query workloads were first run standalone against the unified or split database. For example, the medium transaction workload (four client programs) was executed against the unified environment to determine its response time and other performance characteristics. Or a query stream was run standalone to measure its execution time.

The next step was to test all of the mixed workload combinations against both unified and split databases so as to measure the impact of mixed workloads on both query and transaction performance. There were two basic sets of tests here: one set looked at the impact of concurrent queries on transaction performance, while the second set measured the impact of concurrent transactions on query execution time.

In order to test the impact of queries on transactions, a volume of transactions with known standalone performance characteristics was executed while one or three query streams were executing concurrently. The length of time the transaction workload ran was determined by the number of repetitions it had to execute as specified in a control

table. By specifying the number of repetitions we were able to ensure, in this case, that all of the transaction processing completed prior to any of the concurrently executing query streams. In this way, we could be confident that the transaction performance results accurately reflected the impact of a particular query workload. Clearly, if some or all of the queries had completed prior to the transaction workload then the average transaction response times, for example, would not be a valid measure of mixed workload performance.

Similarly, when the intention was to measure the impact of concurrent transactions on query performance, the transaction volumes (repetitions) were increased such that for the entire time the queries were executing there was a given level of concurrent transaction processing taking place. That is, all four of the client transaction programs would run longer than any of the query streams.

When actually comparing transaction performance standalone to transaction performance in the presence of queries care was taken to ensure that the same number of repetitions were executed in the compared tests. So, for example, if the light transaction workload was run with 1140 repetitions of the payment insert transaction programs and 150 repetitions of the other transaction programs with a concurrent 3-up query stream, then the transaction performance results from this mixed test were compared to the standalone light transaction workload tests that executed the same number of repetitions.

While the number of transaction repetitions varied between tests, the ratios of the volumes of different transaction types was constant in all tests, with the exception of the

heavy transaction workload experiments. So, for example, medium and light transaction workload tests were run with 1900/250, 1520/200, 1140/150, 760/100 or 570/75 repetitions in the unified environment where 1900/250, for example, means that each payment insert program executed 1900 repetitions while the two other transaction programs executed 250 repetitions. Through trial and error it was found that with this particular ratio the four transaction programs would complete in approximately the same elapsed time.

Hence, in any given test, the ratio of payment inserts to, say, beneficiary queries was constant. Indeed, exactly the same ratios were used in the split environment as well. As a result, with the exception of the heavy transaction workloads, the nature of the transaction type mix was constant throughout all tests.

It should be observed that the performance tests were not intended to measure the impact of varying transaction arrival rates on, say, transaction response time. Indeed, there were no fixed arrival rates. Each client transaction program submitted transactions not according to some pre-determined arrival rate distribution, but rather simply after waiting for the previous transaction to complete, and then for the exponentially distributed sleep time to elapse. Hence, the transaction arrival rate in a given test varied as a function of both the transaction workload type, whether heavy, medium or light, and of the concurrently executing query workloads if any. That is, if there were concurrently executing queries, transaction response times would slow down, and hence so would arrival rates even if the mean sleep time remain fixed.

In other words, the intention was not to look at the effect of varying transaction workloads on transaction performance, but rather to investigate the impact of concurrent queries on transaction performance, and vice versa. To this end, the performance of a particular transaction workload was measured on its own, and then when executed with concurrent queries. This simulates a real world environment in which a database system is processing a particular transaction workload. Then a decision is made to execute queries against the same database concurrently with the transactions. What happens to transaction performance? Transaction volumes will remain the same, even if response time increases. Moreover, users will likely continue to need the same time between transactions to prepare the next transaction for input (sleep time remains constant).

Of course, it is also possible that users would reduce the time between transactions in an effort to compensate for longer response times and, in particular, to maintain throughput rates. In effect, this is equivalent to converting, say, a light transaction workload into a medium transaction workload. The feasibility of maintaining throughput rates by reducing response times is discussed in *Chapter 4: Results and Analysis*.

All test types were executed multiple times in order to increase the accuracy of the results. In general, a given test was repeated five to ten times, although two tests that ran for hours were only repeated four times. A 'test' for example, would be executing the light transaction workload against the unified database with three concurrent, generic query streams such that the queries all completed before any of the transaction programs did. This particular test was repeated five times even though each execution lasted over

two hours. On the whole, the shorter the test the more frequently it was repeated, up to ten times.

Tests were executed in 'suites' under the control of a Unix shell script. Between each test the shell script would copy in a new version of the control table that specified which client programs the driver should launch and for how many repetitions. The driver would then be started, and upon test completion the shell script executed a restore of the split or unified database depending upon which had been targeted in the test.

The restore between each test ensured that all tests against a given database were run against exactly the same data. A second function of the restore was to effectively clear any memory buffers between tests so that no repeated tests would benefit from data buffered in memory from the previous execution. It should be noted that restores were even run before and after query-only tests, even though they did not modify the data, precisely so as to clear any data specific to the queries from memory.

All of the application programs output two files from each test repetition: one contained the performance data and the other any error or informational messages. The error and informational messages were sufficient to ensure successful execution of the program. In the case of the transaction programs, every 20th response time was also written to the informational messages file. Every output file was appended with the test number for the particular repetition, making it straightforward to associate all of the output files for a given test run. As mentioned above, the driver program would

Pgm Name	Elapsed	Repe.	Mean	Std Dev.	Rejects
qset1a	3282	1	3282.000	0.000	
qset3	2945	45	64.780	29.010	
qset5a	3139	1	3139.000	0.000	
t1	2372	1900	1.058	0.762	11
t1b	2355	1900	1.042	0.698	13
t2u	2253	1000	0.498	0.598	0
t2i	2253	250	0.094	0.181	0
t2q	2253	500	2.149	1.668	0
t2bu	2180	1000	0.480	0.483	0
t2bi	2180	250	0.139	0.465	0
t2bq	2180	500	2.014	1.551	0

Figure 2: Driver program output from a typical test execution

consolidate the output from the different programs executed in a test run into a single test result file, of the type shown in Figure 2.

In all, 89 different types of performance tests were executed in over 800 actual test runs.

3.6 Metrics

Please see Figure 2 for the output produced by the driver program from a typical test run. The particular test whose output is shown in Figure 2 was intended to measure the impact of the 3-up generic query streams on the medium transaction workload in the unified environment. The three query streams executed were qset1a, qset3 and qset5a. The four transaction programs were t1, t1b, t2 and t2b. All seven programs would have been launched in forked processes by the driver program within less than a second of each other.

The t2 and t2b transaction programs executed the three ‘other’ transaction types: payment updates, beneficiary queries and beneficiary inserts. In order to permit analysis of the performance of the individual transaction types, the results for t2 and t2b were split in three. So, for example, the ‘t2u’ line gives the results for the payment updates executed by program t2, while t2i and t2q provide the beneficiary insert and query results. Programs t1 and t1b only executed payment inserts and so only needed one output line.

The elapsed time indicates the total time in seconds required by the program to execute all of its transactions or queries, which would include sleep time in the case of transactions. ‘Repe.’ is the number of repetitions that the program ran. So, for example, t1 executed 1900 payment inserts. The actual ‘repetitions’ value for program t2 was 250 which corresponds to the number of beneficiary inserts processed. For convenience, the repetitions columns also shows the actual number of payment update and beneficiary query transactions processed by t2 and t2b, namely 1000 and 500 respectively.

The ‘mean’ column is the mean response time in seconds, which is only useful for the transaction programs. This result was not calculated by dividing the program elapsed time by the number of repetitions since, again, the elapsed time included the sleep time. Instead, the transaction programs calculated the response time average based on the amount of time it took the DBMS to process the associated SQL requests, including the final commit.

The standard deviation column represents the standard deviation of the response times for the particular program and transaction type. The rejects column reports the

number of transactions that were rejected by the DBMS due to integrity violations. As mentioned above, the test data was deliberately created in such a way that a small number of transactions would fail an integrity check. For example, a beneficiary might not be admissible.

Having obtained all of these detailed results from several hundred test runs, the next step was to consolidate the data into the following four metrics per run:

- **Weighted average transaction response time:** this was the average response time of all the different transactions types executed in a particular test run weighted by the number of times a particular transaction type was executed. So from Figure 2, the mean response time for the medium transaction workload in the presence of the generic 3-up query streams against the unified data was calculated as $(1900 * 1.058 + 1900 * 1.042 + 1000 * 0.498 + \dots + 500 * 2.014) / 7300 = 0.974$ seconds, where 7300 was the total number of transactions executed in the test run.
- **Weighted average transaction elapsed time:** This is the average completion time of the four transaction programs weighted by the number of transactions executed by each program. For Figure 2, this value is $(2372 * 1900 + 2355 * 1900 + 2253 * 1750 + 2180 * 1750) / 7300 = 2293$ seconds.
- **Weighted average coefficient of variation of the transaction response times:** To determine this value the weighted average standard deviation of the transaction response times was first calculated. The result was then divided by the

weighted average transaction response time to produce a coefficient of variation metric for the test run.

- Average query stream elapsed time: a simple average of the query stream completion times for a particular test run. For Figure 2, this result is $(3282 + 2945 + 3139)/3 = 3122$.

It should be emphasised that while the response time metric excludes sleep time between transactions, the elapsed time metric includes it. So the elapsed time indicates the wall-clock time required to execute the transaction volume in the workload, which is the total number of transactions executed by all four client transaction programs. By dividing the total number of transactions executed by the elapsed time one can arrive at a measure of transaction throughput: transactions executed per unit time. In other words, the elapsed time metric can be viewed as a surrogate for transaction throughput: the longer the elapsed time for a given volume of transactions the lower the throughput rate.

It is the above four metrics from each test run that were the primary input to the analysis of performance results. Of course, not all metrics were relevant to every test run. For example, in the standalone transaction workload tests there would be no average query stream elapsed time. Or again in a mixed workload test focusing on the impact of a transaction workload on query performance only the average query execution time was significant, assuming that no transaction programs completed before any of the query streams.

In order to ensure that the metrics were computed accurately, a program was written that read the test output files from the driver program and calculated the four summary metrics for each test. The summary metrics were then loaded into a spreadsheet program for analysis.

A preliminary calculation performed in the spreadsheet program was to further average the results from all of the test runs for a particular test type. Normally, there were five to ten test runs per test type. So, for example, there were ten test executions of the medium transaction workload in the unified environment with the generic 3-up query streams running. Here the issue was the impact of the queries on transaction performance. When the weighted average response times from these ten test runs were themselves averaged we finally had a response time metric for the transactions in this particular mixed workload: 0.980 seconds. Similarly, the results for the other three metrics were also averaged across all of the test runs of a given type in order to arrive at values which characterised a particular workload.

A question that needs to be addressed is whether or not the observed differences in performance are necessarily statistically significant. In general, the reported differences in performance results in Chapter 4 are sufficiently large that there is no doubt as to their significance.

Consider one of the smallest claimed performance differences reported in Chapter 4. In Figure 9 we report the transaction response times for various standalone and mixed workloads in the split and unified environments. The graph in Figure 9 illustrates that for

the medium transaction workload in the presence of 3-up queries the mean transaction response time in the split environment was 0.928 seconds while in the unified environment it was 0.980 seconds. This is a difference of only 5%. Is it a real difference or is it within the limits of measurement error such that there may in fact be no difference in mean response times between the two environments?

The mean response time for the unified environment is an average of the results from ten tests runs of the medium transaction workload with the 3-up query streams. In the split environment, five similar tests were executed. Here are the mean transaction response times from the individual tests:

Unified Results	Split Results
0.978	0.948
0.974	0.930
0.976	0.930
0.983	0.916
0.969	0.915
0.985	
0.979	
0.964	
1.001	
0.994	

It can first be observed that the response time results in the different tests did not overlap at all. The minimum average response time from any unified test was 0.964 seconds, while the maximum average response time from any of the split tests was 0.948

seconds. That is, the average response time of every split test was less than the average response time of every unified test.

Secondly, consider the variation of the test results within each environment with respect to the mean response time for all tests in the environment. For the unified database, the response time results varied from 0.964 to 1.001 seconds which are, respectively, 98.4% and 102.1% of 0.980 seconds, the average result. For the split database, the observed response times varied from 98.6% to 102.2% of the average result of 0.928 seconds. That is, the results of the individual test runs are within about 2% of the average result for a given environment. So it is fair to conclude that a difference of 5% in the average response times for the split and unified environments is significant.

We have deliberately picked an example here of two results which are fairly close and yet whose difference is claimed to be significant. Virtually all, if not all, of the claimed differences reported in Chapter 4 are greater than 5%, so their statistical significance is not in question in view of the fact that the variability of the individual test results was low.

The case for the significance of the results becomes even stronger, however, when it is noted that the reported results in Chapter 4 are often patterns observed to hold for various combinations of workloads rather than for just a single pair of workloads. For example, when it is claimed that transaction response times are faster in the split than in the unified database as shown in Figure 9, this is not based solely on the 5% difference witnessed in the case of the medium transaction workload with the 3-up query streams. Rather, it is based on a comparison of the results for the six different workloads seen in

Figure 9. For all six of these workloads, transaction response times were shorter in the split than in the unified environment. Obviously, this cannot be the result of mere measurement error, and so the result is significant.

The tables in Appendix B include two measures which indicate the variability of results among different tests of a given type. For those Appendix B tables concerned with the impact of queries on transaction performance, the 'Standard Deviation of Mean Response Times' column records the standard deviation of the weighted average response times computed for the different test runs of the corresponding test type. (The variability of response times within a test is measured by the response time coefficient of variation).

Similarly, for those tables concerned with the impact of transactions on query execution time, the Appendix B column 'Standard Deviation of Average Elapsed Times' measures the variability of query elapsed time results among different tests of the same type.

With few exceptions, the standard deviation of the test results is less than three percent of the mean values. Since the results within each test type exhibited such low variability, we conclude that on the whole they are a good indication of the actual behaviour of the test system.

RESULTS AND ANALYSIS

4.1 Preliminary Results

4.1.1 DBMS Issues

Mixed query and transaction workloads are still relatively rare in practice so commercial DBMSs have not yet had a lot of experience with them. As a result, one has to be concerned about whether the DBMS being used is robust enough for mixed workloads.

On the whole, the particular DBMS employed for this research proved to be quite stable. It executed hundreds of mixed workload tests consistently and reliably. Most of those tests ran thousands of transactions as well as demanding queries.

However, two significant problems did occur which are documented here in order to highlight the types of difficulties that mixed workloads can cause for DBMSs, and to alert users of mixed workloads to potential problems. The fact that these problems exist in a commercial DBMS that is in widespread use is also further evidence that mixed workloads remain unusual.

The first problem occurred when payment deletes were included in the mix of transaction types processed concurrently with queries. An error message would be returned from query 1's cursor open statement saying that 'Tuple at tid already deleted'.

This implies that a row that query 1 was processing was deleted before the query had finished with it, and that the query was unable to recover from the deletion. Query 1 counted the number of services for which each professional had been paid in the last three months.

The query, of course, was being executed without placing any locks. In principle, therefore, the query should have been oblivious to changes in the data, including inserts, updates and deletes, while it executed. In particular, if a row was deleted that the query would have otherwise retrieved while it was executing, the query should have simply continued with or without the row. But that is not how the DBMS being used behaved, at least not in all cases.

As a result of this problem, payment deletes were removed from the transaction workload. But while the number of 'Tuple already deleted' error messages then declined substantially, they did not disappear altogether. Certain other queries would occasionally report 'Tuple already deleted' errors even though the only types of transactions being executed were inserts, updates and queries. Query 1 continued to fail occasionally when opening its cursor, and query 13 would also fail from time to time on a fetch. Query 13 listed the amounts paid by professional category since April 15 for services of a particular service type.

We hypothesise that updates are processed by the DBMS as a delete followed by an insert. Consequently, a query's cursor could be positioned at some tuple, control could be passed to the update which deletes the tuple, and then when the query starts running

again the cursor finds that it is positioned at a non-existent tuple. Once again, in principle the query should either continue without the tuple or use the updated tuple since it is running in read uncommitted mode. But the DBMS was unable to do so, at least in some cases.

It is entirely possible that user error is at the root of these problems. However, the queries always ran without fail if no transactions were executed concurrently. Assuming that it is a real problem, the DBMS vendor has perhaps fixed it in a subsequent release. However, there is no question that mixed workloads will be infeasible if queries abort due to concurrent deletes or updates against data targeted by the queries. Most deletes and updates will affect relatively current data, and it can be expected that many queries will also run against the same data, so the problem is a severe one. Fortunately for our research, once deletes were removed from the transaction mix, 'Tuple already deleted' errors were rare, and so test runs in which they occurred could simply be excluded from consideration.

The second problem with the DBMS that occurred in the course of test execution was that the log file would fill if transactions were run concurrently with long queries that included an insert/select. When the log file filled to a particular threshold level the DBMS would abort the query.

The main purpose of the log file is to support database recovery in the event of transaction failure. Normally, when an administrator-specified percentage of the log file space is filled (20% in our case) the archiver process wakes up and writes the required log

file records to a permanent journal file and then empties the log. But if a long insert/select was being executed at the same time, the archiver did not empty the log file which then quickly filled up with transaction data. Eventually an abort limit was reached which caused the query to be terminated. The total database size including indexes was 126 megabytes, while the log file was configured with 100 megabytes of space. The force abort limit was about 75 megabytes, so the log file was plainly large enough for the database. Typically, log files are much smaller than the overall database size.

Insert/selects are common in decision support applications. In order to execute complex queries, selected data is inserted into intermediate tables for processing by later SQL statements. Testing showed that the insert/select itself took no log file space. It merely interfered with the log file archiving process. This too must be considered a serious problem since it is not realistic to expect DSS applications to avoid insert/selects. Once again, the vendor has perhaps fixed the problem in a subsequent release of the DBMS. However, for purposes of our research we were obliged to remove insert/selects from the queries.

4.1.2 Hot Spot Queries in the Unified Environment

An attempt was made to create a set of 'hot spot' queries for the unified environment. This was done by writing a number of queries with date restrictions; typically the payment date in the Payment Detail table was tested to verify that it was greater than a specified date so as to capture the most recent data. As a result, such queries should have run against only 6% of the payment data once the date restriction had

been applied. The expectation was that since most transactions also ran against the current data a 'hot spot' would be created: a subset of the data that was used intensively relative to the other data.

What we found was that for all but the simplest queries the DBMS's optimiser would not take advantage of the date restriction early in its query execution plan. That is, it would perform a number of joins and other operations with all of the rows in the Payment Detail table before finally limiting the retrieved rows based on date near the end of the query. This result is particularly surprising given that the Payment Detail table was structured as a btree based upon payment date followed by control number as the key. Statistics were collected on the key columns, so the optimiser should have had some information regarding the distribution of date values in the table, and then been able to use the btree key to retrieve the 6% of the rows that satisfied the date restriction. But again, the qualification on date was not normally examined until the end of query execution.

As a result, the hot spot queries in the unified environment behaved just like the other query streams. The test results presented in Appendix B include the hot spot stream outcomes in the unified environment. But since they are so similar to the results for other query streams they are uninteresting and hence not discussed further. In effect, no 'hot spot' was actually created.

In the split environment, the hot spot stream ran against the current month tables. The queries did not need a date restriction to retrieve the current data; they simply

accessed the correct tables. In this case the hot spot results are quite different from those for other query streams.

4.1.3 Similarity of Different Mixed Workloads

Our results show that mixed workload performance with a single query stream does vary depending upon the nature of the queries. However, we also found that with the exception of the hot spot query stream in the split environment the performance of mixed workloads with three concurrent query streams was virtually identical regardless of the actual query streams executed. This is no doubt because the differences between individual query streams tend to average out when three query streams are run together.

An example of this is the behaviour of the putative hot spot query stream in the unified environment. While the queries in this hot spot stream did not behave as expected, they were nonetheless different queries from those used in the other query streams. 3-up mixed workload tests were executed in the unified environment in which the hot spot stream was substituted for one of the three 'generic' query streams. But as Appendix B shows, the mixed workload results were the same whether transactions were run with the three generic query streams or with the hot spot stream plus two of the generic query streams. Other 3-up tests in which the query streams were varied also produced virtually identical results.

Consequently, our tests only report the results for a single set of three 'generic' query streams, as opposed to reporting the results for a variety of 3-up query streams.

The exception, again, is the 3-up stream including the hot spot queries in the split environment.

4.2 Sar Results

The Unix System Activity Reporting utility (sar) enables one to collect resource utilisation statistics during any time period. For purposes of this research, sar statistics were captured every 20 seconds during most or all of the execution of each test run, and stored in files labelled by test number. Sar reports were then generated from the collected statistics files as input to examine resource utilisation.

Please see *Appendix A: Sar Results* for a complete summary of resource utilisation statistics by workload type. For each workload, the statistics for a number of test runs were examined to obtain representative results. On the whole, there was very little variation in resource utilisation between runs for a given workload.

Appendix A focuses on the three key system resources: CPU, Disk, and memory. CPU %usr and CPU %sys are the percentage of time spent by the CPU executing application code and operating system functions respectively. CPU %wio is the time spent by the CPU waiting for an input/output operation to complete, while %idle is self-explanatory. Disk 1 was the data disk on the test platform; all database I/O for data and indexes accessed disk 1.

According to M. Ault in *Unix Systems Administrator's Companion* [Ault96, p. 307], the 'Pages freed per second' value reported by sar can be used as a measure of

memory utilisation. The higher the value the more intensively memory is being used. When there is a shortage of memory, the operating system is obliged to free pages at a greater rate rather than allowing pages to stay in memory longer for potential reuse.

Most of the resource utilisation was by the DBMS since application processing was minimal. In fact, application processing was largely restricted to launching queries or transactions and tracking response times. According to the 'top' dynamic resource utilisation monitor, the t2 and t2b transaction programs each used about 0.4% of the CPU, while t1 and t1b each used about 0.13%. The query programs and sar occupied 0.05% or less of the CPU. The main driver program, sdriver, was reported by top to be consuming 0.00% of the CPU. All of the I/O performed by the transaction and query programs to write out response times and calculate results accessed the second disk, not disk 1. Similarly the sar data was written to the second disk. Consequently, the sar results reported in this chapter and Appendix A are almost completely determined by database as opposed to application resource consumption.

4.2.1 Standalone Workloads

We will first look at CPU, disk 1 and memory utilisation for standalone workloads using the sar data for the unified environment. The results for the split environment are similar. Please see Figure 3 for the CPU utilisation by workload. For purposes of Figure 3, the sar values CPU %usr and CPU %sys were added to produce 'CPU %useful'. This is to highlight the percentage of time the CPU was actually doing useful work rather than merely waiting for an I/O request to complete or sitting idle.

Figure 3: CPU utilisation for standalone workloads

It is clear from Figure 3 that long queries make much more intensive use of the CPU than either transactions or short queries. While the long queries can use 50% or more of the available CPU cycles without any other workload on the system, the transactions and short queries need less than 20% of the CPU. Long queries typically read entire tables and then process every row in the table. While a complete table scan is an expensive operation, long queries typically require the CPU to process every row in every block of the table, which creates a significant CPU workload. Many large intermediate

tables are generated in the course of complex queries, and again the CPU must process all of the rows in the intermediate tables.

In contrast, when a block is retrieved for a transaction, the CPU usually only needs to process one or a few rows in the block. In the case of short queries, indexes are used to locate the blocks that contain the needed data. Normally, only a small number of rows per block retrieved will meet the query selection criteria and hence require CPU processing.

The complementary chart is Figure 4: disk 1 utilisation. Here we see that the long queries made less intensive use of the disk than the short queries and transactions. We also see the essential difference between the medium and light transaction workloads: the medium transaction workload kept disk 1 90% busy while the light transaction workload only needed 72% of disk 1's time.

While the transactions and short queries placed a relatively light load on the CPU, they plainly made heavy use of the disk that contained the data and indexes. On the other hand, even though the long queries had to do complete table scans, they nonetheless spent more time doing CPU processing and less time doing disk I/O than the transactions and short queries. This is confirmed by the CPU %wio results shown in Appendix A. For the short queries and transactions the CPU %wio values are 62% to 75%, while for the long queries the comparable numbers are 37% and 46%.

Figure 5 shows the pages freed per second for each of the principal standalone workloads, which again is an indication of how intensively memory was being used. On

Figure 4: Data disk utilisation for standalone workloads

the test platform, the number of pages freed per second typically ranged from 10 to 100 when there was some activity on the system. It would peak over 200 per second under heavy load.

With one exception, the pages freed per second data suggests that memory was not a critical resource. The exception is the short queries; in their case the volume of data retrieved would have been small enough to fit into memory so that the database would have been able to do most query processing with little disk I/O. It is also possible that since the data needed by the short queries fit in memory the DBMS attempted to keep it there in case another query or transaction required it. The data used by a long query

Figure 5: Memory utilisation for standalone workloads

greatly exceeded memory capacity, so the DBMS would have been obliged to purge the data from memory as it was processed.

One implication of the sar data is that the mixed workload tests were largely I/O bound on the data disk (disk 1). Certainly, disk 1 was the most intensively used of the system resources. Since transaction workloads on their own used 72% or more of disk 1's capacity, there was not enough spare capacity on disk 1 to run any of the query workloads without some loss of performance.

It should be noted that the utilisation of disk 2 rarely exceeded 5%. This is in spite of the fact that the log and journal files were on disk 2, as well as the DBMS code and the

result files written by the test applications and sar. Consequently, one potential opportunity for improving performance would have been splitting the data itself across both disks so that disk use would have been more evenly balanced.

Another important conclusion that one can draw from the sar data is that system resource utilisation by long queries and transactions is complementary to at least some degree. That is, long queries use more CPU cycles and less disk I/O than transactions. Long queries are typically the dominant workload in a DSS environment from the point of view of system resource consumption. Hence, it is reasonable to expect that with a mixed workload the queries will be able to make effective use of the CPU capacity left by the transactions, provided that there is adequate spare disk I/O capacity to meet the queries' needs. In any event, it is clear that long queries are not as dependent on I/O capacity as transactions. Hence the impact of long queries on transaction performance should not be as severe as if long queries, too, used 90% of the disk capacity and 20% of the CPU.

4.2.2 Hot Spot Queries

In the split environment, the hot spot query stream exhibited very different resource utilisation numbers when run standalone compared to the workloads discussed above. As seen in Appendix A, the CPU spent 84% of its time doing useful work while the hot spot queries were executing (79% usr and 5% sys). Only 15% of the CPU cycles were spent waiting on I/O. Disk 1 was idling along, comparatively speaking, at 23%, while the number of pages freed per second was very low, indicating that there was no memory shortage.

The hot spot queries in the split environment targeted the small, current month tables. So what was doubtless happening was that the entire current month payment tables were being read into memory and processed there. The current month tables contained about 7 megabytes of data and indexes, while the total volume of data in all the other tables excluding payment history was about 8.2 megabytes. Given that the test system had 32 MB of memory installed, it would have indeed been feasible for the DBMS to read all of the necessary data for a hot spot query into memory. When the next hot spot query ran, most or all of the required data would still be in memory so that relatively little disk I/O would be required. Such I/O as did take place was likely needed for sorts and other intermediate files. Again, the CPU would typically have needed to process all of the rows in each data page in memory, at least for the payment tables. Since there was little waiting on I/O CPU resource utilisation was high.

4.2.3 Mixed Workloads

How can we characterise the system resource utilisation for mixed workloads? First, note in Appendix A that the disk 1 %busy for mixed workloads was always higher than the disk 1 %busy values for the corresponding standalone workloads. For example, the disk 1 %busy figure for the medium transaction workload running with the 3-up queries in the unified environment is 95%. For the medium transactions standalone the disk 1% busy was 90%, while for the 3-up queries standalone it was 88%.

The same is not true, however, for the time the CPU spends doing useful work (CPU %usr + CPU %sys). Again in the unified environment, the CPU %useful was 41%

for the medium transactions with qset1a. This was greater than the CPU %useful of 18% for the medium transactions standalone, but less than the 59% useful for qset1a standalone.

This data suggests once again that the critical system resource in the test platform used was the data disk. When mixed workloads were run the data disk was pushed harder than in the corresponding standalone workloads, and was doubtless the major constraint on performance. CPU utilisation with mixed workloads, however, was typically the same as or less than the CPU utilisation of the corresponding standalone workloads. That is, the system was typically not able to make more effective use of the CPU when running mixed workloads. There can be little doubt that the constraint on useful CPU work when running mixed workloads was access to the data disk. In particular, while long queries made good use of the CPU when running standalone, they needed to share access to the data disk with transactions when running in a mixed workload, and hence their utilisation of the CPU went down as they spent more time waiting on I/O.

We see a similar result with the hot spot query stream in the split environment. Run by themselves the hot spot queries used 84% of the CPU cycles for useful work. But when they were run with the medium transactions the CPU %useful dropped to 41%. Disk 1 utilisation, again, went up to 86% versus the standalone workload results of 82% for the medium transactions and 23% for the hot spot queries. We can again infer that when run with the medium transactions the hot spot queries were required to do more waiting for I/O than when they were running standalone. Otherwise, whenever the

transactions were waiting on I/O, the hot spot queries would have done useful CPU work and so sar would have reported higher CPU %useful results for the mixed workload.

While the data disk was clearly a bottleneck, it is nonetheless the case that when transactions were run with long queries in a mixed workload the system was able to make more use of both the CPU and the disk than the transaction workload did on its own. In other words, the mixed workload made more effective use of system resources than the transaction workload by itself. The same cannot be said of mixed workloads including the short queries (qset3 and qspl3). In that case, while the disk 1 %busy went up, the CPU %useful was essentially the same as for the transaction workload standalone. Thus we see that from the point of view of effective system utilisation long queries are more promising than short queries in a mixed workload environment.

As final observations on the sar data we note first that system utilisation for corresponding mixed workloads was much the same in the unified and split environments. Compare, for example, in Appendix A the light transactions with qset3 in the unified environment to the light transactions with qspl3 in the split environment. Secondly, within a given environment, unified or split, the sar results for a mixed workload with the light transactions were very similar to the corresponding mixed workload with the medium transactions. Compare, for example, in the unified environment the sar values for the medium transactions run with the 3-up queries to those for the light transactions with the 3-up queries.

4.3 Performance Results

We now turn to the performance of transactions and queries in a mixed workload environment. What happens to transaction response time, throughput, and response time variability in the presence of queries? What is the impact of concurrent transactions on query execution time? How do these results compare for different transaction and query workloads?

For convenience we present a summary of the results in Tables 2, 3 and 4 in this section. But for complete results please see *Appendix B: Performance Results*. In addition to the data shown in Tables 2, 3 and 4, Appendix B includes absolute elapsed and response times, and results for standalone as well as mixed workloads. In Table 4 and elsewhere ‘T/C’ is an abbreviation for ‘transaction’.

In Tables 2 and 3 the last column is titled ‘New/Old CV’. The issue is the impact of concurrent queries on transaction response time variability. ‘New/Old CV’ is the coefficient of variation of transaction response times of the ‘New’, i.e., mixed, workload divided by the coefficient of variation of the ‘Old’ workload: the transactions standalone. So, for example, in Table 2 the New/Old CV for the heavy transaction workload run with the 3-up queries is 0.87. This value is the coefficient of variation for the mixed workload, 0.781, divided by the coefficient of variation for the heavy transactions running standalone, 0.894. Again, the absolute coefficients of variation for transaction response times are in *Appendix B: Performance Results*.

Impact of Queries on a Heavy Transaction Workload

	Elapsed Time Increase	Response Time Increase	New/Old CV
3-up queries	1.79	1.83	0.87
qset1a	1.35	1.36	0.98
qset3	1.43	1.45	0.91
qset5a	1.21	1.22	1.01

Impact of Queries on a Medium Transaction Workload

	Elapsed Time Increase	Response Time Increase	New/Old CV
3-up queries	1.91	2.55	0.77
qset1a	1.40	1.70	0.91
qset3	1.50	1.84	0.85
qset5a	1.33	1.58	0.92

Impact of Queries on a Light Transaction Workload

	Elapsed Time Increase	Response Time Increase	New/Old CV
3-up queries	1.86	3.50	0.71
qset1a	1.41	2.18	0.99
qset3	1.41	2.20	0.81
qset5a	1.29	1.81	0.93

Table 2: Unified data: Impact of queries on transactions

4.3.1 Heavy Transaction Workloads and Execution Priorities

From Table 2 we see that when the heavy transactions were run concurrently with a single query stream (qset1a, qset3 or qset5a), the transaction response time increases ranged from 1.22 to 1.45 times the standalone response times. Compared to the impact of

Impact of Queries on a Medium Transaction Workload

	Elapsed Time Increase	Response Time Increase	New/Old CV
Generic 3-up queries	2.13	3.15	0.73
3-up queries including the hot spot stream	2.26	3.47	0.70
qspl1a	1.56	2.09	1.05
qspl3	1.58	2.13	0.83
qspl5a	1.39	1.77	0.93
Hot spot queries	1.44	1.91	0.92

Impact of Queries on a Light Transaction Workload

	Elapsed Time Increase	Response Time Increase	New/Old CV
Generic 3-up queries	2.25	4.01	0.71
3-up queries including the hot-spot stream	1.95	4.18	0.63
qspl1a	1.52	2.76	0.93
qspl3	1.45	2.49	0.80
qspl5a	1.37	2.28	0.92
Hot spot queries	1.62	3.22	1.33

Table 3: Split data: Impact of queries on transactions

the same query streams on the medium and light transaction workloads, this is a relatively small increase in response time.

Contrast this result with the impact of heavy transactions on the execution time of the same query streams in Table 4: those increases varied from 4.53 to 7.28 times the standalone query execution time. In other words, the impact of queries on transactions and of transactions on queries is asymmetrical: the heavy transactions caused a much

Unified Data Results

	Increase with heavy T/C workload	Increase with medium T/C workload	Increase with light T/C workload
3-up queries		2.99	2.44
qset1a	6.39	3.08	2.21
qset3	4.53	3.39	2.47
qset5a	7.28	3.49	2.39

Split Data Results

	Increase with medium T/C workload	Increase with light T/C workload
Generic 3-up queries	2.58	2.08
3-up queries including hot spot stream	1.43	1.19
qspl1a	2.52	1.91
qspl3	3.05	2.25
qspl5a	2.79	2.08
Hot spot query stream	1.39	1.22

Table 4: Impact of transactions on queries

larger degradation in query performance than the queries caused for the heavy transactions.

Why should this be? We hypothesise that the longer a 'transaction' runs the lower the priority it receives, either from the DBMS or the operating system or both. Here, the term 'transaction' is being used in the technical sense and therefore includes queries since from the DBMS point of view a query is indeed just another transaction which remains alive until it completes or does a commit or rollback.

Since there was no mention of a priority manager in the DBMS vendor's documentation, the most likely scenario is that the DBMS is executing all transactions at the same priority level, while the Unix operating system is lowering the priority of long running queries.

In an attempt to improve query performance, a Unix 'nice' command was added to the client transaction processing applications to reduce their priority as far as Unix allowed (nice(19)). When the qset5a query stream was executed concurrently with the heavy transaction workload without the nice command being applied to the transactions, it ran for 1635 seconds. (The transactions completed prior to the qset5a queries.) With the transactions 'niced' to the lowest permissible priority, qset5a ran for 1639 seconds, an identical result. If the nice command had indeed reduced transaction priority, qset5a should have completed sooner.

One is therefore tempted to conclude that it is not the operating system but rather the DBMS that is lowering the priority of long-running queries versus short-running transactions. However, it is likely that the nice command applied to the client transaction programs had no impact since the client programs do very little processing themselves. In order to execute an SQL statement the DBMS spawns additional Unix processes which will not be subject to the nice command. But after scheduling competing processes in a round-robin fashion for a period of time, Unix typically reduces the number of time slices accorded to long-running tasks. That is, Unix will dynamically lower the priority of time-consuming processes. Some DBMSs do have internal priority managers, and the test

DBMS may also have had a role to play in this regard. If not, however, the fact that long running queries appeared to receive a lower level of priority than short transactions is consistent with normal Unix behaviour.

Dynamically lowering the priority of a transaction as a function of the length of its execution would be a reasonable priority management policy to use in a mixed workload environment. The assumption is that new, short transactions should benefit from reasonably fast and consistent response times, while longer running transactions are assumed to be of lower priority and able to sustain some delay. Note, however, that with this policy all transactions, including long queries, start with the same high priority and only have their priority lowered after they have been executing for some time.

The policy of dynamically lowering the priority of long-running tasks helps to explain a number of the test results, but is most clearly seen in the mixed workload results for the heavy transaction workload: response times for the short transactions degraded very little, while the much longer queries suffered much greater degradation in the presence of the heavy transaction workload. The queries are close to being 'starved' of system resources. Indeed, they would run for so long that it was not possible for the 3-up query workload to complete while the heavy transactions were running given that there was a finite volume of transaction input data.

We can see from the sar data that the heavy transaction workload on its own used 98% of disk 1's capacity. Since Unix or the DBMS apparently gives shorter transactions greater priority than longer ones, this means that very little disk capacity would be

available to queries which had been executing for some time. The heavy transaction workload left no spare disk capacity for longer running queries to use. Hence they took an inordinately long time to complete.

It is noteworthy, however, that the short queries (qset3) did better than the long queries: they only took 4.53 times as long to complete with the heavy transaction workload versus 6.39 and 7.28 times as long for the lengthy queries. This result is further evidence that the longer a transaction executes, including queries, the lower its priority becomes and hence the greater its execution time degradation when run concurrently with a high priority workload.

In our view, a concurrent query and transaction workload will be unacceptable to most users if long queries, which are at the heart of DSS applications, take six or seven times as long to complete as they do standalone. If a concurrent workload is to be feasible at all, at least with the test DBMS and platform being used, the demands of the transaction workload on the system must be reduced. As a result, we did not conduct complete tests with the heavy transaction workload. In particular, no tests with the heavy transaction workload were attempted in the split environment. Thus the focus of our discussion of performance results will be on the medium and light transaction workloads, which were less demanding of system resources.

4.3.2 Impact of Queries on Transaction Response Times

What happens to transaction response time in the presence of a concurrent query workload? In a first step towards answering the question, Figures 6 and 7 compare the

Figure 6: Response time increases for medium and light transaction workloads

response time degradation experienced by medium and light transaction workloads in the presence of queries. Here, a number of results from the unified and split environments have been averaged together.

What we see in Figure 6 is that in relative terms the light transaction workload experienced a greater response time degradation than the medium transaction workload when it was run concurrently with queries. For example, the response time increased

about 3.00 times on average for the medium transactions when they were run with 3-up queries, while light transaction workload response time increased about 3.87 times.

The corresponding absolute transaction response times are shown in Figure 7. Again, these represent average values across a number of unified and split test results. When run standalone, the light transactions had a faster response time than the medium transactions. This is to be expected since the light transaction workload placed fewer demands on system resources and so would have experienced less contention than the medium transaction workload. But when run with concurrent queries the average response times for the light transaction workload were actually marginally greater than the corresponding medium transaction workload values. Hence the relative degradation in response times was much greater for light transactions as seen in Figure 6.

This is an example of a pattern which is seen in the test results generally: lighter workloads tended to be more vulnerable to performance degradation than heavier workloads when executed concurrently with other tasks. Another example is the response time degradation of heavy transactions in the presence of a single concurrent query stream which we saw above: the average degradation was 1.34 times, much less than the 1.86 multiple experienced by the medium transaction workload.

Returning to the specific case of light and medium workload transaction response times, why were they so similar in mixed workloads when the light workload enjoyed a significant advantage standalone? In the discussion of the sar data it was noted that system resource utilisation was much the same for corresponding mixed workloads where

Figure 7: Absolute transaction response times

one mixed workload involved the light transactions and the other the medium transactions. That is, while there was a significant difference in the level of system activity as reported by the sar data between the light and medium transaction workloads when run standalone, that difference disappeared when the transactions were run concurrently with queries.

As a result, in a mixed workload the light transactions experienced the same level of contention for access to system resources as the medium transactions. Hence the similarity in response times. In fact, it appears that transactions in the light workload were less well able to contend with concurrent queries than those in the medium workload since

light transaction response times were marginally higher in mixed workloads. Perhaps the more frequent transactions in the medium workload were making more efficient use of data and code caching than the less frequent transactions in the light workload.

In any event, it is clear from the test results that lighter transaction workloads were more vulnerable to response time increases than heavier workloads, although the absolute response times in the presence of queries were similar.

We now want to compare transaction response time increases in the unified and split data environments. These results are depicted in Figures 8 and 9. Figure 8 illustrates the relative response time increases, while absolute response times are charted in Figure 9. It should be noted that the results shown in these charts, as in Figures 6 and 7, typically average a number of individual results, although the hot spot query stream results are excluded. For example, to derive a relative response time increase for the medium transaction workload running with one concurrent query stream, the results for the three different query streams were averaged. The reader who wishes the most accurate possible numbers should consult *Appendix B: Performance Results*.

What we find from Figures 8 and 9 is that the unified environment did better than the split environment in terms of relative increases in transaction response times for concurrent workloads, but that split data consistently yielded better absolute transaction response times. From Figure 9 it can be seen that the absolute response times for the standalone transaction workloads were also better in the split than in the unified environment.

Figure 8: Unified versus split data response time increases

The sar results in Appendix A show that the standalone transaction workloads placed fewer demands on system resources in the split than in the unified environment. For example, the light transaction workload left 31% of the CPU cycles idle and required only 61% of disk 1's capacity in the split environment, while for the unified data the same workload only left 23% of the CPU idle and needed 72% of disk 1's time. That is, the light transaction workload was even 'lighter' from a resource utilisation point of view when the data was split. So for standalone workloads, there was less contention among transactions for system resources in the split environment, and hence better response times were experienced than when the data was unified.

Figure 9: Absolute transaction response times in the split and unified environment

Turning to the mixed workloads, we have another example of the pattern referred to earlier: lighter workloads were more vulnerable to performance degradation when run concurrently with other tasks than heavier workloads. In this case, the lighter split workloads experienced greater relative response time degradation than the heavier unified workloads, even though, in this case, the absolute response times for the 'light' (i.e., split) workloads were better than for the 'heavy' (i.e., unified) workloads when run concurrently with queries.

It appears that the response time advantage enjoyed by the split environment was largely lost when the transactions were run concurrently with queries: that is, the level of system resource utilisation was much the same for the split and the unified environments for a given mixed workload as noted in the discussion of the sar results. At best, disk utilisation was marginally lower for split than unified data, which is reflected in the marginally superior absolute response times for mixed workloads in the split environment.

But since the overall level of system activity was much the same for a given mixed workload in the split and unified environments, it stands to reason that the response times were much the same as well. And since the standalone transactions ran much faster with the split data, it follows that the relative degradation of transaction response times was greater for split than unified data.

Another obvious and expected result that we obtain from Figures 6, 7, 8 and 9 is that a 3-up query stream caused greater transaction response time degradation than a single query stream. However, what may be somewhat surprising is that the degradation caused by the 3-up query streams was always less than three times that caused by the corresponding single query streams. For example, we see in Figure 8 that for unified data the transaction response time increased 1.71 times on average when the medium transaction workload was run concurrently with a single query stream. If each additional query stream caused a further 71% increase relative to the standalone response time then the 3-up queries should have resulted in a response time increase multiple of 3.13 ($1 +$

3(0.71)). But the actual increase with 3-up queries was only 2.55 times the standalone value.

Suppose that all concurrently executing transactions and queries received the same level of service. Then adding a single query stream could be expected to approximately double transaction response time assuming that both workloads standalone used 100% of the system resources. Adding a second query stream would then split the system resources three ways, and so the transaction response time would be three times its initial value. With a third query stream transaction response time should increase four times. That is, the increase in transaction response time according to this simple model should be linear with the number of concurrent query streams.

But these are not the observed results. Instead, it appears that the first query stream run concurrently with transactions caused a significant response time degradation, while the addition of further concurrent query streams caused incrementally less degradation. Unfortunately, time did not permit running tests with two, four or other numbers of concurrent query streams, so we cannot say whether this pattern would be borne out by them. This is a subject for future research. But we do know that 3-up queries caused less than three times the response time degradation caused by a single query stream.

It should be pointed out here that as response time increased transaction throughput declined. That is, the cost of executing transactions concurrently with queries took two forms from the point of view of the transaction workload: increased individual

transaction response times and an overall decline in transaction throughput. That is, no attempt was made to hold transaction throughput constant while concurrent query streams were added. The feasibility of maintaining constant transaction throughput in the presence of new query workloads is discussed in the last section of this chapter: *Concluding Comments on the Results*.

But even allowing for the decline in transaction throughput, one still wants to know why three concurrent query streams resulted in less than three times the response time increase caused by a single query stream. Perhaps when the transactions were running standalone they were benefiting from a significant degree of data and/or code caching. As soon as one query stream was added, much of the caching efficiency may have been lost resulting in significant increases in transaction response times. When additional query streams were added most of the caching efficiencies had likely already been lost, and so the declines in response times were smaller.

It should be noted that the results for the impact of a single query stream on transaction response times averaged in the impact of the short query stream which was very memory intensive. In fact, according to the sar data the short queries on their own placed more stress on memory than the 3-up queries. So it is indeed possible that most of the performance degradation due to memory shortages and hence the loss of caching efficiencies is reflected in the single-query mixed workloads. Whatever the explanation, it is certainly useful to know that the transaction response time increases caused by the first query stream are greater than those caused by additional query streams.

As a final observation regarding the impact of queries on response times, we note from Tables 2 and 3 that in general the short queries (qset3, qspl3) had a greater impact on transactions response times than the longer queries (qset1a, qspl1a, qset5a, qspl5a). For example, in the unified environment the short queries, qset3, caused the response time of light transactions to increase 2.20 times, while qset5a, a single long transaction, only caused an increase of 1.81 times.

This is further evidence of the priority management scheme hypothesised above: the operating system (or DBMS) gives high priority to new, short transactions and dynamically lowers priority the longer a transaction, including a query, runs. Under this policy, the numerous short queries in qset3 received a higher priority level of service compared to the one or two long queries in qset1a and qset5a. Thus it can be expected that qset3 would cause a greater degradation in transaction response time than qset1a or qset5a, and this expectation is generally borne out by the results. The exception is in the split environment for the light transaction workload: qspl1a caused response time to increase 2.76 times while qspl3 only resulted in a 2.49 times increase. This likely has a specific explanation in terms of the nature of the queries and transactions.

This concludes a somewhat lengthy discussion of the impact of concurrent query execution on transaction response time, certainly a key issue for mixed workloads. For convenience we summarise the key findings:

- when three concurrent query streams were being run the queries could be varied and the impact on transaction response time was essentially the same.

- if only a single query stream was being executed then short queries normally caused a greater increase in response time than long queries.
- concurrent queries had a greater relative impact on response time for the light transaction workload than for the medium workload. In absolute terms, the response times for the light workload were marginally greater than for the medium workload.
- relatively speaking, queries caused larger increases in response times in the split than in the unified environment. But absolute response times were always better with the split data.
- the response time increase caused by 3-up query streams was proportionately less than the increase caused by a single query stream.

4.3.3 Impact of Queries on Transaction Throughput

Transaction throughput is measured in terms of the number of transactions processed per unit time. As noted in *Chapter 3, Methodology*, throughput is the inverse of the elapsed time for a given volume of transactions. As throughput goes down, the elapsed time to execute a given number of transactions increases.

Since application processing time was negligible, the two significant factors that determined throughput were the sleep time between transactions and response time (i.e., processing time in the DBMS). The average sleep time between transactions was held constant as query streams were added to transaction workloads, so it is not surprising that

decreases in transaction throughput (increases in elapsed time) were less than the corresponding increases in response time. For example, while the 3-up queries caused transaction response time to increase 2.55 times for the medium workload, the elapsed time increase multiple was only 1.91.

We have seen that queries caused a greater relative degradation in response time for light transaction workloads than for medium transaction workloads. One might therefore expect that throughput declines for light transaction workloads would have been greater than for medium workloads. Such, however, was not the case. As seen from Tables 2 and 3, the elapsed time increases for corresponding medium and light transaction workloads were much the same when run with either 3-up queries or single query streams. For example, `qset1a` caused the elapsed time of the medium workload in the unified environment to increase 1.40 times, while for the light workload the corresponding figure was 1.41.

The explanation for this result is based on the relative importance of sleep time and response time in determining the throughput of the different transaction workloads. Again, the average sleep time between transactions for a given workload was held constant; indeed, it was the average sleep time that determined whether a workload was heavy, medium or light. So the lighter the workload, the greater the proportion of sleep time, and the less the influence of response time on throughput.

If sleep time had been held constant for all workloads, then response time increases would have been mirrored by proportionate declines in throughput, and hence

proportionate increases in the elapsed time required to execute a given number of transactions. But since light workloads had more sleep time, increases in response time had proportionately less effect on throughput than for medium workloads.

4.3.4 Impact of Transactions on Queries

For most organisations contemplating the execution of mixed workloads in a single database system, transactions will be considered more important than queries. That is, acceptable and consistent transaction response times will be essential to the conduct of day to day business, whereas decision support queries can be allowed to take more time.

Nonetheless, if a mixed workload is to be feasible the DSS queries must still receive a reasonable level of service. Otherwise, organisations that must be able to run large queries will be obliged to establish separate data warehouses with all of the costs that entails. So while the focus thus far has been on the impact of queries on transaction performance, we now turn to query performance in the presence of concurrent transaction workloads.

We have already seen how queries were affected by the heavy transaction workload: on average, a single stream took 6.07 times longer to execute concurrently with a heavy transaction workload than it did standalone. As argued above, this is likely due to the fact that Unix dynamically lowered the priority of long-running queries. Since the heavy transaction workload consumed virtually all of the critical resource, disk 1, by itself, the queries received very little service.

Figure 10: Impact of transactions on query elapsed time

Results for medium and light transaction workloads are depicted in Figures 10, 11 and 12. In Figure 10 we see the relative increases of query elapsed times in the presence of transactions, while Figures 11 and 12 show the average absolute query execution times for 3-up and single query streams both standalone and with concurrent transactions.

Needless to say, the elapsed time increases with medium and light transaction workloads are much less severe than with the heavy workload. The average increase for a single query stream in the unified environment with a medium transaction workload was 3.32 times the standalone query time, while the corresponding figure for the light transaction workload was 2.36.

Figure 11: Absolute execution times for 3-up queries

The most striking result from Figures 10, 11 and 12 is that queries invariably did better in the split environment than when the data was unified. It needs to be kept in mind that the queries in the split environment were only running against 94% of the data volume faced by queries against the unified tables. The other 6% of the data was in the current month tables which the queries in the split environment did not see. Moreover, as payment insert transactions were executed, they added data to the payment tables which queries in the unified environment were reading. In the split environment the new payments went into the current month tables and hence were not seen by the decision support queries.

Figure 12: Absolute execution times for a single query stream

When run standalone, 3-up queries in the split environment only took 88% as long as queries in the unified environment, while single query streams took 91% as long on average (excluding the hot spot stream). So even without concurrent transactions, queries did better in the split environment than the difference in data volume versus the unified tables would explain.

When run with concurrent transactions the gap between the query times in the split and unified environment became even larger: averaging the results with medium and light transaction workloads, the 3-up queries took only 76% as long to complete against the split as compared to the unified data, while the single query streams only took 79% as

long. Figure 10 shows that in relative terms as well, the elapsed time increases for queries against split data were less than in the unified environment.

Why should queries have fared so much better in the split environment in the presence of transactions? One possible explanation is that the queries were benefiting from the premise of the split environment: that long queries should be restricted to unchanging, historical data while transactions modify current month data. Thus, when a query in the split environment accessed the historical `Payment_Detail` table for example, none of the data in the table was changing; no inserts, updates or deletes were being processed against it. All such changes were to the `Current_Detail` table.

The situation was quite different for queries against the unified data. While a query was reading the `Payment_Detail` table, transactions were inserting new rows into the table and updating existing rows. Potentially, deletes could also have been taking place. Since the queries were being run at the read uncommitted isolation level, they would have been obliged to process data pages that were in the process of being modified by transactions. It may well be that coping with such 'dirty data' caused some inefficiencies for query processing in the database, versus processing pages which were not subject to change as in the split environment.

As a final observation regarding the impact of transactions on queries, we note that the relative elapsed time increases for 3-up queries in Figure 10 were approximately the same or less than the increases for single query streams. Transactions did not cause more problems for multiple query streams than for single query streams.

4.3.5 Hot Spot Query Stream in the Split Environment

To this point, the discussion of mixed workload performance results has excluded the hot spot query stream in the split environment because, as shall be seen, the hot spot query stream results are quite distinct from those for other, 'generic' query streams. Note again that all discussion of a hot spot query stream refers to the split environment since as observed in the *Preliminary Results* section above, the attempt to create a hot spot query stream in the unified environment was not successful.

A 'hot spot' in a database environment refers to some portion of the database, whether data, indexes or the data dictionary, that is used much more intensively than the rest of the database. Such hot spots are potential bottlenecks for database performance, and hence can cause overall performance degradation. For this research, the hot spot consisted of the current month tables. Hot spot queries ran exclusively against them, albeit without placing locks. Since most transaction activity was against the current month tables as well, they constituted an intensively accessed subset of the data and hence a hot spot. The chief question then was: would hot spot queries cause greater performance degradation than generic queries?

The hot spot query results are presented in Figures 13, 14 and 15. Figure 13 illustrates the relative increases of transaction response times with concurrent generic and hot spot queries. In Figure 14 the absolute transaction response times in the presence of queries are shown. Figure 15 shows how query execution times were affected by concurrent transactions.

Figure 13: Response time increase caused by generic and hot spot queries

From Figures 13 and 14 we see that hot spot queries had much the same impact as generic queries on the response time of transactions in the medium workload. However, for the light transaction workload, the hot spot queries caused a much greater increase in response time. That is, while the average increase in response time for the light workload with a single generic query stream was 2.51 times, the hot spot queries caused response time for the same workload to increase 3.22 times.

We know from the sar results in Appendix A that the hot spot query stream standalone was a CPU intensive workload. It consumed 84% of the CPU cycles for useful work. Compared to the other query streams it used very little of disk 1: only 23%. So the

Figure 14: Absolute response times with generic and hot spot queries

fact that the hot spot query stream caused a large increase in response time for the light transaction workload cannot be explained by contention for access to disk I/O.

Hot spot queries were typically shorter than generic queries because they only accessed the 6% of the payment data found in the current month tables. Therefore, hot spot queries would on average execute at a higher priority level than the longer running generic queries, given that Unix or the DBMS lowered the priority of a transaction the longer it ran. Hence, one would expect a hot spot query stream to have a more significant impact on transaction response time than a generic query stream.

Yet while this prediction was borne out for the light transaction workload, we see in Figure 13 that the impact of hot spot and generic queries on the medium transaction workload response time was much the same. So we still need to explain why hot spot

Figure 15: Impact of transactions on generic versus hot spot query execution times

queries caused a much larger increase in response time for the light transaction workload than generic queries.

One possibility may be that the light transaction workload was benefiting less from code and data caching in the presence of hot spot queries than when running with generic queries. Hot spot queries were very CPU intensive as noted above, so they were likely making effective use of memory to store at least data and possibly also code. Since hot spot queries also enjoyed relatively high priority, it may be that the application code and data used by the transactions was being paged out, which caused an increase in transaction response time.

The same phenomenon possibly did not occur for the medium transaction workload to as great a degree because paging policies are based on least recently used algorithms. The medium transaction workload used its application code and data more frequently than the light workload, so there was less likelihood that it would be swapped out of memory.

In Figure 15 we see that there is a striking difference between the impact of concurrent transactions on the execution times of generic queries and hot spot queries: generic queries take from 2 to 2.8 times as long to complete with concurrent transactions while the hot spot queries only take about 1.3 times as long. Again, these results are for the split environment only. From the query point of view, there appears to be a good ‘fit’ between the hot spot queries and concurrent transactions.

Again, the hot spot queries run primarily in the CPU and in memory, making relatively little use of disk 1. The transactions, of course, are almost completely disk I/O bound and spend most of their CPU time waiting on I/O, with the result that the transactions have relatively little impact on the execution of hot spot queries, even when the medium transaction workload is running.

Looking back at Table 4, we see that the excellent performance of the hot spot queries in the presence of concurrent transactions appeared to ‘spill over’ into the performance of the 3-up query stream when one of those streams was the hot spot queries. For example, the generic 3-up queries took 2.58 times as long to complete in the split environment when run with the medium transaction workload than they did running

standalone. However, if one of the 3 streams was the hot spot queries, the execution time only increased 1.43 times, a remarkable improvement.

The reader will recall that hot spots usually cause performance degradation in database systems, and indeed hot spot queries do cause a significant degradation in response time for the light transaction workload. But from the point of view of query elapsed times hot spot queries actually did much better than generic queries when run with concurrent transactions.

4.3.6 Response Time Variability

For many users, it is almost as important that transaction response times are consistent as that they are short. It is very frustrating working with a system that is subject to wide variation in the time taken to process transactions of a given type. Consequently, it is critical to examine the impact of concurrent query execution on the variability of transaction response times. Of course, the response times will increase in the presence of queries. But does the *variability* of response times also increase? If so, that will discourage organisations from running mixed OLTP and DSS workloads.

The answer is found in Figure 16 which plots the response time variability of medium and light transaction workloads standalone, and with concurrent queries. The measure being used is the coefficient of variation (CV) which is the ratio of the standard deviation of response times to the mean response time. The higher the CV, the greater the variability of response times relative to the mean response time for the test. It should also be noted that the data in Figure 16 averages a number of results from the split and unified

Figure 16: Impact of queries on response time variability

environments, taking care to ensure that comparable results were averaged for the medium and light transaction workloads. So, for example, the CV for the medium transaction workload with the 3-up queries averaged the split and unified results for generic 3-up query streams; and similarly for the CV of the light workload with 3-up queries.

What Figure 16 says is that far from causing response time variability to increase, concurrent queries actually resulted in an improvement in variability: the response time CVs declined in the presence of concurrent queries. In fact, they declined even more when there were 3 concurrent query streams than when there was only one. Organisations

may find that they are unable to accept the increase in response time caused by concurrent query execution; but the response time variability of mixed workloads should not be a major deterrent. In absolute terms, response time variability certainly went up as well; but relative to the new, higher mean response time, the standard deviation of response time in the presence of queries actually went down.

Looking back to Tables 2 and 3, there did not appear to be any significant difference in variability between the unified and split environments. Again, the column 'New/Old CV' in those tables is the ratio of the 'New' CV with concurrent queries versus the 'Old' CV of the standalone transaction workload. So, for example, the New/Old CV for the light transaction workload running with qset3 in the unified environment was 0.81, while the corresponding figure for qspl3 in the split environment was 0.80. Thus, neither the split nor the unified environment offered any advantage for response time variability.

One result that stands out is the response time variability of transactions in the light workload when run with the hot spot queries in the split environment. There the New/Old CV ratio from Table 3 is 1.33. In this case, the concurrent queries did in fact cause an increase in relative as well as absolute response time variability.

The reader will recall that the hot spot query stream also caused a marked jump in transaction response times for the light workload, as seen in Figure 13. It was argued that one possible explanation for the response time increase was that hot spot queries were causing the application code and data for light transactions to be swapped out of memory. The same phenomenon could also explain the increase in response time variability: if the

code and data was still in memory then the transactions ran relatively fast, whereas if it had to be swapped in from disk response time was higher. So if there was a significant degree of code and data swapping occurring response time variability would have been high. On the other hand, when the light transaction workload ran standalone the application code and data likely remained permanently resident so that variability was relatively low. Hence adding the hot spot query stream would have caused a significant increase in response time variability for the light workload. The medium workload did not suffer from the same increase in variability in the presence of the hot spot queries, perhaps because the transaction code and data remained in memory.

4.3.7 Results by Transaction Type

All of the results for transaction response times and variability that have been discussed to this point are based on weighted averages of the results for the different transaction types within a particular standalone or mixed workload. The four transaction types were payment inserts, payment updates, beneficiary inserts and beneficiary queries. Please see the Methodology chapter for a description of the nature of these different transaction types.

The question we wish to consider here is whether the results for the individual transaction types vary significantly from the average results for all four transaction types. Please see Tables 5 and 6 which break down transaction response time and variability results for some representative mixed workloads in the unified and split environments respectively. In each case the results were calculated based on a sample of the relevant

	Medium T/Cs Standalone (1900/250)		Medium T/Cs + generic 3-up		Light T/Cs Standalone (570/75)		Light T/Cs + qset5a	
	Resp. Time (sec)	CV	Resp. Time Increase	CV	Resp. Time (sec)	CV	Resp. Time Increase	CV
payment inserts	0.392	1.010	2.68	0.72	0.343	1.11	1.86	1.05
payment updates	0.193	1.560	2.53	1.07	0.142	1.39	1.98	1.49
beneficiary inserts	0.061	3.443	1.97	2.80	0.058	3.00	1.98	3.75
beneficiary query	0.898	0.861	2.33	0.78	0.790	0.89	1.74	0.92
workload average	0.385	1.031	2.55	0.796	0.335	1.150	1.81	1.067

Table 5: Selected performance results by transaction type for unified data

tests. For comparison, the weighted average results for the workload, taken from Tables 2 and 3, are also shown in the last row of Tables 5 and 6.

Of course, the issue here is whether concurrent queries had a varied impact on different transaction types and not vice-versa. No attempt was made to test the impact of different individual transaction types on concurrent queries. The transaction types and the volume ratios of the different types were held constant through all tests.

In Tables 5 and 6 we observe that the response time increase for beneficiary insert transactions was lower than for other transaction types in three of the four mixed

	Medium T/Cs Standalone (1520/200)		Medium T/Cs + generic 3-up		Light T/Cs Standalone (304/40)		Light T/Cs + hot spot queries	
	Resp. Time (sec)	CV	Resp. Time Increase	CV	Resp. Time (sec)	CV	Resp. Time Increase	CV
payment inserts	0.277	1.314	3.43	0.89	0.211	1.85	2.92	1.86
payment updates	0.123	1.862	3.42	1.21	0.068	2.82	3.29	2.13
beneficiary inserts	0.052	2.207	2.37	2.98	0.066	2.85	2.80	2.64
beneficiary query	0.799	0.907	2.85	0.75	0.742	0.96	3.40	2.01
workload average	0.294	1.225	3.15	0.889	0.229	1.544	3.22	2.046

Table 6: Selected performance results by transaction type for split data

workloads. That is, concurrent queries seemed to have had less of an impact on response time for beneficiary inserts than for other transaction types.

This is likely due to the fact that beneficiary inserts were the simplest and fastest of the four transaction types. They merely inserted a new row into the beneficiary table and then selected the row back. No pre-existing data needed to be read, which was not the case for the other transaction types. A beneficiary insert could be committed once the log record was written, and the log was on a different disk from the data disk. Hence, beneficiary inserts were less vulnerable to contention from concurrent queries than other transaction types, particularly for disk 1 access.

Apart from the relatively good performance of beneficiary inserts, no other transaction type exhibits a pattern of response time results which distinguished it from the other transaction types. The response time increase multiples for the other three transaction types were all reasonably similar in the presence of queries.

Turning to response time variability, we see in Tables 5 and 6 that in general the relative rankings of the coefficients of variation for different transaction types remained much the same in mixed workloads as they were when transactions were run standalone. That is, concurrent queries appear to have had no differential impact on response time variability by transaction type. For example, beneficiary inserts had the highest coefficients of variation of any transaction type in a mixed workload. But the same was true when transactions were run standalone. Since beneficiary inserts were so fast, small changes in response time resulted in significant coefficients of variation when they were run in either standalone or mixed workloads.

4.4 Mixed Workload Efficiency

The previous section discussed the impact of running mixed workloads on the performance of transactions and queries. But there is another issue that needs to be addressed when comparing standalone and mixed workloads: which approach makes most efficient use of the available system resources?

One way to address the efficiency issue is to ask the following question: Suppose there are two workloads that need to be executed, one of which consists of OLTP-style transactions while the other processes DSS queries. Would it be faster to execute the two workloads serially or in parallel (concurrently)? This is an issue that commonly arises in practice. Typically there is some advantage to be gained from executing more than one workload concurrently since any one workload usually leaves some system capacity unused. But at some point additional concurrent workloads are counter-productive since 'thrashing' behaviour begins to occur: the system spends so much time context switching that less useful work is done.

In order to measure the efficiency of mixed as compared to standalone workloads we begin by converting the elapsed time increase values in Tables 2, 3 and 4 into corresponding throughput percentages. Consider the impact of 3-up queries on the medium transaction workload in the unified environment as seen in Table 2. The elapsed time to execute the transactions in the presence of the 3-up queries increased 1.91 times versus the standalone execution time. Thus, the throughput of the medium transaction workload with the 3-up queries was reduced to $1/1.91$ or 52% of its standalone value.

Similarly, Table 4 shows us that the 3-up queries against unified data took 2.99 times as long to complete with the medium transaction workload as they did running on their own. So query throughput was reduced to $1/2.99$ or 33% of its standalone value.

We can now look at the combined efficiency of the mixed workload: medium transactions plus the 3-up queries in the unified environment. When both the transactions

and queries were running concurrently, the transaction throughput was 52% of the standalone rate, while the query throughput was only 33% of the standalone value. Thus total efficiency of the mixed workload was $52 + 33$ or 85%. (Table 7 shows 86% due to rounding.)

In order for the mixed workload efficiency to equal the standalone efficiency, the mixed workload result should be 100% or more (1.0 or greater in Table 7). Values less than 100% indicate that mixed workloads are taking longer than serial execution of the corresponding standalone workloads. So, in particular, if one has a medium transaction workload and a set of 3-up query streams to execute, it would be faster to execute them serially than concurrently, since their total mixed workload efficiency was only 0.89. One merely needs to look at the fact that transaction throughput drops almost in half (0.52) while query throughput drops to a third of its standalone value (0.33) to see that while they are running concurrently less work is getting done than if the two workloads were executed serially for the same length of time.

Note that we are not making any value judgement about the relative importance or weight of the two workloads in this calculation. It is immaterial to us whether the queries were more or less important than the transactions. In effect, we are adopting the viewpoint of the system administrator who has been informed that she has two workloads to run and it is up to her to figure out how best to run them. Not running one of them is not an option.

Of course, it may not help to be told that a mixed workload is less efficient than corresponding serial workloads. If the queries must be run then in many environments it will simply not be feasible to halt transaction execution while the queries are processed in order to optimise resource utilisation efficiency by means of serial execution. Nonetheless, when evaluating mixed as opposed to standalone workloads, it is important to understand the implications for system efficiency. For example, if mixed workloads are very inefficient then it will be necessary to purchase more processing capability than corresponding standalone workloads would require.

The complete system efficiency results for all of the mixed workloads tested are presented in Table 7. For each mixed workload, Table 7 shows the throughputs of the transactions and queries expressed as a fraction of their standalone workload. These values are then summed to yield the mixed workload's total throughput. Again, a total throughput of more than 1.00 indicates that the mixed workload is running more efficiently than separate, serial execution of the transactions and queries.

The first observation we wish to make regarding the efficiency results is that on the whole the efficiency of mixed workloads is comparable to or better than serial execution of the same workloads. With the exception of mixed workloads involving 3-up query streams, the total throughput is always close to or greater than one. Even when 3-up query streams are included, the worst result is 0.86 for the medium transaction workload plus the generic 3-up query stream in either environment. Many organisations may deem

	T/C throughput	Query Throughput	Total throughput
Unified Results			
Medium T/Cs + 3-up queries	0.52	0.33	0.86
Medium T/Cs + qset1a	0.71	0.32	1.04
Medium T/Cs + qset3	0.67	0.29	0.96
Medium T/Cs + qset5a	0.75	0.29	1.04
Light T/Cs + 3-up queries	0.54	0.41	0.95
Light T/Cs + qset1a	0.71	0.45	1.16
Light T/Cs + qset3	0.71	0.40	1.11
Light T/Cs + qset5a	0.78	0.42	1.19
Split Results			
Medium T/Cs + generic 3-up	0.47	0.39	0.86
Medium T/Cs + hot spot 3-up	0.44	0.70	1.14
Medium T/Cs + qspl1a	0.64	0.40	1.04
Medium T/Cs + qspl3	0.63	0.33	0.96
Medium T/Cs + qspl5a	0.72	0.36	1.08
Medium T/Cs + hot spot queries	0.69	0.72	1.41
Light T/Cs + generic 3-up	0.44	0.48	0.93
Light T/Cs + hot spot 3-up	0.51	0.84	1.35
Light T/Cs + qspl1a	0.66	0.52	1.18
Light T/Cs + qspl3	0.69	0.44	1.13
Light T/Cs + qspl5a	0.73	0.48	1.21
Light T/Cs + hot spot queries	0.62	0.82	1.44

Table 7: Mixed workload efficiency

an efficiency loss of 14% a reasonable price to pay for the ability to execute mixed workloads rather than incur the costs of separately executed workloads.

Another observation that can be made regarding the efficiency results in Table 7 is that mixed workloads with a single query stream were more efficient than those running 3-up queries. It is conceivable that with three concurrent query streams plus a transaction workload executing some thrashing behaviour was beginning to occur causing efficiency to drop. If only a single query stream was running there was less context switching taking place so that overall efficiency benefited.

It is noteworthy that the total throughput results for mixed workloads in which a single stream of short queries was being run (qset3 or qspl3) had total throughput results which were consistently worse than those of longer queries. In our analysis of the sar results we argued that long queries and short transactions were potentially a good fit for a mixed workload since transactions were I/O intensive whereas long queries made relatively more use of the CPU. Short queries, on the other hand, were also I/O intensive. Not surprisingly, then, we find that transactions ran more efficiently with long queries than with short ones.

Another pattern is that from an efficiency point of view the light transaction workload consistently outperformed the medium transaction workload when run with concurrent queries. For example, the medium transactions plus qset5a had a total throughput value of 1.04, while the throughput of the light transactions plus qset5a was 1.19.

The reason the light transaction workloads enjoyed an efficiency advantage is that on their own they left more spare system capacity. For example, in the unified

environment, the medium transaction workload consumed 90% of disk 1's capacity when run standalone, while the light transaction workload only used 72% of disk 1. So there was more opportunity to gain in overall system efficiency by running queries concurrently with the light workload than with the medium workload.

The implication is that if the transaction workload by itself uses all or nearly all of some critical resource, typically disk I/O, then adding a new query workload to run concurrently with the transactions is unlikely to be more efficient than running the workloads serially. However, if the transactions leave spare capacity, then running one or two concurrent query streams will almost certainly be more efficient than serial execution.

Looking at the throughput data for mixed workloads that included the hot spot queries, we see that they yielded the best results of all. The efficiency of mixed workloads with the hot spot queries was quite high, even when the hot spot queries only constituted one of three concurrent query streams. Here we have further evidence that the short, CPU-intensive hot spot queries do very well in a mixed workload environment. Clearly, the hot spot queries were making good use of the CPU while the transactions were waiting on disk I/O.

Another pattern in the efficiency results is that in general the query throughput suffered more than the transaction throughput for a given mixed workload. For example, the medium transaction workload ran at 63% of its standalone throughput rate when executed concurrently with the short queries, `qspl3`, in the split environment. But `qspl3` was reduced to 33% of its standalone throughput. This again suggests that the operating

system or DBMS priority scheme was favouring the shorter transactions over the longer queries. The major exception to the rule was once again the hot spot queries: their throughput declined less than the transaction throughput did in mixed workloads. Perhaps the operating system's priority manager allows transactions that require relatively little disk I/O to proceed at high priority.

As a final observation on the mixed workload efficiency results in Table 7 we note that the total throughput for similar workloads in the unified and split environments were identical or very nearly so. For example, the total throughput figure for the medium transaction workload plus qset1a in the unified environment was 1.04. It was also 1.04 for the medium transaction workload with qspl1a in the split environment. It can be concluded that from an efficiency point of view there is nothing to choose between unified and split data.

4.5 Split versus Unified Data

Before reviewing the split versus unified data performance results, it needs to be asked whether such comparisons are legitimate given the methodology employed. For example, we claim that the transaction response times are faster against the split tables than the unified tables. Are such comparisons meaningful?

We first note that the data structures in the split and unified databases were identical apart from the fact that the payment tables were divided in two in the split environment based on payment date. As shown in Table 1, The total number of rows in

the payment tables in the two databases was exactly the same. The data in the other tables was identical.

Secondly, the SQL executed in the two databases was very nearly identical. The only differences were already noted in Chapter 3:

- with the exception of the beneficiary query, the transactions in the split environment were redirected against the current payment tables.
- in the split environment the beneficiary query was executed by first running one query against the current data and the same query against the historical data and then taking the union of their results.
- the queries in the split environment had their date criteria modified as required to ensure that they retrieved data for the same number of days as the corresponding queries in the unified environment.

Thus, if the same transactions against similar data structures performed faster in the split than in the unified environment, it can only be because of the split itself. Nothing else of consequence changed.

With respect to query performance there was about 6% less data in the Payment_Header and Payment_Detail tables in the split database than in the unified database. So one would expect the same queries against the two databases to perform 6% faster in the split environment. Apart from this expected difference, any variation in query

performance between the two environments can be attributed to having split or unified tables.

When comparing mixed workload behaviour in the two environments, one is typically looking at how the standalone performance of some workload was affected by other concurrent workloads. If, say, queries caused transaction response time to increase $x\%$ in the unified environment, it is fair to compare the corresponding increase in the split environment since the data structures and volumes were the same, the transactions and queries were the same, and, finally, the same query and transaction mixes were executed in both environments.

What then do the experimental results say about the relative merits of the split and unified database models? Here is a summary of the main split versus unified findings:

- the sar data shows that the resource utilisation of corresponding mixed workloads in the split and unified environments was very similar.
 - absolute transaction response times were better with split than unified data, both with and without concurrent queries. However, in relative terms, the response times were subject to greater degradation in the split environment.
 - even allowing for the difference in data volumes, queries ran faster in the split than the unified environment both with and without concurrent transactions.
- In this case, the split environment results were better both in terms of absolute query execution times and the relative increase caused by concurrent transactions.

- the response time variability of transactions was much the same in the split and the unified environments.
- hot spot queries did very well in the split environment, although they did cause the response time of transactions in the light workload to increase significantly. The DBMS optimiser did not take advantage of the date restriction when the data was unified.
- the efficiency of mixed workloads was virtually identical for split and unified data, except that the hot spot stream in the split environment was very efficient.

In our view, these results are inconclusive in that they do not indicate a clear advantage for a split or unified data approach when running mixed query and transaction workloads. The split environment does appear to have some performance advantages for both queries and transactions when they are run concurrently. However, splitting the data into separate tables does have its drawbacks. For example, the current data will need to be periodically migrated to the historical tables. Will the split data performance advantage, which in general was a modest one, be important enough to organisations to justify the costs implied by split tables?

While hot spot queries did very well running against the current data in the split environment, it is arguable that they did too well because they caused a large increase in the transaction response time for the light workload. Assuming a DBMS supports intervention in its priority management scheme, database administrators may elect to lower

the priority given to hot spot queries in order to speed up the transactions, in which case the hot spot queries would not perform quite so well. Moreover, it is reasonable to expect most DBMS optimisers to become sophisticated enough to make good use of data qualifications that significantly restrict the selected data, in which case hot spot queries can be expected to run efficiently in the unified environment as well.

Finally, we note that it is difficult to achieve 'purity' in creating a split data environment. While some of the data may lend itself naturally to being current or historical, much of the data is effectively independent of time and so must be used both by current transactions and historical queries. It is the rare organisation that will be able to completely restrict transactions to current data and queries to historical data without any overlap or data replication.

In conclusion, organisations that adopt the split data approach will likely do so only because there are additional motivations apart from the performance advantage. In particular, it may be a hard and fast rule for some applications that no inserts, deletes or updates take place against data older than some date. In that case, the organisation may wish to segregate the static and volatile data to ensure that static data remains untouched, or again to facilitate the administration of security. In any case, the performance advantage of split data for mixed query and transaction workloads is unlikely, by itself, to be enough to convince many organisations to implement a split database model.

4.6 Concluding Comments on the Results

On the whole, the test results displayed a high degree of consistency. Anomalous results were rare. Sar results for mixed workloads in the unified environment were consistent with those in the split environment. The same was true for the total throughput or efficiency of mixed workloads. Among individual transaction types, the beneficiary query transaction consistently recorded a low coefficient of variation compared to the other transaction types. 3-up query streams consistently resulted in greater transaction response time increases and better response time variability values than single query streams.

The fact that the results were reasonably consistent is an argument in favour of their validity. If there had been problems with the test methodology or execution then the results would likely have been more erratic.

As well as being consistent, the results appear to be reasonable in that one can usually hypothesise a plausible explanation for them. While it is difficult to predict how complex DBMS systems will behave in particular circumstances, none of the results observed seem completely implausible or mysterious. The DBMS did not exhibit 'irrational' behaviour.

That is, given that the test results were consistent and logical we can claim to have characterised how this particular DBMS behaves when processing mixed workloads and are therefore in a position to predict how it will perform when running mixed workloads in the future. In other words, mixed workloads have become manageable. They would not

be manageable if they resulted in inconsistent and/or illogical behaviour. Since this is not the case, managing mixed workloads is like managing any other workload: addressing such issues as scheduling, priority management, capacity planning, service level definition, and so on.

A separate question that has not been addressed thus far is the following: Suppose an organisation had an OLTP application with specific throughput requirements. Then a concurrent query workload is added to the system. Is it possible to maintain transaction throughput at the standalone rate?

It has been assumed throughout that for a given transaction workload average sleep time between transactions remains constant whether or not there are concurrent queries. Given this assumption and fixed system resources then transaction response time and throughput must suffer in the presence of queries. But suppose we allowed sleep times to shorten as response times lengthened in the presence of queries. Could the shorter sleep times permit throughput to be maintained in spite of the longer average response time?

It should first be noted that for many if not most OLTP applications it will not be feasible to shorten the average sleep time as response time increases. However long it takes the DBMS to process a transaction, it will still take human users the same length of time on average to prepare the input for the next transaction from the user interface.

Secondly, the heavy, medium and light transaction workloads that were used in this research were defined in terms of the average sleep time between transactions. The

shorter the average sleep time the heavier the workload. Thus, shortening the average sleep time in an attempt to maintain transaction throughput is equivalent, for example, to converting a light transaction workload into a medium transaction workload.

Consider the light transaction workload in the unified environment. On their own, the transactions in the light workload executed at the rate of 4.50 transactions per second (tps). This is calculated by multiplying the number of transactions executed, say $(760)(2) + (100)(14) = 2920$, and dividing by the overall execution time of 649 seconds as reported in *Appendix B: Performance Results* (Unified data: light transaction workload, 760/100 repetitions). When run in mixed workload with `qset1a` the transaction throughput slowed to 3.20 tps. However, the throughput of the medium transaction workload when run with `qset1a` was 4.33 tps, which is close to the standalone rate of the light workload. This implies that if users cut their average sleep times in half, which was the difference between the average sleep times of the medium and light workloads, then most of the transaction throughput loss caused by the concurrent execution of `qset1a` could be recovered.

On the other hand, when we look at the light workload in the unified environment together with the 3-up query stream we find that throughput declined to 2.48 tps. The medium workload only achieved a throughput of 3.17 tps when executed with the 3-up query stream. This is well below the 4.50 tps for the standalone light workload. Thus, it is not feasible to maintain the transaction throughput rate of the standalone light workload when running 3 concurrent query streams by reducing the average sleep time.

Hence, it may be possible in some circumstances to maintain transaction throughput rates when concurrent queries are added by reducing average sleep times. The lighter the transaction workload is to begin with in relation to system capacity and the fewer the number of concurrent query streams the better the prospects will be for maintaining transaction throughput by means of reduced sleep times.

Ideally, one would also like to maintain fast transaction response times in the presence of concurrent queries. The test results indicate that two key factors for determining the performance of transactions in mixed workloads are disk I/O capacity and priority management. If it is essential to ensure fast transaction response times in the presence of queries then it will be necessary to control the relative priorities accorded to transaction and query workloads by the DBMS and/or the operating system. In particular, queries should run at a lower priority than transactions from the moment they start executing rather than just after they have been running for some time.

But while it may be acceptable to lower the priority of queries relative to transactions, queries still need to receive a reasonable level of service. In this regard, disk I/O appears to be a critical resource. The transaction workload on its own must leave a significant percentage of the system disk I/O capacity unused so that lower priority queries will still execute in a reasonable length of time. We have also found that light transaction workloads yielded better efficiency results when run with queries than medium transaction workloads, which again means that the transactions on their own should not use all or nearly all of the critical system resources.

Hence if the transaction workload is not onerous, and there are mechanisms available to manage priorities, it may be possible to run mixed workloads in which not only transaction throughput but response time as well is maintained when concurrent queries are added. Even if this ideal cannot be attained, the test results indicate that mixed workloads perform in a consistent and therefore manageable manner. It will thus often be feasible to scale the processing platform and/or manage the transaction and query workload volumes such that mixed workloads provide acceptable performance.

RELATED WORK

The author is not aware of any experimental performance studies of mixed workloads in the literature. However, a number of papers have addressed the issue of mixed query and transaction workloads, and some of them include simulation studies which attempt to predict how their techniques would perform in practice.

5.1 Mixed Workload Proposals

The mixed workload proposals which have been made in the literature fall into two major categories. One is ‘transient versioning’ which requires the DBMS to maintain prior versions of updated records for as long as any executing query may need them. By accessing the prior versions of modified records, a query can run against a consistent view of the data even if that view is slightly outdated. The prior versions are merely ‘transient’, however, because as soon as there are no more queries that might need them, they may be deleted. Hence the transient record versions are distinct from a permanent database log used for recovery purposes.

The second category of mixed workload proposals may be called ‘enhanced serialisation’. In most of today’s DBMSs, a long-running query will lock all of the data it reads until it completes, thus blocking any concurrent transactions from executing against

the same data for the duration of the query. In the enhanced serialisation schemes, transactions are permitted to execute as long as either all of the data they modify has been completely read by the query, or none of it has been read by the query. That is, if the transaction can be serialised before or after the query in virtue of the data it needs and that the query has already read, then the transaction can execute concurrently with the query.

Most of this chapter is a review of proposals in the literature that fall into the above two categories. The first three papers we examine, [Mohan92], [Wu93], and [Bober92], describe transient versioning schemes. We will then discuss some enhanced serialisability techniques. The chapter will conclude with some comments on other mixed workload mechanisms that have been developed or suggested.

5.2 Stable Version Scheme

The first versioning approach that we discuss is described by C. Mohan, H. Pirahesh and R. Lorie in [Mohan92], here referred to as the ‘stable version scheme’. Their goal is to enable long-running, read-only queries to execute against stable versions or ‘snapshots’ of the data while updates execute against the current data.

Minimally, the stable version scheme requires the DBMS to support at least three different potential versions of any database record:

- the ‘stable’ version, which is the version used by long-running queries,
- the non-stable version, which represents data that has been updated and committed since the stable version was created, and

- the uncommitted version, which is data that has been inserted, modified or deleted by a transaction that has not yet committed. The only transaction that can access the uncommitted version is the one that created it.

Since long-running queries are restricted to viewing the stable version of the data, they do not see any changes that have been made since the stable version was created. Ideally, a query would see the data in the database as it was at the moment the query began executing. However, that is rarely the case. Instead, a query will see the data as it was at the time the last stable version (or 'snapshot') was created. If there is only one stable version and it has been in existence for some time then the data a query sees could be minutes or even hours old. No record version can be deleted from the database as long as some query might potentially need to read it.

In the stable version scheme, a version record is associated with the primary key of every record in the database. So when a search is done on the primary index for a particular key value, the corresponding version record is found in the index leaf page. The version record tracks the identifiers of the transactions that created each of the data record's extant versions. It also contains pointers to the actual base records for the different versions as illustrated in Figure 17. In the figures and text, 'T/C' is used as an abbreviation for 'transaction', whether committed or uncommitted, and 'id' is an abbreviation for 'identifier'.

If only a single stable version is being maintained, then in addition to the version records the DBMS maintains two transaction lists: one list tracks the uncommitted

Figure 17: Version record for a primary key value

transactions, while the other, which is called the 'non-stable' list, records the transactions which have committed since the stable version was created. If multiple stable versions are being used, then there would be multiple non-stable lists of transactions committed since each version. The transaction lists are used in conjunction with the version records to enable queries and transactions to identify the version of the row that they should access.

In Figure 18, the query will see the data as it was when the stable version was created. The last committed transaction to update a particular record prior to the snapshot creation was T/C id 3, which will therefore be in neither the uncommitted nor the non-stable transaction lists. The same record has been modified by T/C id 1 which has not yet committed. So T/C id 1 will appear in the uncommitted list and the query will not use the version it created. T/C id 2 also modified the same record and has committed. But even though T/C id 2 committed prior to the query's start time, the query will discover

Figure 18: An example of timing relationships in the stable version scheme

that T/C id 2 is in the non-stable list and so will skip the T/C id 2 version of the record as well.

When a query needs to read the row, it will first examine the associated version record in order to find the stable version. The query will begin with the most recent entry in the version record, and check whether the transaction id is in the uncommitted or non-stable transaction lists. If not, then the stable version has been found, and the query will use the associated pointer to retrieve the row. Otherwise, it will look at the transaction id of the next most recent entry in the version record, and check whether it is in the non-stable transaction list. If not, then the corresponding pointer is followed to the data record which will be the stable version. If the two most recent entries in the version record were either uncommitted or non-stable, and if only a single stable version of the data is being maintained, then the third and oldest entry in the version record must be for the stable version. In this case the query will follow the pointer in the third entry of the version record to the stable version of the row.

In the stable version scheme, a version 'switch' occasionally takes place when no queries are running against the stable version. That is, a new logical snapshot of the data is created against which subsequent queries can run. Hence the data that queries see is kept reasonably up to date. All that is required to create a new stable version is to empty the current non-stable list of transactions. When a query next tests whether a version record entry corresponds to the stable version, it will only fail if the transaction id is in the uncommitted list. Of course, over time new transactions will commit and be moved from the uncommitted to the non-stable list.

Unless query volumes are very low and queries are always short, maintaining only one stable version of the data could cause serious problems. If any queries are running against the stable version a version switch cannot take place since the logical snapshot needs to be maintained until the queries complete. As a result, in a busy environment there could be very long periods between version switches so that queries would run against unacceptably old data. Similarly, a single query which runs for hours would prevent any version switches during its execution.

To circumvent this problem, the scheme is capable of supporting more than one stable version. In principle, a new stable version could be created whenever a long-running query was started. But the overheads for tracking a large number of different versions would become prohibitive, and so the maximum number of stable versions would typically be less than ten.

Nonetheless, the fact that more than one stable version could be maintained improves the scheme's flexibility. New queries would use the most recent stable version, and old versions would be removed when no queries were executing against them. Since new queries would never be started against old versions, the old versions would eventually become inactive and could be deleted. A given record would still have at most one uncommitted and one committed but non-stable copy, in addition to its stable versions.

In an environment with multiple stable versions, each would typically have queries executing against it. A given query must clearly use one and only one stable version during its lifetime. There would be a separate list of non-stable transactions for each version, as well as the list of currently uncommitted transactions. A query would use the correct lists for the stable version it was running against in order to identify the appropriate data rows to access via the version record.

[Mohan92] extends the scheme to accommodate secondary indexes as well. Secondary indexes consist of a set of index value-primary key pairs. If an index value-primary key pair appears in the secondary index then at least one of the extant versions of the base row corresponding to the primary key contains the target secondary index value. However, while at least one version of the base row contains the desired index value, other versions of the same row could have a different value for the same attribute since it might have been modified. The question then is how to find the version of the row that has the target secondary index value.

This is done by maintaining a version record in the secondary index itself for each index value-primary key pair. As in a primary index version record, a secondary index version record would contain the ids of transactions that created the uncommitted, non-stable, and stable versions of the base row. But while the version record in the primary index contains pointers to data rows, a secondary index version record instead contains flags indicating whether or not the particular index value existed in each of the base row's versions. If an appropriate version has the desired secondary index value, then the correct base row version would be retrieved via the primary index.

Suppose, for example, that an update transaction needs the latest committed rows with a particular secondary index value. Suppose further that the secondary index identifies the primary key of a row that contains the desired index value in one or more of its versions. Then the update transaction would examine the associated secondary index version record for the index value-primary key pair. The update can immediately skip any version of the row that does not contain the target secondary index value according to the version record flags.

The version record flags could indicate that the most recent version of the row contains the desired secondary index value. But if the id of the transaction that created the most recent version is in the uncommitted list then that particular version has not yet been committed and so cannot be used by the update. If, on the other hand, the transaction id is found in the committed but not stable list, the update will retrieve the row via the primary index for processing. Another possibility is that the secondary index version

record indicates that the most recent committed version of the row does not contain the target secondary index value, so that the update can skip the row altogether.

In our view, the stable version scheme is theoretically workable. Unfortunately, nothing is said in [Mohan92] about the expected performance of this transient versioning approach. What would be the impact of the stable version scheme on query and update performance as opposed to, say, running the two workloads on separate systems?

At the very least, we know that the stable version scheme requires all access to the database to take place via the primary index. Both updates and queries need to examine the primary index version records in order to locate the row versions they need.

In particular, queries will no longer be able to do a straightforward full table scan of the base data blocks. Those data blocks will potentially contain multiple versions of the same record. The only way the query can determine the appropriate records to read is by referencing the version records in the primary index. As a result, queries which would previously have used a full table scan are bound to execute more slowly in a transient-versioning DBMS based on the stable version scheme.

It is also significant that secondary index entries no longer point directly to the base data row or block containing the row that has the secondary index value. Instead, the secondary index supplies a primary key value, as well as flags indicating which versions of the base row with that primary key contain the secondary index value. The query or transaction must then do a lookup in the primary index to find the primary key and its corresponding version record. Only then will it obtain a pointer to the desired base

row version. Thus, a level of indirection has been introduced into secondary index access which will clearly cause performance degradation versus traditional databases that do not support versioning.

While the stable version scheme introduces significant processing overhead, it also has important benefits for mixed queries and transactions. Above all, it permits query and transaction workloads to execute concurrently without blocking each other, and at the same time guarantees that queries will see a consistent view of the data. Moreover, in most business environments the stable version scheme will reduce storage costs versus maintaining a complete, separate copy of all the data in a data warehouse. The reason is that only a small fraction of the rows in the database will be inserted, updated or deleted by transactions between version switches, and so there will only be a single copy of most rows.

5.3 Dynamic Finite Versioning

Kun-Lung Wu, Philip Yu and Ming-Syan Chen describe a transient versioning scheme in [Wu93] which is conceptually very similar to the stable version approach. They call their technique Dynamic Finite Versioning (DFV).

The DFV term for a stable version is a 'snapshot'. As in the stable version scheme, any finite number of snapshots can theoretically be supported. Once again the intention is to permit read-only queries to execute against the last snapshot created before the query began without placing any locks. Transactions that do updates and deletes

access the current data through the traditional locking mechanism. When a particular snapshot no longer has any queries executing against it, it can be 'advanced', which is to say discarded and replaced with a new, current snapshot.

The differences between the stable version and DFV approaches are at the implementation level. Whereas the stable version scheme used record-level versioning, the DFV technique uses page-level versioning. That is, a change to a single record on a page will result in a new page version being stored if it will potentially be needed by some query.

While version information was kept in the primary index in the stable version scheme, it is placed with the data pages in DFV. The version data is also somewhat different in DFV: each page is stamped with the time it was created (not necessarily committed) and the id of the transaction that created it. While the stable version scheme maintained separate lists of uncommitted and 'non-stable' transactions, DFV only tracks the uncommitted ('active') transactions, both current and as of the time that each snapshot was created.

As mentioned above, each read-only query using DFV will run against the last snapshot created before the query began. In order to find the appropriate version of a page that it needs to read, the query will first examine the timestamps on each page. If the timestamp is greater than the time of the snapshot the query is using, then the query will proceed to the next earlier version of the page. If the page was created before the snapshot, then the active transaction list for the snapshot is checked to see whether the

transaction that created the page is in the list. If so, the page was merely 'created' and not yet uncommitted at the time the snapshot was taken. So the query will need to read the next earlier version of the page, which would have been both created and committed before the snapshot.

Please see Figure 19 which is similar to Figure 18. A0 through A4 are subsequent versions of the same page in time. Only the versions marked with a circle, A0, A1 and A4, actually exist. Two snapshots are being maintained. A0 is the oldest snapshot and will be read by query Q0. A1 is the most recent snapshot, and will be used by query Q1 even though a number of other versions of the page have been created since A1 and before Q1 started running. A4 is the current committed version of the page, and will be used in the next snapshot if the page is not again modified and committed before the next snapshot is created. There could also be a fourth version of the page in existence if an active transaction had created a new but uncommitted copy.

As in the stable version scheme, creating a new snapshot is a straightforward process in DFV. One simply needs to track each snapshot's creation time, a list of transactions that were active when the snapshot was created, and a list of queries using the snapshot. This last list is empty when the snapshot is first created, and must become empty again before the snapshot can be discarded.

The results of a performance analysis of DFV are found in [Merchant96]. The analysis is based upon a simulation study. The two major metrics used were query obsolescence and storage requirements. Query obsolescence measures the age of the data

Figure 19: An example of timing relationships with Dynamic Finite Versioning

against which queries execute. The objective is to use snapshots which are as current as possible and so minimise query obsolescence. Storage requirements depend upon the number of different versions of each page that need to be maintained; this number should also be minimised. The performance study did not attempt to determine the cost of transient versioning itself in terms of reduced throughput or increased response time versus, say, executing queries and transactions against the same data at the same time with no query read locks.

The study did find that when a number of snapshots are being maintained, it is important to be able to 'advance' *any* snapshot which no longer has any queries running against it as opposed to just the oldest snapshot. If only the oldest snapshot can be advanced, then it may block more recent snapshots which are no longer being used by any queries from advancing. The cost of allowing only the oldest snapshot to advance is

increased query obsolescence and a larger average number of versions per page (greater storage requirements).

As could be anticipated, the performance analysis also found that there is a trade-off between reduced query obsolescence and increased storage requirements. The more snapshots there are, the more recent the latest snapshot is likely to be, and so queries will run against reasonably current data. But this comes at the cost of maintaining a larger average number of versions per page.

[Wu93] argues that page-level as opposed to record-level versioning has its advantages. The amount of version information that needs to be maintained is reduced, and a query which is scanning a table can find the versions it needs at the page level rather than having to examine each record's version data. The fact that DFV places the version information in the page rather than in the primary index is an immediate advantage versus the stable version scheme in that it permits full table scans of the base data.

However, page-level versioning also has its drawbacks. If a single record is changed on a page that contains, say, 100 records, then a new version is created for all 100 records and not just the one that was changed. This obviously increases the storage costs associated with transient versioning: one is maintaining many versions of records which have not in fact changed. When a query needs one of the other 99 records on the page, it will have to search for the right page for its snapshot even though the desired record is the same in all the existing page versions. There is a need for further research to

determine whether transient versioning schemes are best implemented at the page or at the record level.

Another difficulty with DFV is that [Wu93] says nothing about indexes. Primary indexes could presumably point to the most recent committed page, which would then start a pointer chain to older pages. But it is not at all clear how secondary indexes would be managed. If a particular secondary index value exists in one page version but not another, are multiple secondary index entries created which point to the different page versions? In this case, if an update follows a secondary index entry to a page, how is the update to know whether or not the page is the latest committed version? Indexing issues are not trivial and need to be examined in some detail in order to ensure that DFV is a viable proposal.

5.4 Version per Query Scheme

The third and last of the transient versioning schemes that we review was developed by Paul M. Bober and Michael J. Carey. They describe their work, which we will refer to as the 'version per query scheme' in [Bober92] and [Bober97].

Like the transient versioning approaches discussed above, the version per query scheme maintains a finite number of row versions to provide queries with a consistent view of the data while update activity against the same data continues. However, in the earlier schemes the DBMS created a fixed number of snapshots which were shared by different queries. That is, the snapshot times were defined by the DBMS based on

algorithms that determined when new snapshots could be created. In the version per query scheme, as its name implies, each query effectively defines its own snapshot. There are no 'pre-set' snapshots which queries are obliged to use. Instead, the snapshot used by a given query is defined by the start time of the query. As a result, every query uses all of the committed data as of the time it begins. From a user point of view, it is clearly preferable for queries to use the most recent data available rather than having to go back in time to some previously created snapshot.

As in the other transient versioning schemes, read-only queries do not place any locks. Update transactions access records through the normal locking scheme.

In [Bober92] and [Bober97] Bober and Carey provide the most detailed and complete description of a transient versioning scheme of the three that are discussed in this chapter. In particular, they pay careful attention to where previous versions of modified records are stored, and to possible indexing schemes for multi-version data. Their approach is based on a multi-versioning scheme that was previously developed and implemented by Computer Corporation of America, and later used in Digital Equipment Corporation's Rdb relational database system. Unfortunately, neither the CCA nor Rdb products are still widely used today, so no information is available as to how effective their multi-versioning implementations were.

The version per query scheme uses record-level versioning. A key feature of the technique is 'on-page caching'; that is, to the extent possible prior versions of records are kept on the same data page as the latest version of the record. This means that data pages

must be created with some free space so that a number of older record versions may be cached on them. Of course, there will always be some overflow: there will not always be space on the data page to store a previous record version. In this case, the older versions are placed in a separate storage area on disk known as the ‘version pool’. The version pool is a common store used for any overflow version records that need to be retained. It is managed as a circular buffer. Over time, garbage collection processes free up space in both the version pool and in the on-page caches when prior versions of records will no longer be needed by any executing query.

The use of on-page caching is an important optimisation. If a query determines that it cannot use the current version of some record, but the prior version it needs is on the same page as the current version, then no additional I/O cost is incurred. If it is necessary to go to the version pool to retrieve the required version then at least one more logical I/O will be required. Consequently, performance is improved if large on-page caches are available, at the cost of increased storage space.

[Bober97] documents the results of simulation studies of a variety of potential indexing schemes, including the placement of version data in the primary index as proposed in [Mohan92]. But based on their simulation results, Bober and Carey advocate putting version information on the data page in a structure called the ‘Version Selection Table’ (VST). Please see Figure 20. Both primary and secondary index entries point to the Version Selection Table in the data page that contains the desired record. The query

Figure 20: Bober and Carey's version per query scheme

or update will then use the information in the VST to determine which record version to access.

For each record, the VST maintains the following data:

- the delete time stamp (DTS) of the record. When a record is deleted it may still be needed by some query that began before the delete was committed. So deleted records must be retained until no query will need them. If the delete time stamp is infinite then the record has not been deleted.

- a commit timestamp (CTS) and pointer for each committed version of the record that still exists. The pointer may be to a version stored in the on-page cache or in the version pool.

In order to determine which version of a record it needs, a query follows a simple rule:

$$\text{record commit time} \leq \text{query start time} < \text{record delete time}$$

As seen in Figure 20, the primary index entry for a particular key value will also contain delete and commit timestamps so that queries and updates can immediately determine whether a record version with the key value might exist in the appropriate time range. If the delete timestamp has been set to some non-infinite value, for example, then an update transaction will know that no current record exists with the given key. The commit timestamp is needed in the primary index entry since a query may have started before the corresponding record was created.

A single row may have multiple secondary index entries for a particular secondary index attribute, as seen again in Figure 20. From times 25 to 50, versions of the base record contained secondary index value b1. From time 50 to the present the record's versions contain value b2. Thus a query that started at time 45 and needs value b2 will not access the base record at all since the secondary index information indicates that at time 45 the value in the base record was b1.

All primary and secondary indexes point to the Version Selection Table in the data page containing the current version of the record. The VST is then used to locate the version needed by the query or update.

[Bober92] reports the results of simulation studies designed to evaluate the likely performance of the version per query scheme. In some of their tests Bober and Carey compared the performance of transient versioning to executing queries in read-uncommitted ('dirty read') mode at the same time as updates. Of course, queries executed at the read-uncommitted isolation level will not see consistent data. Bober and Carey's simulations indicate that if the on-page cache for previous versions is sufficiently large the performance degradation caused by transient versioning versus read-uncommitted queries is not excessive (about 10 to 20%) and is reasonably constant over a range of workloads. Performance begins to deteriorate significantly when version pool use increases.

These performance results suggest that relatively large data blocks should be used in which it will be possible to maintain a sizeable on-page cache. Of course, the free space reserved in the data blocks for the on-page cache will often be unused, implying wasted space and increased storage costs. Moreover, while large data blocks improve query performance they typically reduce update performance since updates usually only need a small number of the records in a block.

These costs, however, need to be compared to the alternatives. One option is to maintain two completely independent copies of the data, one for update and the other for

queries, which obviously has significant costs and means that queries cannot execute against current data. Another option is to execute queries in read uncommitted mode, but that is not always acceptable depending upon the semantics of the query. In an environment where queries must run against the most recent, consistent data available, the version per query scheme may be worth the cost.

5.5 Comparison of the Transient Versioning Schemes

The basic idea in all three of the foregoing transient versioning schemes is the same: the DBMS automatically maintains row versions so that queries can access a consistent, even if slightly outdated, view of the data while transactions concurrently modify the most recent data. Versioning is 'transient' because the DBMS also automatically deletes old versions of the rows when there is no longer any query executing that might need them. Users can thus submit queries as if no transactions were running; and transactions are not blocked by lengthy queries.

From a functional point of view the version per query scheme from Bober and Carey has the advantage of creating a new logical snapshot with every query. In the other two schemes queries are restricted to one of a small number of stable versions created at some previous point in time. Thus the version per query approach promises the most recent possible consistent data for each new query.

With respect to implementation specifics, the version per query technique is also the most carefully developed. [Mohan92] does not explain where versions of the same

data record would be stored in the stable version scheme. Would they be on the same page as the most recent version, in a separate version pool, or in some other location? [Wu93] says nothing about secondary index structures for DFV, which is a key issue for database implementation and performance.

It is clear that the storage cost associated with the DFV approach will be high since if a single row on a page is modified then a new version of the entire page is created. In the version per query scheme storage overhead results from the free space left on each page for new versions of records. But at least the version per query scheme creates new data versions at the record and not the page level which reduces storage costs versus DFV. Since [Mohan92] does not discuss how record versions would be stored we cannot say what the storage cost of the stable version technique would be.

The simulations in [Bober92] are encouraging with respect to the potential performance of the version per query scheme compared to the concurrent execution of read-uncommitted queries and transactions. Although no simulations results were reported for the other two schemes, it is clear that the version record techniques used in the stable version approach will exact a significant performance penalty. For example, the fact that a secondary index lookup must be followed by a primary index lookup rather than a direct access to a base data row will have a performance cost. Since we do not know how secondary indexes would function in the DFV scheme we cannot meaningfully predict its performance.

Transient versioning is theoretically attractive in that it permits the DBMS to transparently support mixed query and transaction workloads. Its drawbacks are that it is complex and will impose storage and performance costs. The key issue is whether or not there will be enough demand for the technology to warrant its commercial development.

Many users will be satisfied with concurrent execution of read-uncommitted queries and transactions. Those that must execute queries against stable data will often need to specify the point in time at which the query executes, such as month end. These users will not want the data as at the time the query started. The ability to execute many queries against the same *user-defined* snapshot will be critical. Hence it remains to be seen how large a market there will be for transient versioning techniques in which the definition of the snapshots is controlled by the DBMS itself.

5.6 Enhanced Serialisability

Apart from transient versioning, the other major mixed workload strategy that is discussed in the literature is based on the observation that it is often possible to serialise short transactions with a long-running query even if they execute concurrently. Hence the term ‘enhanced serialisability’. The key is to ensure that either all of the data or none of the data used by the transaction has been read by the query. See references [Pu88], [Ammann95] and [Salem94].

In [Pu88], a two-colour scheme is described which can be used to determine whether a transaction can be serialised before or after a query, which is also termed a

‘global read’. When a query starts it colours all of the data it is going to read white. As it proceeds it changes the colour of the data that it has read to black. So when a transaction executes it verifies whether the records it modifies are all white or all black. In either case, the transaction may proceed: if the data is all white then the transaction is effectively running before the query; if the data is all black then the transaction is serialised after the query.

The problematic case, of course, is when the transaction needs to modify both black and white data. In that case it is neither before nor after the query. Various strategies are now possible. The simplest is to abort the transaction and resubmit it. Another possibility is to turn the colour of the black data white, let the transaction complete and then force the query to reread the modified, white data. A third approach is to ‘turn the transaction’ black by requiring it to wait until the query has read the needed white data and turned it black.

In spite of these problems cases, the enhanced serialisability approach represents an improvement over traditional two-phased locking in terms of the level of query and transaction concurrency that it will support. In a two-phase locking environment, the global read will typically lock all of the tables it references thus blocking all update transactions from executing against them during the life of the query. That is, the level of concurrency is zero. In the enhanced serialisability scheme, many, if not most transactions will be able to serialise before or after the global read and hence execute concurrently with it.

Enhanced serialisability has two noteworthy properties. First, the query does not view a snapshot of the data as in the transient versioning schemes. That is, the data seen by the global read cannot be deemed to have been complete and consistent at some point in time. After the query begins and has read some data, white transactions may update data that it will later read before the query gets to it. The query will not see the updated data as it was when the query started. What the query does see is ‘transaction consistent’ data in that either all or none of the modifications made by a given transaction will be viewed by the query. But transaction consistent data is semantically not the same as the snapshot created by the transient versioning schemes. In many cases it will be acceptable for the data to change ‘under the query’ as it executes, as long as it is transaction consistent. But in other situations a query will in fact need a snapshot or ‘time consistent’ view of the data.

The second property of enhanced serialisability is that while the two-colour scheme may serialise transactions with respect to the query, the ‘repeatable read’ character of the query has been lost. According to the SQL standard, the serialisable isolation level is stronger than the repeatable read isolation level. But one cost of the enhanced serialisability schemes is that a query cannot be guaranteed that if it rereads some data it has already read before it terminates then it will read the same values. A black transaction may have modified the data after the query first read it during the query’s lifetime. Thus the enhanced serialisability approach is only acceptable when it can be guaranteed that the query *will never need to reread data it has already read*. Otherwise it serialises neither before nor after the black transaction.

Another term for enhanced serialisability that is used in [Salem94] is 'altruistic locking'. In [Salem94], long-lived transactions (long queries) 'donate' locked data that they have already read prior to completion. Update transactions may then run in the 'wake' of the query as long as they are restricted to the data which the query has already read and donated, even though the query still has locks on the donated data. In fact, [Salem94] also proposes that queries donate data that they know they will not read so as to increase the probability that transactions will be able to execute concurrently with them. The queries that donate data receive no benefit for doing so; they are enabling updates to execute concurrently. Hence the term 'altruistic' locking.

The demands placed upon the DBMS by enhanced serialisability strategies are quite different from transient versioning. Enhanced versioning does not require multiple copies of data records or pages to be maintained. Indexes are unaffected. However, enhanced serialisability does require more work on the part of queries. In the two-colour scheme, queries must 'colour' the data as they go, which means that queries are performing writes as well as reads. Similarly, the altruistic locking strategy requires queries to donate data that they have read, which again implies that a write takes place. Both schemes effectively require the queries to place locks on the data to be read. In transient versioning schemes, queries neither place locks nor do any writes. The burden of supporting mixed query and transaction execution in a transient versioning environment effectively falls upon the update transactions since they must maintain multiple versions of the data and indexes.

The enhanced serialisability algorithms are quite subtle. [Ammann95] uncovers a flaw in the two-colour scheme proposed in [Pu88], even though [Pu88] claimed that the serialisability of queries and updates governed by the two-colour protocols had been proven. In [Ammann95] it is argued that two colours are in fact not enough, and that 'pure' white and 'pure' black need to be distinguished from 'off-white' and 'off-black'. For example, a black transaction (one that only writes data the query has already read) that *reads* any pure white data changes that data to off-white. Different rules govern what actions transactions may perform against off-coloured data as opposed to pure-coloured data. What can be concluded from this is that enhanced serialisability schemes are not trivial. More research is required to verify that the integrity of the data would not be compromised by them.

[Pu88] and [Salem94] include simulation studies which attempt to assess the likely performance of their proposed mixed workload strategies. [Salem94] found that acceptable performance could only be obtained if two conditions were satisfied. First, queries needed to be intelligent and altruistic enough to donate not only the data they had already read but also other data that transactions running in their wake would likely need. Otherwise, update transactions would be blocked since the altruistic locking protocol requires that a transaction using donated data from some query can *only* use data that the query has donated and nothing else. The second condition that needs to be satisfied for acceptable performance is to abort update transactions that do block so that they do not occupy system resources. Even in the best of circumstances, [Salem94] found that there was a significant variation in transaction response time depending upon how far the single

simulated query had progressed. Early in the query's life few transactions would block. But as the query progressed and donated more data an increasing number of transactions would block when they did not remain completely in the query's wake (i.e., needed data that the query had not donated.)

The simulation study conducted in [Pu88] is even less encouraging regarding the likely performance of enhanced serialisability. The only technique found that yielded acceptable performance was to use transient versioning in addition to the two-colour protocol. Specifically, when a black transaction (running after the query) needed white data (not yet read by the query) it would make its modifications, colour the data black, but require the DBMS to retain a 'white' copy of the original record in a temporary buffer for the query to use. But if an enhanced serialisability scheme needs transient versioning to obtain acceptable performance then it does not add any value. Transient versioning can be used to support concurrent query and transaction workloads by itself without the added complexity of enhanced serialisability techniques.

None of the enhanced serialisability simulation studies were executed with more than one simulated query running at a time. Suppose there were five long-running queries accessing the same data at the same time. Then in the two-colour scheme, all five would need to maintain independent colour data regarding the rows that they had and had not read. None of the papers describing enhanced serialisability discussed how the protocols might work efficiently in the presence of concurrent queries.

The enhanced serialisability schemes do not enable transactions and queries to execute successfully without waiting for each other 100% of the time. There will always be cases in which either the transaction or the query is obliged to wait, or the transaction is aborted, or the query is forced to reread data it has already read. This is a serious drawback compared to transient versioning in which transactions and queries are free to execute as if the other type of workload did not exist. Performance will be an issue for transient versioning, but neither transactions nor queries will ever block, be aborted, or forced to redo work they have already done. Indeed, with transient versioning queries are not required to place locks at all, let alone colour or donate the data.

5.7 Other Related Work

5.7.1 Oracle's Read Consistency

The Oracle DBMS uses a form of transient versioning to support the concurrent execution of queries and transactions. A query will use data as it existed when the query began, which Oracle terms 'read consistency'. This is default behaviour for an Oracle transaction (including queries), requiring no explicit user intervention.

The technique Oracle uses to support read consistency is to leverage its rollback segments. Rollback segments in Oracle are storage areas used to maintain before images of modified rows so that if transactions fail or are aborted for any reason their effects may be rolled back. Rollback segments are distinct from the Oracle redo log, which is the permanent journal used to rollforward lost work in the event of system or media failure.

If a query finds that a record it is reading has been modified since it began it will look in the rollback segments to find the previous version. The technique is not foolproof however. Rollback segments are limited in size and operate as a circular buffer. Older rollback data is periodically deleted to make space for the rollback data created by new transactions. Thus a query may well find that the rollback segments do not contain the previous versions of the data that it needs. The longer the query and the greater the level of update activity, the greater the risk that required prior versions of the data will not be available. Indeed, the primary function of the rollback segments is to support transaction rollback and not read consistency.

Clearly, Oracle is not using on-page caching of prior versions of the data as recommended in [Bober92]. In effect, all prior versions are being sent to a 'version pool' rather than being retained in the same data page as the current row. The performance simulations in [Bober92], however, clearly indicate that on-page caching is vital for good transient versioning performance. If all accesses to prior record versions require a random lookup to a rollback segment which is typically stored on disk then performance is likely to degrade significantly versus a query which uses current versions only. No performance data regarding Oracle's read consistency technique is available in the literature at the time of writing.

Oracle also supports read-consistency across multiple queries. The intention is to allow a number of queries which, for example, are all executed to create a single report to run with the same read-consistent view of the data. All that is required is for the

application to issue a 'Set Transaction Read Only' statement prior to executing the first query, and all subsequent queries will effectively run with the same start time.

A similar notion, called 'view sharing' is proposed in [Bober92]. The intention is to allow different queries to run with the same effective start time so that the number of different versions of records that need to be maintained is minimised. While the purpose of view-sharing is to optimise the performance of transient versioning, Oracle is right to recognise that an application may also need to execute multiple queries against the same version of the data. Transient versioning schemes should support this capability at the application level provided that the start times of the related queries are sufficiently close.

5.7.2 Temporal Databases

Transient versioning is transparent to the user and application. From the user point of view, the only data that exists in the DBMS is the current data, at least as at the time a query began. All maintenance of and access to prior versions of records is managed internally by the DBMS.

A very different model, known as a temporal database, has also been described in the literature. In effect, a time dimension is added to the data so that it is always known for what period of time a particular row or even data value within a row was valid. In general, previous rows and values are retained permanently along with the temporal information needed to indicate when the previous data applied. Thus, a temporal database may be regarded as implementing permanent versioning. Typically, the period during

which specific data values are valid is defined by the time of the inserts, updates and deletes against the data.

In a temporal database, users are aware of the existence of data in time, and may explicitly query the data as of some point or interval in time. Thus, the conceptual model of the database changes significantly from the non-temporal database.

In principle, a temporal database could support the execution of concurrent queries and transactions. Queries could be executed, for example, as of their start time. That is, they would be explicitly constrained to retrieve data values that were valid as of the moment they began. Update transactions could continue to modify the latest data values, which would not affect the value of the data as of the time the query began.

However, one would not implement a temporal database merely to support the execution of mixed workloads. Temporal databases are costly since they keep every data value ever created. They will only be implemented in environments where the nature of the application makes it critical that a permanent, historical record be maintained and also made accessible via explicit temporal queries. For most applications, either no record of previous data values needs to be permanently retained, or selected prior data is kept in the form of a history or journal table created for the purpose. Thus for most applications, it will be more appropriate to use transient versioning to support mixed workloads than temporal databases.

5.7.3 *Cursor Stability*

Cursor stability is sometimes mentioned in the literature as a technique that can be used to support concurrent transaction and query execution. It has been implemented in some database systems such as IBM's DB2/MVS.

The idea of cursor stability is that as a query executes it obtains a lock on the page it next needs to read, reads it, and then releases the lock. That is, the query violates the two-phase locking protocol: it releases locks even though it will subsequently acquire more locks. Of course, by releasing locks on pages after it reads them a query minimises its interference with concurrently executing transactions.

The cost is that the query is not serialisable with transactions that access data both before and after the query. If a transaction updates data that the query later reads, then the transaction should serialise before the query. But if the same transaction subsequently updates data that the query has already read and released, then it should serialise after the query. As a result the query may see inconsistent data. While all of the data that a query reads will be committed since it locks a page before reading it, a consistent view of the data still cannot be guaranteed. Thus cursor stability does not represent a significant improvement in data integrity versus queries that execute in read-uncommitted mode.

By virtue of the fact that cursor stability does still lock pages, it has its disadvantages compared to read-uncommitted mode. Locking uses system and DBMS resources not required by read-uncommitted queries. Moreover, some transactions will

inevitably need to access the page that the query currently has locked, which will result in response time delays.

CONCLUSIONS AND FUTURE WORK

6.1 Summary of Results and Conclusions

The key result of our research is that the performance of mixed OLTP and DSS workloads is manageable. Whether or not mixed workloads are used will be determined primarily by business rather than performance issues.

Certainly, the addition of a query workload to an existing OLTP system will cause transaction response times to increase. But first, those increased response times may still be acceptable, especially given that response time variability as measured by the coefficient of variation actually improves. Second, in some cases new databases will be created and used immediately for both transactions and queries, thus setting realistic transaction response time expectations. And third, normally when new workloads are added to an existing system the platform is scaled up proportionately, especially if it was already running near capacity. In that case a new query workload may have little or no impact on transaction response time.

Hence, one cannot simply rule out mixed workloads on the grounds that the impact on transaction performance will necessarily be disastrous, nor that query and transaction execution times will become erratic and unpredictable. Our research shows that transaction and query performance in a mixed workload environment is consistent and

reasonable in the sense that it is not irrational or inexplicable. As a result, mixed workload performance becomes manageable; even a mundane capacity and workload planning issue.

The major caveats regarding the feasibility of mixed OLTP and DSS workloads are at the business and not the technical level. The key question is whether or not organisations can live with the inexact query results produced by the read uncommitted isolation level. If not, and if the business has rules excluding updates to historical data, then it may be possible to partition the data into current and historical tables and then to restrict transactions to the current data and queries to the historical data. A cleanly partitioned environment, however, is difficult to achieve.

There are other reasons for maintaining separate OLTP and DSS databases: the OLTP database may be non-relational and so not suited to decision support queries. The data may be geographically distributed rendering large scale queries impractical.

But notwithstanding all of these caveats, there are many users of sizeable relational databases who will be in a position to execute both queries and transactions against the same database at the same time. For them, inexact answers will be acceptable, and/or partitioning will be feasible. The issue then is whether the performance of mixed workloads can be managed so that it is acceptable to the user communities, and our answer is an unequivocal yes.

We did find that a light transaction workload suffered a larger relative response time increase in the presence of queries than a medium transaction workload. However, after the queries were added the absolute transaction response times for both the light and

medium workloads were about the same. It is also the case that different types of queries can have a different impact on transaction response time which is a potential issue if only a single query queue is used. That is, transaction response times would vary depending upon the particular type of query executing at the time. However, when there were three concurrent query streams running, the impact on transaction response time was consistent whatever query types were executed.

There is no question that if mixed workloads are to be feasible then some form of query management system will be needed. It will not be acceptable to simply allow DSS users to submit demanding, ad hoc queries directly against the database at will. This would result in a 'lumpy' query workload and corresponding lumpiness in query and transaction performance. Any query management system will be based on queues, and may well place queries in different queues depending on their estimated execution time.

From our research it is clear that careful attention will need to be paid to the disk I/O capacity of the mixed workload platform. If the transaction workload itself uses all or nearly all of the disk I/O bandwidth then queries may receive little or no service, depending on the priority management scheme being used. In any case, ensuring that there is sufficient I/O capacity available to cope with the workloads placed on the system is a straightforward capacity planning and data placement issue.

It is also clear that the priority management scheme is critical to the success of mixed workload execution. First, it would be helpful if DBMS supported a priority management scheme since controlling priorities at the operating system level is generally

not feasible: administrators have no control over the processes spawned by the DBMS. Secondly, short OLTP transactions need to be given priority versus long DSS queries. In the test platform used for this research, priority appeared to be lowered dynamically as a function of execution time. While this is a reasonable policy, one would like to be able as an administrator to specify a priority for a workload at the DBMS level. In this way, queries could be assigned a lower priority than transactions and so would begin execution at a lower priority rather than at the same priority as transactions. This would help maintain short and consistent transaction response times.

A DBMS priority manager should also ensure, however, that the long transactions (i.e., the queries) receive some minimum level of service since it is essential that the queries execute in a predictable, maximum length of time. Lower priority queries should not be starved, even in the presence of heavy transaction volumes. Most organisations will likely also want to create a queue for short queries which execute with a reasonably high level of priority such that rapid turnaround of short queries would be the norm.

Resource utilisation data indicates that OLTP and DSS workloads are to some degree complementary. OLTP workloads are normally I/O intensive and make less use of the CPU. DSS workloads, on the other hand, are more CPU intensive. Thus, there is a potential opportunity to make more efficient use of system resources by mixing the two types of workloads.

The platform used for this research was very much I/O bound. Even so we found that mixed workload system resource utilisation was about as efficient as the

corresponding standalone workloads. There was some small loss in efficiency when the medium transaction workload was run with queries, but there were gains with the light transaction workload. In other words, mixed workloads did not cause performance to degrade to the point that more hardware would be needed than if the workloads were run on separate systems.

The tests of mixed workload performance in a partitioned or split database environment versus a traditional unified database did show that the split database offered some performance benefits. But the improvements in transaction and query response times in the split database were modest so that in our opinion few organisations would likely implement a partitioned data environment with its associated administrative cost and complexity for performance reasons alone. If the nature of the business is inherently periodic and prohibits changes to historic data then a partitioned database model may be worthwhile.

An attempt to create a hot spot query stream in the unified environment failed because the DBMS optimiser did not take advantage of the date restriction to current month data. But in the split environment, the hot spot stream against the current partition was both fast and efficient, to the point of having a significant detrimental effect on transaction performance.

Since queries against current data are important, good hot spot performance could be viewed as a significant advantage for the split environment. However, we expect optimisers to improve so that such queries will eventually perform well against unified

databases as well. Moreover, in the split environment, hot spot query priority would likely need to be reduced, with a resulting slowdown in performance, in order to not have too adverse an impact on transaction response time.

The most important conclusion regarding hot spot queries, however, is that they can indeed be executed in a mixed workload environment. They are very CPU intensive and so help to make very efficient use of system resources in conjunction with concurrent transactions. While, again, they can cause a significant increase in transaction response time, it should be possible to manage the response time impact in a DBMS that allows administrators to set an appropriate priority level for the hot spot query stream.

A number of proposals have been made in the literature for adding sophisticated capabilities to databases so that mixed workloads could be supported without obliging queries to run at the read-uncommitted isolation level. On the whole, these ideas have not been implemented in commercial DBMS products.

In our view, the most promising of these proposals is the transient versioning scheme in which the DBMS automatically maintains prior versions of modified rows just as long as any executing query may need them to reconstitute the state of the data as it was when the query began. While the scheme is theoretically attractive, it would definitely exact a significant performance price since the DBMS would be obliged to dynamically maintain multiple versions of rows and their indices. The performance of a versioning scheme would certainly be no better than the mixed workload results seen in our testing where there was only one copy of any row and queries accepted inaccurate results. When

and if robust transient versioning implementations become available, organisations will need to decide whether completely correct results to DSS queries are sufficiently important to warrant the performance cost of versioning.

6.2 Key Issues for Mixed Workloads

What are the key issues that will determine whether or not an organisation should execute mixed OLTP and DSS workloads against a single database system? We address these in two groups: business and technical.

6.2.1 *Business Issues*

- Can the inexact results of DSS queries executed at the read-uncommitted isolation level be accepted? If not, the organisation will either need to adopt a partitioned data strategy, or build a separate data warehouse, or wait for robust implementations of new technologies such as transient versioning.
- Where is the data now? Is it in a non-relational DBMS that does not support DSS query access? Is it geographically distributed at many locations? In either case, mixed workloads of any size will not be feasible.
- Is there any batch window left for executing queries when the OLTP applications are inactive? If there is little or no such batch window then the organisation will either need to build a data warehouse or run OLTP and DSS applications on the same database at the same time.

- Have user expectations for transaction response times and throughput already been set? That is, is a new query workload being added to an existing OLTP environment? In this case, mixed workloads are only likely to be acceptable if the platform can be scaled so as to maintain the performance characteristics of the OLTP applications.
- Is the business inherently periodic in nature with strict rules regarding the updatability of historic data? In so, a partitioned data strategy may be advisable.

6.2.2 Technical Issues

- Is the DBMS robust enough to support mixed workloads? Or does it ‘break’ when significant mixed workloads are attempted?
- What is the priority management scheme used by the DBMS, if any? Does it enable administrators to execute queries at a lower priority than transactions? Can it also ensure that queries receive a minimum level of service?
- Is there an effective query management system available that will support the execution of queries through queues? It would be especially helpful if queries could automatically be placed in the appropriate queue after submission by users based on estimated execution times, time of day, and other appropriate parameters. The DBMS should be able to execute queries from the various queues at different priorities as specified by the administrator.

- How mature is the optimiser? In particular, does it support efficient execution of queries that target current data in a unified database? If not, partitioned tables would be useful for such queries.
- How effectively does the DBMS support the management of the available I/O capacity through the correct placement of data, index and other structures?
- What spare capacity is available on the hardware platform, and is the platform scalable? In particular, is the I/O capacity scalable?

6.3 Future Work

There is definitely a need for further research into the performance characteristics of mixed OLTP and DSS workloads. First, experiments need to be conducted using a variety of DBMS products, hardware configurations, and data placement options. Two areas of particular interest are parallel processing DBMSs, and configurations in which there is a better balance of disk I/O and CPU capacity than was the case in our experiments. That is, if there are more I/O paths to the CPU(s) and the data is placed appropriately to take advantage of them how well do mixed workloads perform?

We have found that priority management plays a key role in determining mixed workload performance, but were unable to modify task priority in the test platform used. A subject for future research, then, would be to test DBMSs that have different priority management schemes, and that preferably allowed some administrator control of priorities.

In particular, can transaction performance be improved if queries begin at a lower priority, while at the same time guaranteeing a minimum level of service to queries?

The priority manager has a role to play in another key issue for future research: can transaction performance characteristics be held constant when new DSS workloads are added? Apart from the priority manager, the other major factor here is the scalability of the DBMS and hardware platform. Does scalability give organisations the option of adding DSS workloads to existing OLTP environments without compromising OLTP performance? Our research results suggest that it should, but only future research can confirm this conjecture.

We have suggested that OLTP and DSS workloads are potentially complementary in their utilisation of system resources. Further testing using a variety of configurations is needed to determine whether or not this is true. If so, then the impact of queries and transactions on each other's performance will be less than it would be if they were both, say, disk I/O intensive workloads.

We have reported the results of experiments in which one and three query streams were executed concurrently with transaction workloads, and observed that three query streams had proportionately less effect on transaction performance than one stream. We suspect that four or more query streams would continue to demonstrate a decreasing impact on transaction performance, but again further research is needed to verify the hypothesis.

Finally, the whole area of transient versioning, two-colour schemes for enhanced serialisability, and so on is a rich field for future performance research, once commercial implementations of such techniques become available. Even at this stage, it would be very helpful to have some independent performance studies of Oracle's 'read consistency' technology.

6.4 Final Words

So why, once again, do mixed workloads matter? They matter because there are many organisations that either have or are building large relational databases and who want to be able to use them for decision support purposes as well as for traditional transaction processing applications. Can they simply run both workloads on the same database at the same time without unacceptable performance results, particularly for the lifeblood OLTP systems? Because if not, they must either live without DSS query access to the database, replicate the data into a data warehouse with all of the costs that entails, or perhaps try to run some reports during the few hours left in the week, if any, when the OLTP systems are down.

In other words, mixed workloads are important because for many organisations the alternatives are either expensive, not palatable, or non-existent. And because in today's business environment having DSS query access to one's data is no longer optional. If you must be able to run DSS queries against your data, it would certainly help if running them against the same data the OLTP applications are using were an option.

The whole purpose of databases, of course, is to create a single data store that can be used to satisfy the needs of multiple applications. Relational database theory in particular has gone to great lengths to develop application-neutral data structures. To the extent that we insist on separate, special-purpose databases ‘optimised’ for either OLTP or DSS use, but certainly not both, we are retreating from the original intention of database systems, and relational database systems in particular.

While there are many reasons why organisations may choose not to run OLTP and DSS workloads against their data at the same time, our research results argue strongly that unacceptable performance should not be one of them. In the right circumstances users do have a choice; mixed workloads are an option. What’s more, as DBMSs become more robust, and as optimisers, priority managers and query managers improve, that option will become more attractive.

APPENDIX A: SAR RESULTS

Unified Data Results:

	CPU %usr	CPU %sys	CPU %wio	CPU %idle	Disk 1 % Busy	Pages freed per second
Heavy transactions	14	7	79	0	98	48
Medium transactions	11	7	75	7	90	43
Light transactions	10	5	62	23	72	35
3-up queries	30	9	60	1	88	130
qset1a	50	9	37	4	54	64
qset3	8	6	72	14	86	181
qset5a	40	10	46	4	59	40
Medium t/cs + 3-up queries	29	9	62	0	95	95
Medium t/cs + qset1a	31	10	59	0	87	60
Medium t/cs + qset3	12	7	80	1	99	88
Medium t/cs + qset5a	26	11	63	0	90	55
Light t/cs + 3-up queries	29	9	62	0	95	99
Light t/cs + qset1a	36	11	53	0	81	58
Light t/cs + qset3	11	6	81	2	98	102
Light t/cs + qset5a	31	13	56	0	84	51

Split Data Results:

	CPU %usr	CPU %sys	CPU %wio	CPU %idle	Disk 1 %busy	Pages freed per second
Medium transactions	15	7	67	11	82	44
Light transactions	11	5	53	31	61	34
Generic 3-up queries	33	9	57	1	87	138
3-up queries incl. hot spot	43	6	48	3	73	113
qspl1a	54	8	34	4	52	66
qspl3	8	7	70	15	86	165
qspl5a	43	10	46	1	56	37
hot spot stream	79	5	15	1	23	25
Medium t/cs + generic 3-up	31	9	60	0	94	101
Medium t/cs + 3-up incl. hot spot	34	8	58	0	94	96
Medium t/cs + qspl1a	38	11	51	0	82	63
Medium t/cs + qspl3	13	6	80	1	98	96
Medium t/cs + qspl5a	33	12	55	0	86	56
Medium t/cs + hot spot stream	33	8	59	0	86	53
Light t/cs + generic 3-up	33	9	58	0	94	108
Light t/cs + 3-up incl. hot spot	34	8	58	0	94	101
Light t/cs + qspl1a	42	11	47	0	76	66
Light t/cs + qspl3	12	7	79	2	98	111
Light t/cs + qspl5a	36	13	51	0	79	59
Light t/cs + hot spot stream	45	7	48	0	70	48

APPENDIX B: PERFORMANCE RESULTS

Notes of explanation for the following tables can be found at the end of this appendix.

Unified Data: Impact of 3-up Queries on Transactions

	Absolute Elapsed Time (sec)	Elapsed Time Increase Factor	Absolute T/C Response Time (sec)	Standard Deviation of Mean Response Times	Response Time Increase Factor	Response Time Coefficient of Variation
Heavy transaction workload (1500/250 repetitions)	980		0.574	0.003		0.894
Heavy T/Cs with 3-up queries (1500/250)	1752	1.79	1.050	0.016	1.83	0.781
Medium transaction workload (1900/250 repetitions)	1203		0.385	0.004		1.031
Medium transaction workload (1520/200 repetitions)	961		0.379	0.003		1.063
Medium T/Cs with 3-up queries (1900/250)	2304	1.91	0.980	0.011	2.55	0.796
Medium T/Cs with 3-up queries incl. hot spot stream (1520/200)	1832	1.91	0.967	0.008	2.55	0.829
Light transaction workload (1140/150 repetitions)	950		0.294	0.006		1.145
Light T/Cs with 3-up queries (1140/150)	1766	1.86	1.030	0.012	3.50	0.811
Light T/Cs with 3-up queries incl. hot spot stream (1140/150)	1739	1.83	1.006	0.011	3.42	0.828

Unified Data: Impact of One Query on Transactions

	Absolute Elapsed Time (sec)	Elapsed Time Increase Factor	Absolute T/C Response Time (sec)	Standard Deviation of Mean Response Times	Response Time Increase Factor	Response Time Coefficient of Variation
Heavy transaction workload (1500/250 repetitions)	980		0.574	0.003		0.894
Heavy transaction workload with qset1a (1500/250)	1320	1.35	0.783	0.009	1.36	0.878
Heavy transaction workload with qset3 (1500/250)	1398	1.43	0.831	0.026	1.45	0.817
Heavy transaction workload with qset5a (1500/250)	1188	1.21	0.702	0.011	1.22	0.907
Medium transaction workload (1900/250 repetitions)	1203		0.385	0.004		1.031
Medium transaction workload (1520/200 repetitions)	961		0.379	0.003		1.063
Medium transaction workload (1140/150 repetitions)	722		0.378	0.002		1.083
Medium transaction workload with qset1a (1520/200)	1348	1.40	0.643	0.006	1.70	0.963
Medium transaction workload with qset3 (1900/250)	1808	1.50	0.706	0.009	1.84	0.872
Medium transaction workload with qset5a (1140/150)	962	1.33	0.596	0.011	1.58	0.992
Medium transaction workload & hot spot stream (1520/200)	1280	1.33	0.596	0.008	1.58	0.941
Light transaction workload (1140/150 repetitions)	950		0.294	0.006		1.145
Light transaction workload (760/100 repetitions)	649		0.307	0.008		1.196
Light transaction workload (570/75 repetitions)	512		0.335	0.004		1.150
Light transaction workload with qset1a (760/100)	913	1.41	0.668	0.005	2.18	1.186
Light transaction workload with qset3 (1140/150)	1342	1.41	0.647	0.005	2.20	0.933
Light transaction workload with qset5a (570/75)	661	1.29	0.608	0.012	1.81	1.067
Light transaction workload with qset6 (570/75)	656	1.28	0.599	0.011	1.79	1.136

Unified Data: Impact of Transactions on 3-up Queries

	Average Elapsed Time (sec)	Standard Deviation of Average Elapsed Times	Increase Factor
Generic 3-up queries standalone	1735	29	
Generic 3-up queries with medium transaction workload	5186	169	2.99
Generic 3-up queries with light transaction workload	4233	36	2.44
Hot spot plus two other query streams standalone	1873	28	
Hot spot plus two other query streams with medium T/Cs	6014	314	3.21
Hot spot plus two other query streams with light T/Cs	4838	156	2.58

Unified Data: Impact of Transactions on One Query Stream

	Elapsed Time (sec)	Standard Deviation of Average Elapsed Times	Increase Factor
qset1a standalone	597	9	
qset3 standalone (45 repetitions)	813	12	
qset5a standalone	436	7	
hot spot stream standalone	634	10	
qset1a with heavy T/C workload	3817	N/A	6.39
qset3 with heavy T/C workload	3679	N/A	4.53
qset5a with heavy T/C workload	3179	N/A	7.28
qset1a with medium T/C workload	1840	17	3.08
qset3 with medium T/C workload	2752	32	3.39
qset5a with medium T/C workload	1523	36	3.49
hot spot stream with medium T/C workload	2119	50	3.34
qset1a with light T/C workload	1320	8	2.21
qset3 with light T/C workload	2005	27	2.47
qset5a with light T/C workload	1044	28	2.39
hot spot stream with light T/C workload	1450	16	2.29

Split Data: Impact of 3-up Queries on Transactions

	Absolute Elapsed Time (sec)	Elapsed Time Increase Factor	Absolute T/C Response Time (sec)	Standard Deviation of Mean Response Times	Response Time Increase Factor	Response Time Coefficient of Variation
Medium transaction workload (1520/200 repetitions)	826		0.294	0.005		1.225
Medium transaction workload (760/100 repetitions)	408		0.282	0.005		1.301
Medium T/Cs with generic 3- up queries (1520/200)	1761	2.13	0.928	0.013	3.15	0.889
Med. T/Cs with 3-up queries incl. hot spot queries (760/100)	919	2.26	0.976	0.022	3.47	0.907
Light transaction workload (1140/150 repetitions)	729		0.229	0.006		1.328
Light transaction workload (570/75 repetitions)	446		0.240	0.007		1.375
Light T/Cs with generic 3-up queries (1140/150)	1639	2.25	0.918	0.020	4.01	0.940
Light T/Cs with 3-up queries incl. hot spot queries (570/75)	869	1.95	1.004	0.020	4.18	0.870

Split Data: Impact of One Query on Transactions

	Absolute Elapsed Time (sec)	Elapsed Time Increase Factor	Absolute T/C Response Time (sec)	Standard Deviation of Mean Response Times	Response Time Increase Factor	Response Time Coefficient of Variation
Medium transaction workload (1140/150 repetitions)	610		0.285	0.003		1.197
Medium transaction workload (950/125 repetitions)	509		0.285	0.005		1.246
Medium transaction workload (570/75 repetitions)	306		0.275	0.006		1.306
Medium transaction workload with qsp11a (1140/150)	951	1.56	0.596	0.004	2.09	1.254
Medium transaction workload with qsp13 (1140/150)	965	1.58	0.607	0.019	2.13	0.988
Medium transaction workload with qsp15a (950/125)	710	1.39	0.505	0.011	1.77	1.153
Medium transaction workload with hot spot queries (570/75)	442	1.44	0.525	0.010	1.91	1.204
Light transaction workload (760/100 repetitions)	585		0.236	0.005		1.400
Light transaction workload (570/75 repetitions)	446		0.240	0.007		1.375
Light transaction workload (380/50 repetitions)	295		0.231	0.006		1.526
Light transaction workload (304/40 repetitions)	239		0.229	0.006		1.544
Light transaction workload with qsp11a (570/75)	678	1.52	0.664	0.007	2.76	1.280
Light transaction workload with qsp13 (760/100)	845	1.45	0.587	0.014	2.49	1.114
Light transaction workload with qsp15a (380/50)	404	1.37	0.527	0.019	2.28	1.408
Light transaction workload with hot spot queries (304/40)	387	1.62	0.735	0.045	3.22	2.046

Split Data: Impact of Transactions on 3-up Queries

	Average Elapsed Time (sec)	Standard Deviation of Average Elapsed Times	Increase Factor
Generic 3-up queries standalone	1529	21	
Generic 3-up queries with medium transaction workload	3949	193	2.58
Generic 3-up queries with light transaction workload	3184	139	2.08
Hot spot plus two other query streams standalone	1884	29	
Hot spot plus two other query streams with medium T/Cs	2689	39	1.43
Hot spot plus two other query streams with light T/Cs	2238	33	1.19

Split Data: Impact of Transactions on One Query Stream

	Elapsed Time (sec)	Standard Deviation of Average Elapsed Times	Increase Factor
qspl1a standalone	559	10	
qspl3 standalone	736	5	
qspl5a standalone	383	6	
hot spot stream standalone	518	12	
qspl1a with medium T/C workload	1410	32	2.52
qspl3 with medium T/C workload	2247	56	3.05
qspl5a with medium T/C workload	1070	13	2.79
hot spot stream with medium T/C workload	719	28	1.39
qspl1a with light T/C workload	1069	13	1.91
qspl3 with light T/C workload	1660	33	2.25
qspl5a with light T/C workload	795	5	2.08
hot spot stream with light T/C workload	632	13	1.22

Notes of explanation regarding the performance results tables in this appendix:

The transaction workloads in the charts are qualified by the number of repetitions they executed. The notation (1500/250) means that the two copies of the payment insert transaction program each processed 1500 inserts, while the two copies of the t2 transaction program each executed 250 'cycles'. A 'cycle' in the t2 transaction program consisted of 7 transactions: 4 payment updates, 2 beneficiary queries and 1 beneficiary insert. Thus 250 'cycles' implies that the two t2 programs processed $2 \times 250 \times 7 = 3500$ transactions. Hence, the (1500/250) notation means that the transaction workload included $2 \times 1500 + 3500 = 6500$ transactions.

The phrase '3-up queries including the hot spot stream' means that 3 concurrent query streams were executed, one of which was the hot spot stream. In fact, the hot spot query stream replaced qset5a or qspl5a which consisted of a single long query. The phrase 'generic 3-up queries' means three query streams none of which was the hot spot stream.

The 'Absolute T/C Elapsed Time' is the weighted average time taken by the four transaction programs to execute their workloads, where the weightings were their respective transaction volumes. The 'Absolute T/C Response Time' is also an average of the response times for the different transaction types in a workload weighted by the number of times they were executed.

The 'Elapsed Time Increase Factor' in the tables showing the impact of queries on transactions was obtained by dividing the transaction elapsed time in the mixed workload

by the standalone transaction elapsed time for the same transaction volume. Similarly, the 'Response Time Increase Factor' was calculated by dividing the transaction response time in the mixed workload by the response time of the standalone workload that had the same number of transactions.

Four to ten tests runs were executed for each test type, and where applicable the weighted average transaction response time was calculated for each test. The standard deviation of the weighted average transaction response times for the different test runs of a given type is recorded in the column 'Standard Deviation of Mean Response Times'. Note that this is not a measure of the variability of transaction response times; rather it measures the consistency of test results across test runs.

In order to calculate the coefficient of variation for transaction response times, the average standard deviation of transaction response times was first computed, again weighted by the number of transactions of each type in a workload. Then the weighted average standard deviation was divided by the weighted average transaction response time for the workload to arrive at the coefficient of variation. The coefficient of variation is thus a measure of transaction response time variability; it does not indicate the variability of test results between different test runs.

In the charts reporting the impact of transactions on queries, the average elapsed time of the queries in 3-up streams is a simple average of the completion times of the three streams. The 'Increase Factor' is obtained by dividing the elapsed time of a query stream or streams in a mixed workload by its standalone elapsed time.

The column 'Standard Deviation of Average Elapsed Times' measures the variability of the query elapsed time results between different test runs of the same type. In three tests involving the heavy transaction workload, only one test of a particular type was executed, and so the standard deviation column is shown as 'N/A' (not applicable).

qset3 and qspl3 each executed 2 short transactions in a single 'repetition'. Both programs always executed 45 repetitions or 90 short queries in total. The elapsed times for qset3 and qspl3 thus reflect the time taken to execute the 90 short queries.

APPENDIX C: TABLE DEFINITIONS

Following are the create table statements used for the test tables in the unified database. The split database tables definitions were the same, except that there were two additional tables, Current_Header and Current_Detail, with the same columns as the Payment_Header and Payment_Detail tables.

```
/* June 3, 1997          */

Create table Beneficiary
( Beneficiary_Id      Integer      Not Null
, Birthdate           Date         Not Null
, Civil_Status        Char(1)
, Admissibility_Code  Char(1)      Not Null
, Admissibility_Start Date         Not Null
, Admissibility_End   Date
, Sex                 Char(1)      Not Null
, Family_Name         Char(30)     Not Null
, Other_Names         Char(20)     Not Null
, Address_1           Char(30)     Not Null
, Address_2           Char(30)     Not Null
, Address_3           Char(30)
, Postal_Code         Char(7)      Not Null
, Social_Security_Ind Char(1)      Not Null
)
with journaling
;

Create table Category_of_Prof
( Prof_Id             Integer      Not Null
, Prof_Category       Integer1     Not Null
, Start_Date          Date         Not Null
, End_Date            Date
, Create_Date         Date         Not Null
, Expiry_Date         Date
, Principal_Flag      Char(1)      Not Null
)
with journaling
;

Create table Diagnosis
( Diagnosis_Code       Smallint     Not Null
, Diagnosis_Name       Char(30)     Not Null
)
with journaling
;
```

Create table Establishment

```
( Establishment_No      Integer      Not Null
  ,Type_Code             Char(1)      Not Null
  ,Start_Date            Date         Not Null
  ,End_Date              Date
  ,Name                  Char(30)     Not Null
  ,Address_1             Char(30)     Not Null
  ,Address_2             Char(30)     Not Null
  ,Address_3             Char(30)
  ,Postal_Code           Char(7)      Not Null
)
```

with journaling
;

Create table Norm

```
( Agreement_No          Smallint     Not Null
  ,Version_No           Integer1     Not Null
  ,Service_Code         Integer      Not Null
  ,Prof_Category        Integer1     Not Null
  ,Start_Date           Date         Not Null
  ,End_Date             Date
)
```

with journaling
;

Create table Payment_Detail

```
( Control_No            Integer      Not Null
  ,Line_No              Integer1     Not Null
  ,Service_Code         Integer      Not Null
  ,Service_Date         Date         Not Null
  ,Decision_Date        Date
  ,Payment_Date         Date
  ,Payment_Amount       Money
  ,Payment_Decision     Integer1
  ,Reason_Code          Char(2)
)
```

with journaling
;

Create table Payment_Header

```
( Control_No            Integer      Not Null
  ,Prof_Id              Integer      Not Null
  ,Establishment_No     Integer
  ,Date_Admitted        Date
  ,Date_Discharged      Date
  ,Beneficiary_Id       Integer      Not Null
  ,Diagnosis_Code       Smallint     Not Null
  ,Create_Date          Date         Not Null
)
```

with journaling
;

Create table Professional

```
( Prof_Id               Integer      Not Null
  ,Prof_Class           Char(1)      Not Null
  ,Name                 Char(30)     Not Null
  ,Birthdate            Date         Not Null
)
```

```

,Sex                Char(1)      Not Null
,SIN                Integer      Not Null
,Telephone          Char(12)     Not Null
,Language           Char(1)      Not Null
,Citizenship        Char(1)      Not Null
,Civil_Status       Char(1)      Not Null
,Location           Smallint     Not Null
,Create_Date        Date         Not Null
,Expiry_Date        Date         Not Null
,Bank               Integer      Not Null
,Profile_No         Integer      Not Null
,University         Integer      Not Null
,Edu_Specialty      Smallint     Not Null
,Geographic_Code    Smallint     Not Null
,CLSC_Code          Smallint     Not Null
,Registered_Date    Date         Not Null
)
with journaling
;

Create table Prof_Address
( Prof_Id            Integer      Not Null
,Address_Type       Char(1)      Not Null
,Address_1          Char(30)     Not Null
,Address_2          Char(30)     Not Null
,Address_3          Char(30)     Not Null
,Postal_Code        Char(7)      Not Null
)
with journaling
;

Create table Prof_Admissibility
( Prof_Id            Integer      Not Null
,Admis_Status       Char(1)      Not Null
,Status_Start_Date  Date         Not Null
,Status_End_Date    Date         Not Null
,Create_Date        Date         Not Null
,Expiry_Date        Date         Not Null
)
with journaling
;

Create table Prof_Category
( Prof_Category      Smallint     Not Null
,Category_Description Char(30)    Not Null
)
with journaling
;

Create table Region
( Region_Code        Integer1     Not Null
,Region_Name         Char(30)     Not Null
)
with journaling
;

Create table Region_Postal_Code

```



```
( Region_Code          Integer1    Not Null
  ,Postal_Code_3       Char(3)     Not Null
)
with journaling
;
```

```
Create table Service
( Service_Code          Integer     Not Null
  ,Agreement_No         Smallint    Not Null
  ,Version_No           Integer1    Not Null
  ,Start_Date           Date        Not Null
  ,End_Date             Date
  ,Service_Type_Code    Smallint    Not Null
  ,Rate_Type            Integer1    Not Null
  ,Base_Rate            Money       Not Null
)
with journaling
;
```

```
Create table Service_Type
( Service_Type_Code     Smallint    Not Null
  ,Service_Type_Name    Char(30)    Not Null
)
with journaling
;
```

APPENDIX D: TRANSACTION AND QUERY SQL

This appendix contains all of the SQL used in the various test application programs. It first shows the SQL used by the transaction (OLTP) applications, and then the query (DSS) SQL.

D.1 Transaction SQL

First, here is the SQL used to create the procedures that insert payment header and detail rows. These procedures were invoked by the payment insert transaction program (ti.sc in Appendix E). In order to use these procedures in the split database environment, the references to the payment_header and payment_detail tables were changed to 'current_header' and 'current_detail' respectively.

```
/* insert a payment_header row */

create procedure insert_payhdr
    (control_no          integer
    ,prof_id             integer
    ,establishment_no    integer
    ,date_admitted       date
    ,date_discharged     date
    ,beneficiary_id      integer
    ,diagnosis_code      smallint
    ,create_date         date
    ) as
declare
    cnt    integer;
begin
    select count(*) into :cnt from professional
    where prof_id = :prof_id
    ;
    if :cnt = 0 then
        return -1;
    endif;
    if :establishment_no is not null then
        select count(*) into :cnt from establishment
        where establishment_no = :establishment_no
        ;
        if :cnt = 0 then
            return -2;
        endif;
    endif;
    select count(*) into :cnt from beneficiary
    where beneficiary_id = :beneficiary_id
```

```

;
if :cnt = 0 then
    return -3;
endif;
insert into payment_header values
(:control_no
, :prof_id
, :establishment_no
, :date_admitted
, :date_discharged
, :beneficiary_id
, :diagnosis_code
, :create_date
);
if irowcount <> 1 then
    return -4;
else
    return 0;
endif;
end;

/* insert a payment_detail row */

create procedure insert_detail
(control_no          integer
, line_no           integer1
, service_code       integer
, service_date       date
, decision_date      date
, payment_date       date
, payment_amount     money
, payment_decision   integer1
, reason_code        char(2)
, prof_id            integer /* for integrity check */
, beneficiary_id     integer /* for integrity check */
) as
declare
    cnt integer;
begin
    select count(*) into :cnt from payment_header
    where control_no = :control_no
    ;
    if :cnt = 0 then
        return -5;
    endif;
    select count(*) into :cnt from prof_admissibility pa
    where pa.prof_id = :prof_id
        and pa.admis_status = 'A'
        and int4(interval('days', :service_date -
            pa.status_start_date)) >= 0
        and (int4(interval('days', pa.status_end_date -
            :service_date)) >= 0
            or pa.status_end_date is null)
        and int4(interval('days', 'today' -
            pa.create_date)) >= 0
        and (int4(interval('days', pa.expiry_date - 'today')) >= 0
            or pa.expiry_date is null)

```

```

;
if :cnt = 0 then
    return -1;
endif;
select count(*) into :cnt from beneficiary b
where b.beneficiary_id = :beneficiary_id
    and b.admissibility_code = 'A'
    and int4(interval('days', :service_date -
        b.admissibility_start)) >= 0
    and (int4(interval('days', b.admissibility_end -
        :service_date)) >= 0
        or b.admissibility_end is null)
;
if :cnt = 0 then
    return -2;
endif;
select count(*) into :cnt from service
where service_code = :service_code
;
if :cnt = 0 then
    return -3;
endif;
insert into payment_detail values
(:control_no
, :line_no
, :service_code
, :service_date
, :decision_date
, :payment_date
, :payment_amount
, :payment_decision
, :reason_code
);
if rowcount <> 1 then
    return -4;
else
    return 0;
endif;
end;

```

The following code in `tl.sc` invoked the `insert_payhdr` procedure. If the `establishment_no` was -1 then it was null.

```

if(payhdr.establishment_no != -1)
    exec sql execute procedure insert_payhdr
        (control_no = :payhdr.control_no
        , prof_id = :payhdr.prof_id
        , establishment_no = :payhdr.establishment_no
        , date_admitted = :payhdr.date_admitted
        , date_discharged = :payhdr.date_discharged
        , beneficiary_id = :payhdr.beneficiary_id
        , diagnosis_code = :payhdr.diagnosis_code
        , create_date = :payhdr.create_date
        ) into :result_code;
else
    exec sql execute procedure insert_payhdr
        (control_no = :payhdr.control_no

```

```

,prof_id = :payhdr.prof_id
,establishment_no = null
,date_admitted = null
,date_discharged = null
,beneficiary_id = :payhdr.beneficiary_id
,diagnosis_code = :payhdr.diagnosis_code
,create_date = :payhdr.create_date
) into :result_code;

```

Similarly, t1.sc used the following code to invoke the insert_detail procedure. If payment_decision was zero, then it was null.

```

if(paydet.payment_decision != 0)
  exec sql execute procedure insert_detail
  (control_no = :paydet.control_no
  ,line_no = :paydet.line_no
  ,service_code = :paydet.service_code
  ,service_date = :paydet.service_date
  ,decision_date = :paydet.decision_date
  ,payment_date = :paydet.payment_date
  ,payment_amount = :paydet.payment_amount
  ,payment_decision = :paydet.payment_decision
  ,reason_code = :paydet.reason_code
  ,prof_id = :payhdr.prof_id
  ,beneficiary_id = :payhdr.beneficiary_id
  ) into :result_code;
else
  exec sql execute procedure insert_detail
  (control_no = :paydet.control_no
  ,line_no = :paydet.line_no
  ,service_code = :paydet.service_code
  ,service_date = :paydet.service_date
  ,decision_date = null
  ,payment_date = null
  ,payment_amount = null
  ,payment_decision = null
  ,reason_code = null
  ,prof_id = :payhdr.prof_id
  ,beneficiary_id = :payhdr.beneficiary_id
  ) into :result_code;

```

t2.sc sql: Program t2.sc executed three transaction types: payment updates, beneficiary inserts and beneficiary queries. The payment update transaction first selected the existing data and then updated it using the input data with the following SQL. In the split environment the table reference was changed from payment_detail to current_detail.

```

EXEC SQL repeated select payment_amount
  into :old_amount:amount_ind
  from payment_detail
  where control_no = :control_no
  and line_no = :line_no
;

EXEC SQL repeated update payment_detail
  set payment_amount = :new_amount

```

```

,decision_date = '1997.05.18'
,payment_date = '1997.05.20'
,payment_decision = :paydet.payment_decision
,reason_code = :paydet.reason_code
where control_no = :control_no
and line_no = :line_no
;

```

The next SQL statements inserted a new beneficiary row and then selected some of the new data back. 'ben' was a structure that included all 14 fields of the beneficiary table.

```

EXEC SQL repeated insert into beneficiary
values (:ben:ben_ind);

EXEC SQL repeated select beneficiary_id
, family_name
, address_3
into :ben.beneficiary_id:ben_ind[0]
, :ben.family_name:ben_ind[7]
, :ben.address_3:ben_ind[11]
from beneficiary
where beneficiary_id = :ben.beneficiary_id
;

```

The most complex transaction executed by t2.sc was the beneficiary query. First the beneficiary table data was retrieved. Then all of the payment headers for the beneficiary were retrieved, and for each header all of the corresponding payment details were selected.

```

EXEC SQL select * into :ben:ben_ind from beneficiary
where beneficiary_id = :ben.beneficiary_id
;

EXEC SQL declare header_cur cursor for
select * from payment_header
where beneficiary_id = :ben.beneficiary_id
order by control_no
;

EXEC SQL declare detail_cur cursor for
select * from payment_detail
where control_no = :payhdr.control_no
order by line_no
;

```

In the split environment, the beneficiary query cursors were defined as a union as follows to retrieve both the current and historical data.

```

EXEC SQL declare header_cur cursor for
select * from payment_header
where beneficiary_id = :ben.beneficiary_id
union
select * from current_header
where beneficiary_id = :ben.beneficiary_id
order by control_no
;

```

```
EXEC SQL declare detail_cur cursor for
  select * from payment_detail
  where control_no = :payhdr.control_no
  union
  select * from current_detail
  where control_no = :payhdr.control_no
  order by line_no
;
```

D.2 Query SQL

qsetla.sc: executed two long queries. The first retrieved the beneficiaries in descending order of the total amount spent on them. Only the 100 most expensive beneficiaries were fetched in the application program.

```
EXEC SQL DECLARE query5_cur CURSOR FOR
  Select ph.beneficiary_id
        ,ifnull(sum(pd.payment_amount), 0.0)
  from payment_header ph
        ,payment_detail pd
  where ph.control_no = pd.control_no
  group by ph.beneficiary_id
  order by 2 desc
;
```

The second query performed in qsetla selected the id, name and category of each professional, as well as the number of services performed and amount paid for those services in the last 90 days. In the split environment, the query retrieved data for the 90 days to April 30, 1997, which was the most recent date in the payment_detail table.

```
EXEC SQL declare query1_cur cursor for
  Select ph.prof_id
        ,pro.name
        ,cop.prof_category
        ,count(*)
        ,ifnull(sum(payment_amount), 0.0)
  from payment_header ph
        ,payment_detail pd
        ,category_of_prof cop
        ,professional pro
  where ph.control_no = pd.control_no
        and int4(interval('days', '1997.05.20' -
        pd.payment_date)) < 90
        and pd.payment_amount is not null
        and pro.prof_id = ph.prof_id
        and pro.prof_id = cop.prof_id
        and cop.principal_flag = 'Y'
  group by ph.prof_id
        ,pro.name
        ,cop.prof_category
  order by ph.prof_id
```

qset3.sc: Short queries. These queries each took an input parameter. For the first query, the input parameter was a professional id. The costs incurred by the professional in the last 90 days were retrieved by service code. The application program fetched the 10 most expensive service codes. Again, in the split environment, the data for the 90 days prior to April 30, 1997 was retrieved.

```
/* query1a: For a particular professional, find the amount */
/* paid by service code for the 10 most expensive service codes */
/* in the past 90 days. Prof_ids are randomly generated. */
```

```
EXEC SQL DECLARE query1a_cur CURSOR FOR
  Select pd.service_code
        ,ifnull(sum(pd.payment_amount), 0.0)
  from payment_detail pd, payment_header ph
  where ph.prof_id = :prof_id
        and ph.control_no = pd.control_no
        and int4(interval('days', date('1997.05.15') -
        pd.payment_date)) < 90
  group by pd.service_code
  order by 2 desc
;
```

The second short query in qset3.sc took a date as a parameter. It would retrieve payment data for the given date by establishment (hospital) within region.

```
EXEC SQL declare query7_cur cursor for
  Select r.region_name
        ,e.name
        ,ifnull(sum(pd.payment_amount), 0.0)
  from region r,
        region_postal_code rpc,
        establishment e,
        payment_header ph,
        payment_detail pd
  where ph.establishment_no = e.establishment_no
        and ph.control_no = pd.control_no
        and r.region_code = rpc.region_code
        and left(e.postal_code, 3) = rpc.postal_code_3
        and pd.payment_date = :report_date
  group by r.region_name
        ,e.name
  order by 1,2
;
```

qset5a.sc: One long query. For a particular service type (142), the amounts paid by professional category since April 15, 1997 were reported. This represented a 30 day period since the 'current' date was May 15, 1997. In the split environment, the data from April 1 to April 30 was selected.

```
EXEC SQL declare query13_cur cursor for
  Select pc.category_description
        ,ifnull(sum(pd.payment_amount), 0.0)
```



```

from category_of_prof cop
    ,prof_category pc
    ,payment_detail pd
    ,payment_header ph
    ,service s
where ph.prof_id = cop.prof_id
    and cop.principal_flag = 'Y'
    and cop.prof_category = pc.prof_category
    and ph.control_no = pd.control_no
    and int4(interval('days', '1997.05.15' -
        cop.create_date)) >= 0
    and (int4(interval('days', cop.expiry_date -
        '1997.05.15')) >= 0 or
        cop.expiry_date is null)
    and int4(interval('days', pd.payment_date -
        '1997.04.15')) >= 0
    and pd.service_code = s.service_code
    and s.service_type_code = 142
group by pc.category_description
order by 1
;

```

qspl6.sc: These are the hot spot queries for the split environment. All six were executed three times in three 'cycles'.

The first hot spot query counts the number of distinct beneficiaries seen by each professional in the current month for which the professional has received some payment.

```

EXEC SQL declare query1c_cur cursor for
    Select ch.prof_id
        ,pro.name
        ,cop.prof_category
        ,count(distinct ch.beneficiary_id)
    from current_header ch
        ,current_detail cd
        ,category_of_prof cop
        ,professional pro
    where ch.control_no = cd.control_no
        and cd.payment_amount is not null
        and pro.prof_id = ch.prof_id
        and pro.prof_id = cop.prof_id
        and cop.principal_flag = 'Y'
    group by ch.prof_id
        ,pro.name
        ,cop.prof_category
    order by 4 desc
;

```

The second hot spot query calculates the total amounts paid in the current month by beneficiary age group within region.

```

EXEC SQL Insert into query4b
    Select r.region_name
        ,int4(interval('years', '1997.05.20' -
            b.birthdate)) / 10
        ,ifnull(sum(cd.payment_amount), 0.0)

```

```

from beneficiary b,
     region r,
     region_postal_code rpc,
     current_header ch,
     current_detail cd
where left(b.postal_code, 3) = rpc.postal_code_3
and rpc.region_code = r.region_code
and b.beneficiary_id = ch.beneficiary_id
and ch.control_no = cd.control_no
group by r.region_name
        ,b.birthdate
;

```

The third hot spot query was used to retrieve the 10 most costly beneficiaries in the current month.

```

EXEC SQL DECLARE query5b_cur CURSOR FOR
Select ch.beneficiary_id
      ,ifnull(sum(cd.payment_amount), 0.0)
from current_header ch
     ,current_detail cd
where ch.control_no = cd.control_no
group by ch.beneficiary_id
order by 2 desc
;

```

The fourth hot spot query reports the total costs by establishment within region month-to-date.

```

EXEC SQL declare query7b_cur cursor for
Select r.region_name
      ,e.name
      ,ifnull(sum(cd.payment_amount), 0.0)
from region r,
     region_postal_code rpc,
     establishment e,
     current_header ch,
     current_detail cd
where ch.establishment_no = e.establishment_no
and ch.control_no = cd.control_no
and r.region_code = rpc.region_code
and left(e.postal_code, 3) = rpc.postal_code_3
group by r.region_name
        ,e.name
order by 1,2
;

```

The fifth hot spot query totals the amounts paid to each professional in the current month for services to beneficiaries less than 18 years old.

```

EXEC SQL declare query9_cur cursor for
Select p.prof_id
      ,p.name
      ,ifnull(sum(cd.payment_amount), 0.0)
from professional p
     ,beneficiary b

```

```

        ,current_header ch
        ,current_detail cd
where cd.control_no = ch.control_no
      and ch.beneficiary_id = b.beneficiary_id
      and int4(interval('years', '1997.05.31' -
        b.birthdate)) < 18
      and ch.prof_id = p.prof_id
group by p.prof_id
        ,p.name
order by p.name
;

```

The sixth and last hot spot query counted the number of services billed by each professional in the current month for which the professional was not authorized per the Norm table.

```

EXEC SQL declare queryllb_cur cursor for
  Select ch.prof_id
        ,count(*)
  from current_header ch
        ,current_detail cd
  where ch.control_no = cd.control_no
        and cd.service_code not in
        (Select norm.service_code
  from norm
        ,category_of_prof cop
  where norm.prof_category = cop.prof_category
        and ch.prof_id = cop.prof_id
        and int4(interval('days', cd.service_date -
        cop.start_date)) >= 0
        and (int4(interval('days', cop.end_date -
        cd.service_date)) >= 0 or cop.end_date is null)
        and int4(interval('days', 'today' - cop.create_date)) >= 0
        and (int4(interval('days', cop.expiry_date - 'today')) >=
0
        or cop.expiry_date is null)
        and int4(interval('days', cd.service_date -
norm.start_date))
        >= 0
        and (int4(interval('days', norm.end_date -
cd.service_date))
        >= 0 or norm.end_date is null)
        )
  group by ch.prof_id
  order by 1
;

```

APPENDIX E: SAMPLE PROGRAMS

Two sample programs are included here to illustrate how transactions and queries were executed. `t1.sc` is a transaction program that processed payment inserts. `qset3.sc` was the query application program that processed short DSS queries. Both were forked by the driver program as required in appropriate tests, and ran against the unified environment. `qset3.sc` was always executed with 45 'repetitions' (i.e., the two queries were both executed 45 times).

E.1 `t1.sc`: Sample Transaction Program

```
/* t1.sc: Payment insert transaction. Reads a file of payment */
/* header and detail data and inserts it into the database after */
/* performing a number of integrity checks. */
/* The program takes two arguments: The first is the test number. */
/* The second is the number of transactions to run. A 'transaction' */
/* is a payment header insert plus the inserts of its associated */
/* payment detail lines. */
/* Input: tldata.d */
/* Output: -t1.n, where n is the test number argument */
/* The output includes the transaction name, */
/* test number, tcs executed, mean time, standard deviation, */
/* overall execution time and number of failures (transaction */
/* rejects). */
/* -t1msgs.n, where n is the test number argument. This file */
/* includes error messages from rejected transactions. */
/* Jim Paulin. June 9, 1997 */
/* June 13: removed the whenever statements. Moved connect and */
/* disconnect before and after the sigaction calls respectively to */
/* eliminate 'alarm clock' errors. */
/* June 18: added max transaction time. */
/* -increased max number of tcs to 5000. */
/* June 19: added ttime printout every 20th tc */
/* July 3: increased max number of t/cs to 8000 from 5000. */

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>
#include <stdlib.h> /* for atol and atof functions */
#include <math.h> /* for pow function */
```

```

#include <signal.h>

#define TEST 0

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;

    struct payhdr_ {
        long   control_no;
        long   prof_id;
        long   establishment_no;
        char   date_admitted[26];
        char   date_discharged[26];
        long   beneficiary_id;
        short  diagnosis_code;
        char   create_date[26];
    } payhdr;

    struct paydet_ {
        long   control_no;
        short  line_no;
        long   service_code;
        char   service_date[26];
        char   decision_date[26];
        char   payment_date[26];
        double payment_amount;
        short  payment_decision;
        char   reason_code[3];
    } paydet;

    int result_code; /* for execute procedure statements */
    char errmsg[101]; /* used by inquire_sql */

EXEC SQL END DECLARE SECTION;

/* function prototypes */
int  header(void);
void detail(int line);
void hold(int mcsecs);
void sig_alm(void);

FILE *tldata; /* input data file */
FILE *tlmsgs; /* messages file */
FILE *output; /* result file */
char msgfile[15] = "tlmsgs."; /* message file name */
int  lines; /* number of detail lines for a header */
int  sleep_usecs; /* microseconds to sleep after a transaction */
double ttime[8000]; /* transaction execution times */
double maxtime = 0.0; /* maximum transaction execution time. */
int  tcs; /* number of transactions to execute (2nd argument) */
int  tcounter = 0; /* number of transactions executed */
struct timeval tcstart, tcend; /* tc start and end times */
char tempstr[40]; /* temporary holder for input data */
int  c; /* tempstr subscript */
int  hdr_rejects = 0; /* count of rejected headers */
int  det_rejects = 0; /* count of rejected details */

```

```

struct itimerval wait_times;
sigset_t  newmask, oldmask, suspmask;

int  main(int argc, char * argv[]) {

    char  outname[15] = "t1."; /* output file name */
    struct timeval  starttime, endtime; /* pgm start and end */
    double  seconds, usecs; /* used to execute a tc */
    int  i; /* loop counter */
    int  hdr_return; /* header() return code */
    double  sumtimes = 0.0; /* sum of trans execution times */
    double  mean, stddev; /* of trans execution times */
    double  diff_squared = 0.0; /* used in stddev calculation */
    struct sigaction  newaction, oldaction; /* signal handling */

    if(gettimeofday(&starttime, (char *) 0) == -1) {
        perror("t1 error at gettimeofday - 1");
        exit(-1);
    }

    printf("t1 starttime is %s\n", ctime(&starttime.tv_sec));

    if(argc != 3) {
        printf("t1 expects 2 arguments: testno "
               "and transactions to run\n");
        exit(-1);
    }

    if((tldata = fopen("tldata.d", "r")) == NULL) {
        perror("t1 failed to open tldata.d\n");
        exit(-1);
    }
    strcat(outname, argv[1]);
    if((output = fopen(outname, "w")) == NULL) {
        perror("t1 failed to open output file");
        exit(-1);
    }
    strcat(msgfile, argv[1]);
    if((tlmsgs = fopen(msgfile, "w")) == NULL) {
        perror("t1 failed to open message file");
        exit(-1);
    }
    ;
    tcs = atol(argv[2]);
    if(tcs > 8000) {
        fprintf(tlmsgs, "Cannot process more than "
                "8000 transactions\n");
        exit(-1);
    }
    ;
    fprintf(tlmsgs, "Number of transactions to run is %d\n",
            tcs);

    /* The connect should come before the signal handling */
    /* function calls to avoid alarm clock errors. */
    EXEC SQL CONNECT 'irpdb';
    if(sqlca.sqlcode < 0) {
        exec sql inquire _sql (:errmsg = errortext);
        fprintf(tlmsgs, "connect failed with "

```

```

        "message: %s\n", errmsg);
    exit(-1);
}

/* Set up for hold timer function. */
/* See ~/thesis/test/timer.c for documentation. */
newaction.sa_handler = sig_alm;
sigemptyset(&newaction.sa_mask);
newaction.sa_flags = 0;
if(sigaction(SIGALRM, &newaction, &oldaction) != 0) {
    fprintf(tlmsgs, "tl: first call to sigaction "
        "failed\n");
    exit(-1);
}
sigemptyset(&newmask);
sigaddset(&newmask, SIGALRM);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
suspmask = oldmask;
sigdelset(&suspmask, SIGALRM);
wait_times.it_interval.tv_sec = 0;
wait_times.it_interval.tv_usec = 0;
wait_times.it_value.tv_sec = 0;

/* If header() returns -1 then a serious error occurred */
/* or the end of the input file has been reached. */
/* If header() returns 1 then the header insert failed */
/* and the detail lines must be skipped. But the next */
/* transaction can be attempted. */
/* If header() returns 0 then the insert was successful. */
while(tccounter < tcs && (hdr_return = header()) != -1) {
    tccounter++;
    if(hdr_return == 0)
        for(i = 0; i < lines; i++)
            detail(i + 1); /* i - 1 = line_no */
    exec sql commit;
    if(sqlca.sqlcode < 0) {
        exec sql inquire sql (:errmsg = errortext);
        fprintf(tlmsgs, "commit failed with "
            "message: %s\n", errmsg);
        exit(-1);
    }
    if(gettimeofday(&tcend, (char *) 0) == -1) {
        perror("tl error at gettimeofday - 4");
        exit(-1);
    }
    seconds = (double) (tcend.tv_sec - tcstart.tv_sec);
    if(tcend.tv_usec > tcstart.tv_usec)
        usecs = (double) (tcend.tv_usec -
            tcstart.tv_usec) / 1000000.0;
    else {
        usecs = (double) (1000000.0 - tcstart.tv_usec
            + tcend.tv_usec) / 1000000.0;
        seconds -= 1.0;
    }
    ttime[tccounter - 1] = seconds + usecs;
    if(ttime[tccounter - 1] > maxtime)
        maxtime = ttime[tccounter - 1];
}

```

```

        if(tcounter == 1 || tcounter % 20 == 0)
            fprintf(tlmsgs, "ttime[%d] = %.2f "
                    "seconds\n", tcounter - 1,
                    ttime[tcounter - 1]);
        hold(sleep_usecs);
    }

    /* reset the SIGALRM handling */
    if(sigaction(SIGALRM, &oldaction, NULL) != 0) {
        fprintf(tlmsgs, "tl: second call to sigaction "
                "failed\n");
        exit(-1);
    }
    sigprocmask(SIG_SETMASK, &oldmask, NULL);

    /* The disconnect should come after the signal handling */
    /* has been reset to avoid alarm clock errors. */
    EXEC SQL DISCONNECT;
    if(sqlca.sqlcode < 0) {
        exec sql inquire sql (:errmsg = errormsg);
        fprintf(tlmsgs, "disconnect failed with "
                "message: %s\n", errmsg);
        exit(-1);
    }

    if(gettimeofday(&endtime, (char *) 0) == -1) {
        perror("tl error at gettimeofday - 2");
        exit(-1);
    }
    /* note the the mean and standard deviation of the */
    /* transaction execution times do not include the */
    /* sleep times */
    for(i = 0; i < tcounter; i++)
        sumtimes += ttime[i];
    if(tcounter > 0)
        mean = sumtimes / tcounter;
    else
        mean = sumtimes;
    if(tcounter > 1) {
        for(i = 0; i < tcounter; i++)
            diff_squared += pow(ttime[i] - mean, 2);
        stddev = sqrt(diff_squared / (tcounter - 1));
    }
    else
        stddev = 0.0;
    if(TEST)
        for(i = 0; i < tcounter; i++)
            fprintf(output, "ttime[%d] = %f\n",
                    i, ttime[i]);
    /* output: transaction name, starttime, elapsed time, */
    /* number of transactions executed, the mean and */
    /* standard deviation of the execution times, */
    /* and the counts of rejected headers and detail lines. */
    fprintf(output, "tl: %d %d %.3f %.3f %.3f %d %d\n"
            "starttime: %s",
            endtime.tv_sec - starttime.tv_sec,
            tcounter, mean, stddev, maxtime,

```



```

        hdr_rejects, det_rejects,
        ctime(&starttime.tv_sec));
printf("t1 endtime is %s\n", ctime(&endtime.tv_sec));
fclose(output);
fclose(tlmsgsgs);
fclose(tldata);

return(0);
}

int header(void) {
    int i; /* loop counter */

    if(gettimeofday(&tcstart, (char *) 0) == -1) {
        perror("t1 error at gettimeofday - 3");
        exit(-1);
    }

    /* read the header data in */

    if((tempstr[0] = fgetc(tldata)) == EOF) {
        printf("End of tldata reached\n");
        return(-1);
    }
    c = 1;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;
    tempstr[c] = '\0';
    payhdr.control_no = atol(tempstr);
    c = 0;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;
    tempstr[c] = '\0';
    payhdr.prof_id = atol(tempstr);
    c = 0;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;
    tempstr[c] = '\0';
    payhdr.establishment_no = atol(tempstr);
    c = 0;
    while((payhdr.date_admitted[c] = fgetc(tldata)) != ',')
        c++;
    payhdr.date_admitted[c] = '\0';
    c = 0;
    while((payhdr.date_discharged[c] = fgetc(tldata)) != ',')
        c++;
    payhdr.date_discharged[c] = '\0';
    c = 0;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;
    tempstr[c] = '\0';
    payhdr.beneficiary_id = atol(tempstr);
    c = 0;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;

```

```

tempstr[c] = '\0';
payhdr.diagnosis_code = atol(tempstr);
c = 0;
while((payhdr.create_date[c] = fgetc(tldata)) != ',')
    c++;
payhdr.create_date[c] = '\0';
lines = fgetc(tldata) - '0';
fgetc(tldata); /* skip comma */
c = 0;
while((tempstr[c] = fgetc(tldata)) != '\n')
    c++;
tempstr[c] = '\0';
sleep_usecs = atol(tempstr);

if(payhdr.establishment_no != -1)
    exec sql execute procedure insert_payhdr
        (control_no = :payhdr.control_no
        ,prof_id = :payhdr.prof_id
        ,establishment_no = :payhdr.establishment_no
        ,date_admitted = :payhdr.date_admitted
        ,date_discharged = :payhdr.date_discharged
        ,beneficiary_id = :payhdr.beneficiary_id
        ,diagnosis_code = :payhdr.diagnosis_code
        ,create_date = :payhdr.create_date
        ) into :result_code;
else
    exec sql execute procedure insert_payhdr
        (control_no = :payhdr.control_no
        ,prof_id = :payhdr.prof_id
        ,establishment_no = null
        ,date_admitted = null
        ,date_discharged = null
        ,beneficiary_id = :payhdr.beneficiary_id
        ,diagnosis_code = :payhdr.diagnosis_code
        ,create_date = :payhdr.create_date
        ) into :result_code;

if(sqlca.sqlcode < 0) {
    exec sql inquire sql (:errmsg = errormsg);
    fprintf(tlmsgs, "insert_payhdr failed for "
        "control_no %d with error message: \n\t%s\n",
        payhdr.control_no, errmsg);
    hdr_rejects++;
    return(-1);
}

if(result_code != 0)
    hdr_rejects++;
switch(result_code) {
    case -1:
        fprintf(tlmsgs, "header insert failed since "
            "prof_id %d not found\n",
            payhdr.prof_id);
        return(1);
        break;
    case -2:
        fprintf(tlmsgs, "header insert failed since "

```

```

        "establishment_no %d not found\n",
        payhdr.establishment_no);
    return(1);
    break;
case -3:
    fprintf(tlmsgs, "header insert failed since "
        "beneficiary_id %d not found\n",
        payhdr.beneficiary_id);
    return(1);
    break;
case -4:
    fprintf(tlmsgs, "header insert failed "
        "for control_no %d\n",
        payhdr.control_no);
    return(1);
    break;
case 0:
    return(0);
    break;
default:
    fprintf(tlmsgs, "invalid result_code %d "
        "received from insert_payhdr\n",
        result_code);
    return(1);
    break;
}

;

void detail(int line) {

    int i; /* loop counter */

    /* read the detail data in */

    c = 0;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;
    tempstr[c] = '\0';
    paydet.control_no = atol(tempstr);
    /* skip the line number */
    fgetc(tldata); fgetc(tldata);
    paydet.line_no = line;
    c = 0;
    while((tempstr[c] = fgetc(tldata)) != ',')
        c++;
    tempstr[c] = '\0';
    paydet.service_code = atol(tempstr);
    c = 0;
    while((paydet.service_date[c] = fgetc(tldata)) != ',')
        c++;
    paydet.service_date[c] = '\0';
    c = 0;
    while((paydet.decision_date[c] = fgetc(tldata)) != ',')
        c++;
    paydet.decision_date[c] = '\0';
    c = 0;

```

```

while((paydet.payment_date[c] = fgetc(tldata)) != ',')
    c++;
paydet.payment_date[c] = '\\0';
c = 0;
while((tempstr[c] = fgetc(tldata)) != ',')
    c++;
tempstr[c] = '\\0';
paydet.payment_amount = atof(tempstr);
c = 0;
while((tempstr[c] = fgetc(tldata)) != ',')
    c++;
tempstr[c] = '\\0';
paydet.payment_decision = atol(tempstr);
for(c = 0; c < 2; c++)
    paydet.reason_code[c] = fgetc(tldata);
paydet.reason_code[2] = '\\0';
if(fgetc(tldata) != '\\n') {
    printf("t1 exiting: missing newline in tldata "
           "detail line\\n");
    exit(-1);
}

if(paydet.payment_decision != 0)
    exec sql execute procedure insert_detail
        (control_no = :paydet.control_no
        ,line_no = :paydet.line_no
        ,service_code = :paydet.service_code
        ,service_date = :paydet.service_date
        ,decision_date = :paydet.decision_date
        ,payment_date = :paydet.payment_date
        ,payment_amount = :paydet.payment_amount
        ,payment_decision = :paydet.payment_decision
        ,reason_code = :paydet.reason_code
        ,prof_id = :payhdr.prof_id
        ,beneficiary_id = :payhdr.beneficiary_id
        ) into :result_code;
else
    exec sql execute procedure insert_detail
        (control_no = :paydet.control_no
        ,line_no = :paydet.line_no
        ,service_code = :paydet.service_code
        ,service_date = :paydet.service_date
        ,decision_date = null
        ,payment_date = null
        ,payment_amount = null
        ,payment_decision = null
        ,reason_code = null
        ,prof_id = :payhdr.prof_id
        ,beneficiary_id = :payhdr.beneficiary_id
        ) into :result_code;

if(sqlca.sqlcode < 0) {
    exec sql inquire sql (:errmsg = errortext);
    fprintf(tlmsgs, "insert_payhdr failed for "
           "control_no %d with error message: \\n\\t%s\\n",
           payhdr.control_no, errmsg);
    det_rejects++;
}

```

```

        return;
    }

    if(result_code != 0)
        det_rejects++;
    switch(result_code) {
        case -1:
            fprintf(tlmsgs, "detail insert failed since "
                    "prof_id %d not admissible\n",
                    payhdr.prof_id);
            break;
        case -2:
            fprintf(tlmsgs, "detail insert failed since "
                    "beneficiary_id %d not admissible\n",
                    payhdr.beneficiary_id);
            break;
        case -3:
            fprintf(tlmsgs, "detail insert failed since "
                    "service_code %d not found\n",
                    paydet.service_code);
            break;
        case -4:
            fprintf(tlmsgs, "detail insert failed "
                    "for control_no %d and line_no %d\n",
                    paydet.control_no, paydet.line_no);
            break;
        case -5:
            fprintf(tlmsgs, "detail insert failed since "
                    "header missing for control no %d\n",
                    paydet.control_no);
            break;
        case 0:
            break;
        default:
            fprintf(tlmsgs, "invalid result_code %d "
                    "received from insert_detail\n",
                    result_code);
            break;
    }
}

void hold(int mcsecs) {
    if(TEST)
        printf("Holding for %d mcsecs\n", mcsecs);
    wait_times.it_value.tv_usec = mcsecs;
    if(setitimer(ITIMER_REAL, &wait_times, NULL) != 0) {
        fprintf(tlmsgs, "t1: setitimer call failed "
                "with mcsecs = %d\n", mcsecs);
        exit(-1);
    }
    if(sigsuspend(&suspmask) != -1) {
        fprintf(tlmsgs, "t1: sigsuspend call failed\n");
        exit(-1);
    }
    if(TEST)

```

```

        printf("Exiting hold\n");
    }

    void sig_alrm(void) {
        return; /* Do nothing. Just wake up sigsuspend. */
    }

```

E.2 qset3.sc: Sample Query Program

```

/* qset3.sc: Some shorter running queries that are executed a */
/* number of times. For each 'repetition' all of the short */
/* queries are executed. */
/* The program takes two arguments. The first is the test */
/* number. The second is the number of repetitions. */
/* The queries are: */
/* query1a: For a particular professional, find the amount */
/* paid by service code for the 10 most expensive service codes */
/* in the past 90 days. Prof_ids are randomly generated. */
/* query7: For a randomly chosen date, find total costs */
/* by establishment. Sort by establishment within region. */
/* June 16, 1996. Jim Paulin */
/* July 4: added ifnull clauses and open cursor checks. */
/* Also added a message file to qset3.sc */

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <stdlib.h>
#include <math.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;

struct qlareult_ {
    long service_code;
    double sum_amount;
}; qlareult;

struct q7result_ {
    char region_name[31];
    char estab_name[31];
    double sum_amount;
}; q7result;

int prof_id; /* used in query1a */
char report_date[11]; /* used in query 7 */

char errmsg[101]; /* for inquire_sql function */

EXEC SQL END DECLARE SECTION;

#define TEST 0 /* running in test mode? */

```

```

/* function prototypes */
int  exec_query1a(void);
int  exec_query7(void);

FILE *qset3msgs; /* message file pointer */
double rndm; /* random number in [0.0, 1.0) */
double qset3time[100]; /* individual repetition execution times */
double qlatime[100], q7time[100]; /* individual query exec. times */
int  j; /* repetition loop counter */
struct timeval qlastart, qlaend, q7start, q7end;

int main(int argc, char * argv[]) {

    FILE *times_fp; /* execution times file pointer */
    char timesfile[15] = "qset3.";
    char msgfile[17] = "qset3msgs.";
    int repetitions; /* no. of times to repeat query */
    struct timeval starttime, endtime; /* for the whole program */
    int k; /* loop counter */
    int sumtimes = 0; /* sum of repetition execution times */
    double mean, stddev; /* of repetition execution times */
    double diff_squared = 0.0; /* used in stddev calculation */
    double qset3sec, qset3usec;

    gettimeofday(&starttime, (char *) 0);
    printf("qset3 starttime is %s\n", ctime(&starttime.tv_sec));
    if(argc != 3) {
        printf("qset3: expects 2 arguments\n");
        exit(-1);
    }

    strcat(timesfile, argv[1]); /* argv1 contains the test number */
    if((times_fp = fopen(timesfile, "w")) == NULL) {
        printf("qset3: Cannot open file %s\n", timesfile);
        exit(-1);
    }

    strcat(msgfile, argv[1]);
    if((qset3msgs = fopen(msgfile, "w")) == NULL) {
        printf("qset3: Cannot open file %s\n",
            msgfile);
        exit(-1);
    }

    EXEC SQL CONNECT 'jrpdb';
    if(sqlca.sqlcode != 0) {
        printf("qset3: connect failed\n");
        exit(-1);
    }

    EXEC SQL set lockmode session where readlock = nolock;
    if(sqlca.sqlcode != 0) {
        printf("qset3: set session nolock failed\n");
        exit(-1);
    }
}

```

```

    srand48(time(0));

    repetitions = atoi(argv[2]);
    if(repetitions < 1 || repetitions > 100) {
        printf("queryla: repetitions = %d. Exiting.\n",
            repetitions);
        exit(-1);
    }
    for(j = 0; j < repetitions; j++) {
        exec_queryla();
        exec_query7();
        qset3sec = (double) (q7end.tv_sec - qlastart.tv_sec);
        if(q7end.tv_usec > qlastart.tv_usec)
            qset3usec = (double) (q7end.tv_usec -
                qlastart.tv_usec)/1000000.0;
        else {
            qset3usec = (double) (1000000.0 -
                qlastart.tv_usec +
                q7end.tv_usec)/1000000.0;
            qset3sec -= 1.0;
        }
        qset3time[j] = qset3sec + qset3usec;
    }

    EXEC SQL DISCONNECT;
    gettimeofday(&endtime, (char *) 0);
    printf("qset3 endtime is %s\n", ctime(&endtime.tv_sec));
    for(k = 0; k < repetitions; k++)
        sumtimes += qset3time[k];
    mean = (double) sumtimes / repetitions;
    if(repetitions > 1) {
        for(k = 0; k < repetitions; k++)
            diff_squared += pow(qset3time[k] - mean, 2);
        stddev = sqrt(diff_squared/(repetitions - 1));
    }
    else
        stddev = 0.0;
    fprintf(times_fp, "qset3 %d %d %.2f %.2f\n",
        endtime.tv_sec - starttime.tv_sec,
        repetitions, mean, stddev);
    if(fclose(times_fp) == EOF)
        printf("Error closing qset3 timesfile\n");
    if(TEST)
        if(fclose(qset3msgs) == EOF)
            printf("Error closing qset3 msg file\n");
    return(0);
}

int exec_queryla(void) {
    int i; /* loop counter */
    double qlasec, qlausec;

    EXEC SQL DECLARE queryla_cur CURSOR FOR
        Select pd.service_code

```



```

        ,ifnull(sum(pd.payment_amount), 0.0)
    from payment_detail pd, payment_header ph
    where ph.prof_id = :prof_id
        and ph.control_no = pd.control_no
        and int4(interval('days', date('1997.05.15') -
            pd.payment_date)) < 90
    group by pd.service_code
    order by 2 desc
;

gettimeofday(&qlastart, (char *) 0);
rndm = drand48();
prof_id = 3001 + (int) (rndm * 1000.0);
if(prof_id == 3211 || prof_id == 3420 || prof_id == 3933)
    prof_id++;
if(TEST) {
    fprintf(qset3msgs, "The prof_id is %d\n\n",
        prof_id);
}

if(TEST)
    printf("queryla: about to open cursor\n");
EXEC SQL OPEN queryla_cur FOR READONLY;
if(sqlca.sqlcode != 0) {
    EXEC SQL INQUIRE SQL (:errmsg = errormsg);
    fprintf(qset3msgs, "Open queryla_cur failed "
        "with message: %s\n", errmsg);
    EXEC SQL ROLLBACK;
    return(-1);
}

if(TEST)
    fprintf(qset3msgs, "Following are results "
        "from queryla:\n\n");
i = 0;
while(i < 10) {
    EXEC SQL FETCH queryla_cur INTO :qlareult;
    if(sqlca.sqlcode != 0)
        break;
    if(TEST)
        fprintf(qset3msgs, "%d %.2f\n",
            qlareult.service_code,
            qlareult.sum_amount);
    i++;
}

EXEC SQL COMMIT;

gettimeofday(&qlaend, (char *) 0);
qlasec = (double) (qlaend.tv_sec - qlastart.tv_sec);
if(qlaend.tv_usec > qlastart.tv_usec)
    qlausec = (double) (qlaend.tv_usec -
        qlastart.tv_usec)/1000000.0;
else {
    qlausec = (double) (1000000.0 - qlastart.tv_usec +
        qlaend.tv_usec)/1000000.0;
    qlasec -= 1.0;
}

```

```

    }
    qlatime[j] = qlasec + qlausec;
    if(TEST)
        printf("queryla qlatime[%d] = %.2f\n", j, qlatime[j]);
    return(0);
}

int exec_query7(void) {
    int    i; /* loop counter */
    double  q7sec, q7usec;
    int    ryear, rmonth, rday;

    /* ifnull added July 4 */
    EXEC SQL declare query7_cur cursor for
        Select r.region_name
            ,e.name
            ,ifnull(sum(pd.payment_amount), 0.0)
        from region r,
            region_postal_code rpc,
            establishment e,
            payment_header ph,
            payment_detail pd
        where ph.establishment_no = e.establishment_no
            and ph.control_no = pd.control_no
            and r.region_code = rpc.region_code
            and left(e.postal_code, 3) = rpc.postal_code_3
            and pd.payment_date = :report_date
        group by r.region_name
            ,e.name
        order by 1,2
        ;

    gettimeofday(&q7start, (char *) 0);
    /* generate a random payment date from 1996.05.01 */
    /* to 1997.05.15 using only days 1 to 28 of the month. */
    /* See the corresponding logic in payment.c. */
    rndm = drand48();
    if(rndm < .638) {
        ryear = 1996;
        rndm = drand48();
        rmonth = 5 + (int) (rndm * 8.0);
    }
    else {
        ryear = 1997;
        rndm = drand48();
        if(rndm < .882)
            rmonth = 1 + (int) (rndm/.882 * 4.0);
        else
            rmonth = 5;
    }
    rndm = drand48();
    if(ryear == 1997 && rmonth == 5)
        rday = 1 + (int) (rndm * 15.0);
    else
        rday = 1 + (int) (rndm * 28.0);
}

```

```

sprintf(report_date, "%d.%d.%d", ryear, rmonth, rday);

if(TEST)
    printf("query7: about to open cursor\n");
EXEC SQL OPEN query7_cur FOR READONLY;
if(sqlca.sqlcode != 0) {
    EXEC SQL inquire_sql (:errmsg = errormsg);
    fprintf(qset3msgs, "Open query7_cur failed "
        "with message: %s\n", errmsg);
    EXEC SQL rollback;
    return(-1);
}

if(TEST)
    fprintf(qset3msgs, "\nFollowing are "
        "query7 results:\n\n");
i = 0;
while(1) {
    EXEC SQL FETCH query7_cur INTO :q7result;
    if(sqlca.sqlcode != 0)
        break;
    if(TEST) {
        fprintf(qset3msgs, "%s %s %.2f\n",
            q7result.region_name,
            q7result.estab_name,
            q7result.sum_amount);
    }
    i++;
}

EXEC SQL COMMIT;

gettimeofday(&q7end, (char *) 0);
q7sec = (double) (q7end.tv_sec - q7start.tv_sec);
if(q7end.tv_usec > q7start.tv_usec)
    q7usec = (double) (q7end.tv_usec -
        q7start.tv_usec)/1000000.0;
else {
    q7usec = (double) (1000000.0 - q7start.tv_usec +
        q7end.tv_usec)/1000000.0;
    q7sec -= 1.0;
}
q7time[j] = q7sec - q7usec;
if(TEST)
    printf("query7 q7time[%d] = %.2f\n", j, q7time[j]);
return(0);
;

```

REFERENCES

- [Ammann95] P. Ammann, S. Jajodia, P. Mavuluri, "On-The-Fly Reading of Entire Databases". *IEEE Transactions on Knowledge and Data Engineering*. Vol. 7, No. 5, October 1995, pp. 834-838.
- [Ault96] Michael R. Ault, *Unix System Administrator's Companion*, John Wiley & Sons, 1996.
- [Bober92] P.M. Bober, M.J. Carey, "On mixing queries and transactions via multiversion locking". *Proceedings of the International Conference on Data Engineering*. pp. 535-45. Tempe, Arizona, February, 1992.
- [Bober97] P.M. Bober, M.J. Carey, "Indexing for Multiversion Locking: Alternatives and Performance Evaluation". *IEEE Transactions on Knowledge and Data Engineering*. Vol. 9, No 1, January-February 1997, pp. 68-83.
- [Kosciuszko96] E. Kosciuszko, "Best Practices for Locking and Read Consistency". At: www.pinpub.com/level3/od/od96039.htm (Re: Oracle Read Consistency)
- [Merchant96] A. Merchant, K-L. Wu, P.S. Yu, M-S. Chen, "Performance Analysis of Dynamic Finite Versioning Schemes: Storage Cost vs. Obsolescence". *IEEE Transactions on Knowledge and Data Engineering*. Vol. 8, No. 6, December 1996, pp. 985-1001.
- [Mohan92] C. Mohan, H. Pirahesh, R. Lorie, "Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions". *Proceedings of the*

- ACM SIGMOD conference on the management of data*. San Diego, June 1992, pp. 124-133.
- [Oracle90] Oracle Corporation, *Database Administration Version 6 Class Notes*. Belmont, CA., 1990.
- [Pu88] C. Pu, C.H. Hong, J.M. Wha, "Performance Evaluation of Global Reading of Entire Databases". *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*. IEEE Computer Society Press. Austin, Texas, December 1988, pp. 167-176.
- [Salem94] K. Salem, H. Garcia-Molina, J. Shands, "Altruistic Locking". *ACM Transactions on Database Systems*. Vol. 19, No. 1, March 1994, pp. 117-165.
- [Tansel93] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass, *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings, Redwood City, CA., 1993.
- [Wu93] K-L. Wu, P.S. Yu, M-S. Chen, "Dynamic Finite Versioning: An Effective Versioning Approach to Concurrent Transaction and Query Processing". *Proceedings of the International Conference on Data Engineering*. 1993, pp. 577-86.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved