Design and Implementation of a Transaction Manager for a Relational Database

A thesis submitted for the degree of Master of Science
of The Australian National University

Jason C.C. Lan,

February 1994

Statement

I here by state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

Chih-Chien Lan

Acknowledgements

It is my pleasure to acknowledge the help I have received in developing this thesis and in my studying in the Computer Science Department.

I sincerely thank my supervisor, Prof. Mike Papazoglou for his valuable advice, constant encouragement, and providing incisive comments on enhancing the quality of the thesis. My thanks also go to Dr. Vicki Peterson for her helpful comments and suggestions.

I thank Dr. Brendan McKay for providing the data structure I needed and for answering all my questions, even the stupid ones. I thank Prof. Robin Stanton for creating and maintaining such an atmosphere in the department which makes studying enjoyable.

My special thanks go to my colleague Kerry Taylor for tirelessly reading the manuscript, and providing invaluable suggestions and comments which make the thesis more accurate and precise. I also thank James Popple for his kind help in answering my endless questions about LaTeX.

I am grateful to our programmers and secretaries in the department: David Hawking, Drew Corrigan, Ivan Dean, Steven Ball, Bev Johnstone, Jeannie Haxell, Claire Hinton, Mary Gilmour, and Sally Begg, for their efficient assistance whenever it was requested.

I thank all my friends in the department: William Sun, Jian Yang, Dan Zhou, Rachel Onate, Graham Willians, Ross Parker, Markus Zellner, Dharmendra Sharma, Ely Albacea, Gavin Michael, Robert Cohen, and Mostafa Haghjoo, for their friendship which makes me feel like I was studying at home.

I sincerely thank my parents for their love, encouragement and endless support.

Finally, I would like to express my sincerest thanks to my wife Candy, Chi Ying

Fan for her unlimited support, encouragement and love during the development of this thesis, and to my son Simon, Wei-En Lan. This thesis is dedicated to them.

Abstract

Multi-user database management systems are in great demand because of the information requirements of our modern industrial society. A clear requirement is that database resources be shared by many users at the same time. *Transaction management* aims to manage concurrent database access by multiple users while preserving the consistency of the database.

In this thesis a single-user relational database management system, REQUIEM, is used as a vehicle to investigate improved methods for achieving this. A module, called the REQUIEM Transaction Manager (RTM), is built on top of the original REQUIEM to achieve a multi-user database management system.

The design work of the present thesis is founded upon various techniques for transaction management proposed in published literature which are critically assessed and a mechanism which combines appealing features from existing methodologies.

The problems of transaction management considered in this thesis are:

1. concurrency control,
2. granularity control,
3. deadlock control, and
4. recovery control.

The RTM is also compared with the transaction management facilities of conventional commercial systems such as DB2, INGRES and ORACLE.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Use of database systems has increased dramatically in recent years due to their ease of use and the fast development of the industrial society. Single-user systems, which provide a resource for only one user, no longer meet the demands of database users. A modern database system needs to support sharing of resources by many users at the same time. Such a system is called a multi-user system.

Access to a database is mainly handled by a logical unit of work called a *transaction.* Businesses such as banks and airlines have been moving their transaction processing activities *on-line* since the early seventies. This allows the processing of transactions to proceed while data is being entered. On-line processing is important to banks because the processing of transactions must be concluded as soon as possible, and it is similar to airline reservations.

The on-line databases which are shared by many users have to be maintained up-to-date and data should be stable at all times. This is achieved by a *database management system* (DBMS)[1] which also employs *multiprogramming* principles for sharing data between groups of users and attaining high performance. Multiprogramming allows many application programs to interleave their pattern of execution.

When many programs attempt to simultaneously access a database, interference between them may occur. How to avoid interference between active programs is

[1]The concepts underlying DBMS will be discussed in Section 1.3.1.

known as the *concurrency control* problem. A database could also be damaged due to internal malfunction, such as a hardware crash or software errors, or external damage caused by human mistakes. In such cases, the DBMS should be able to restore the database to the earlier but latest version of consistent state. This is known as the *recovery* problem. Thus, "*transaction management is the task of supervising the execution of transactions in such a way that each transaction can be considered as an all-or-nothing proposition, given the possibility of arbitrary failures on the part of individual transactions or on the part of the system itself, and given also the fact that multiple independent transactions may be executing concurrently and accessing the same data*" [Date 90].

The transaction management is taken care of by a component of the DBMS, called the *Transaction Manager*. A single-user DBMS REQUIEM is used as a vehicle to investigate the improved method for achieving this.

In this thesis, we critically assess various solutions to transaction management that exist in contemporary commercial products or are proposed in literature, and propose an upgraded transaction management methodology for the existing single-user DBMS REQUIEM.

In the following sections of this chapter we first examine the features of general transactions. The aim and approach of the thesis are outlined. Also, the DBMS - REQUIEM, the key system of this thesis, is introduced. Then general problems of transaction management will be discussed together with an indication of the problems considered in this thesis. The structure of the thesis then will be summarized.

1.1 Introduction to Transactions

1.1.1 The Transaction

We commence our introduction by considering some examples. A transaction processing system is a collection of software that processes transactions. Some examples of the systems are:

- banking systems,

- airline reservations systems,
- public library database access systems, and
- vehicle registration systems.

In these systems, transactions can be illustrated in terms of the following examples:

- banking: deposit or withdraw funds to or from an account; transfer funds from one account to another;
- airlines: make or cancel a reservation;
- library access: insert a new book entry or remove a lost book entry to or from the library database;
- vehicle registration: add a new car entry or delete an obsolete car entry to or from the registration.

The common feature of the above examples is that they can modify the database in terms of the transaction. These kinds of transactions are called *write transactions.*

In addition, users may only want to view the information held in the database, such as:

- banking: check the current balance of an account, produce an account statement;
- airlines: find out the available flights for a given destination and time, or find out the number of available seats on a given flight;
- library access: find out the call number of a book from its title;
- vehicle registration: find out the owner and registration expiry date for a given vehicle identified by its registration number.

The above examples are mainly to observe the database rather than change it. These transactions are called *read transactions.*

The division of transactions into two kinds, *read* and *write* transactions simplifies the examination of the conflicts between transactions. Simple *read* and *write* conflicts become major problems to be solved in concurrency control. We will discuss conflicts further in Section 2.2.1.

In all of the above examples, there is a primary database which is shared by several users. Since the database is shared, conflicts may occur due to the interference between numerous read or write transactions. Therefore, the transaction has to guarantee:

- **consistency**: each transaction accesses shared data without interfering with other transactions, and
- **atomicity**: if a transaction terminates normally, all of its effects are made permanent, otherwise it has no effect at all.

That is, transactions must be concurrency controlled (*consistency*) and recoverable (*atomicity*).

1.1.2 Commit and Rollback

In this section, two important operations, *commit* and *rollback*, are introduced. These two operations are the key to provide the atomicity of a transaction.

- Commit - The *commit* operation signals the success of completing a transaction. This is to tell the system that the transaction has been successfully completed and that the database is in a consistent state again. All of the changes made by the unit of work can now be *committed* or made permanent.
- Rollback - A *rollback* operation is required when a transaction is not successfully completed, or has aborted before the time of commit. In this case, all the changes made by this transaction must be *rolled back* or undone.

Commit and *rollback* ensure that a transaction is atomic, since all effects made by the transaction are either made permanent (*commit*), or have no effect at all (*rollback*).

1.1.3 Serializability

Serial execution is an execution of transactions where one transaction is performed at a time. A serial execution preserves database consistency when each individual transaction preserves database consistency. Nonserial execution can change the database to an inconsistent state.

An execution of transactions is said to be *serializable* if it has the same effect as a serial execution on the database. Since a serial execution preserves consistency, a serializable execution also preserves consistency. One goal of the transaction management is to order the operations executed by different transactions so that the resulting execution is serializable and recoverable.

1.2 Thesis Aim and Approach

The aim of this thesis is to develop a multi-user version of a single-user relational DBMS, namely REQUIEM. This is done by adding a separate module, named REQUIEM Transaction Manager (RTM), to the original REQUIEM version. One characteristic of this RTM design is that it is based on *modularity* principles, that is, isolating design decisions and decomposing the system into modules which hide these decisions, resulting in a much more flexible design.

Two main aspects of the RTM are *concurrency control* and *recovery control*.

- Concurrency control - this is to ensure that transactions run by different users will not interfere with each other and lead to database inconsistency. For example, one might try to update the data which another user is trying to delete from the same database at the same time. This produces one of the concurrency control problems, namely the *lost update* problem[2]. Concurrency control makes data sharing transparent to each user.

- Recovery control - the database can become inconsistent due to a failure occurring in the computer system. The failure may be caused by human error,

[2]More detailed concurrency control problems will be discussed in Section 1.4.1.

e.g. typing mistakes, or a crash in the computer system itself. In such cases, the DBMS should be able to restore the database to the latest version of a consistent state according to the type of failure that occurs. We will detail the types of failures later in Section 1.4.2.

Although RTM is implemented on the relational DBMS, REQUIEM, the concepts underlying RTM are applicable in general and could be used in either relational or non-relational database systems.

The design principles of the RTM are motivated by the existing transaction management methodologies proposed in literature. The RTM is a better transaction manager than the others because the mechanisms used to solve each component problem of the transaction manager are comparatively better than others currently available. A comparison between RTM and other transaction managers will be made in Chapter 5.

1.3 A Relational Database Management System - REQUIEM

1.3.1 The Database Management System

As indicated in [Date 90], "*a database system is essentially nothing more than a computerized record-keeping system*". The database can be regarded as an electronic cabinet which stores a collection of computerized data files. The task of a database management system is to maintain information and to make that information available on demand. Moreover, the DBMS should be able to provide reliable data to the user, in the presence of concurrency and recovery problems.

1.3.2 The Relational Database

A relational database is one in which the data and the relationships among data are represented by a collection of tables each of which has a number of columns

with unique names. Moreover, a number of operations allow the user to extract a table from old tables. We demonstrate the relational database by the well known example, namely the *supplier-and-product* database, as shown in Table 1.1. This

SUPPLIER

SUPID	SUPNAME	LOCATION
605	Jones	London
855	Dick	Bonn

PRODUCT

PRODNO	PRODNAME	WEIGHT	FINISH
21	cabinet	80	oak
25	soft	20	oak
33	dresser	90	pine

SUPPLY

SUPID	PRODNO	QUANTITY
605	21	150
855	25	130
605	33	100

Table 1.1: The Supplier-and-Product Database

example consists of three tables, namely *supplier*, *product*, and *supply*.

- Table *supplier* represents suppliers. Each supplier has a unique supplier number (*supid*), a supplier name (*supname*), and a location (*location*). This assumes that each supplier is located in exactly one location.

- Table *product* represents products. Each kind of product has a unique product number (*prodno*), a product name (*prodname*), a weight (*weight*), and a finish (*finish*).

- Table *supply* represents the relationship between suppliers and products. For example, the first row of table *supply* in Table 1.1 shows that product with product number 21 is supplied by the supplier, whose supplier number is 605,

with the quantity of 150. This assumes that there can be at most one supply at any time for a given supplier and a given product. Thus the combination of *supid* value and *prodno* value is unique with respect to the set of supply currently appearing in the table *supply*.

Most relational database systems provide a query language for access to the database. We introduce the most popular query language called SQL to illustrate how data in the relational database can be accessed by users.

SQL is a successful experimental relational language which was developed under IBM System R. Now many commercial database systems are not only relational, but they also support a dialect of the relational language SQL. The following are examples of SQL language using the above *supplier-and-product* database.

Example 1.3.1 *To find all product number and quantity of products which are supplied by the supplier with supplier number 605.*

The SQL query is expressed as follows:

```
SELECT PRODNO, QUANTITY
FROM SUPPLY
WHERE SUPID = '605';
```

Result table is shown below:

PRODNO	QUANTITY
21	150
33	100

Table 1.2: The Result Table of Example 1.3.1 Query

Example 1.3.2 *Get supplier number and supplier name for all suppliers in London.*

The SQL query statement is expressed as follows.

SELECT SUPID, SUPNAME

FROM SUPPLIER

WHERE LOCATION = 'London';

Result table is :

SUPID	SUPNAME
605	Jones

Table 1.3: The Result Table of Example 1.3.2 Query

1.3.3 REQUIEM

REQUIEM (RElational Query and Update Interactive SystEM) is an extensible, relational DBMS developed in the C programming language by Mike Papazoglou and Willy Valder [Papazoglou and Valder 89] in the late 80's. REQUIEM was developed as a research and educational tool and has been used to investigate various issues in database design. The system offers an open-ended interface facility so that modules such as a *transaction manager* can be incorporated without much difficulty. One of the goals of the work described in this thesis was to research existing concurrency control and recovery methodologies and to propose an upgraded transaction management for REQUIEM.

1.3.4 Access to REQUIEM Database

The REQUIEM query language, called RQL (Relational Query Language), is a relational dialect which presents many similarities with the query language SQL. A significant feature of RQL, is that it can be used *interactively*, as an *embedded language*, and worked through the use of *external files*. RQL is available at two different interfaces, namely an *interactive interface* and an *application programming interface*.

When the REQUIEM interactive interface is invoked, the RQL statements can be entered and executed immediately. The results will be displayed right after the execution of the RQL statements. Also, the query language can be embedded in an application language program such as a C language program. In this case, the query statements will be executed while the program is executed, and the result will be bound into the program defined variables. Alternatively, in REQUIEM, RQL statements can be put in an external file called a *command file*. In such case, many statements can be executed at once. A command file which includes RQL statements may be executed in either interactive or non-interactive way as shown in Figure 1.1. We will discuss them in terms of some examples in the following sections.

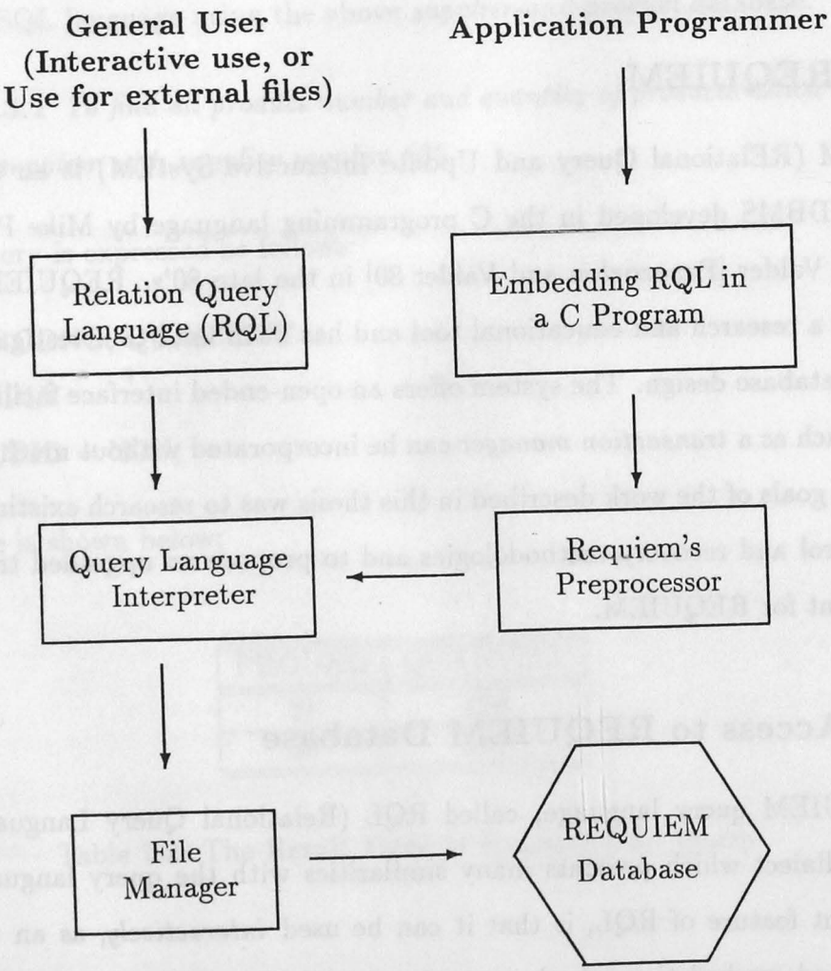


Figure 1.1: Accessing the REQUIEM Database

Interaction with REQUIEM Database

The main method of accessing to the REQUIEM database is interaction with the database via various query language statements supported by the DBMS. When the REQUIEM interactive interface is invoked, the prompt "REQUIEM:" is displayed at a terminal (or a *window* on a workstation) and the system waits for the input of RQL statements. Once the statements are input, they are processed by the query language interpreter which sends requests to the *file manager* to perform the appropriate operations. Then the resulting table (or return message) will show immediately. The following is an example of interactive RQL statements.

The *supplier-and-product* database in Table 1.1 is used again to illustrate RQL statements.

Example 1.3.3 *Find the name and location of a supplier, whose supplier number is 605:*

select supname, location **from** supplier **where** supid = "605";

This query can be entered at the REQUIEM interface when REQUIEM is invoked. REQUIEM will respond with the result table as shown in Table 1.4.

SUPID	SUPNAME	LOCATION
605	Jones	London

Table 1.4: The Result Table of Example 1.3.3 Query

Embedded RQL Language

In addition to providing a query language, REQUIEM also offers users the option of writing application programs in a general purpose programming language. The same RQL statements, with slightly modified syntax [Power 91], may be embedded in the C program, the host language of REQUIEM, by prefixing them with the keywords *EM RQL*, indicating the start of an *EM*bedded *RQL* statement, and terminated by

a semi-colon, which is how RQL statements are normally terminated in REQUIEM. Further, a program variable in an embedded RQL statement is identified by prefixing the variable name with a colon. The embedded queries can be replaced with the set of corresponding *Programmable Procedural Interface*[3] routines by *REQUIEM's Preprocessor* (RPP), then passed through the *query language interpreter* and *file manager* to retrieve data from the database. The same example as in Example 1.3.3 can be written in the embedded form in the following syntax:

EM RQL **select** supname :Sup_name, location :Location **from** supplier
where supid = "605";

In the above statement, *Sup_name* and *Location* are host language defined variables. The result values of *supname* and *location* will then be assigned to *Sup_name* and *Location*, respectively.

Use of External Files

REQUIEM can also read and process a series of valid RQL statements from a *command file*. The user can process the statements included in the command file in two ways [Papazoglou and Valder 89]:

1. The command file can be processed from within RQL by typing a "@" character before the command file name.

2. Alternatively, the name of the file can be passed as a command-line argument when the user invokes REQUIEM.

The first way is used interactively with REQUIEM interface. Some query statements are quite long, so that it is not convenient to use them interactively. The easy way to run the long query statements is to put them in a file. Instead of typing the whole statement, a short name of the command file can be used. For example, the following statements are stored in a file named *create_supplier*.

[3] *Programmable Procedural Interface* provides an interface to REQUIEM that allows C programs to utilize embedded queries.

```
create supplier (supid (num(5), UNIQUE),
                 supname (char(20), SECONDARY),
                 location (char(20)));
```

This is an RQL statement to create the table *supplier*. Since the statements are long enough to make a typing mistake likely, it is better to store them in the file and to type "@create_supplier" rather than to type the whole statements to create a file. All the statements in the table then will be executed and responded to after the command file is processed.

The second way is that a command file is executed as an application program, except that the content of the command file is purely RQL statements. These provide a more convenient way to deal with some transactions, such as inserting a large amount of data into the database. More information can be found in [Papazoglou 89].

1.3.5 Transactions in REQUIEM

Transactions can be formally defined in different ways, as long as consistency and atomicity are preserved. If a transaction processes without producing any errors, the transaction will *commit* and put the changed data into the database permanently. Once a transaction commits, the change made to the database can not be undone. If a transaction terminates before it commits, it is called an *aborted* transaction.

A transaction is a unit of work to access the database. It can be expressed either explicitly or implicitly to point out the start and end of a transaction. Some DBMSs, such as System R [Gray *et al* 81], issue explicit statements like *begin_transaction* to start a transaction and *commit_transaction* to commit the transaction in the application programs. The transaction is a user defined logical unit of work. However, the concepts of a transaction may apply equally to the end user environment. For example, DB2[4] provides the user with the ability to enter SQL statements interactively through DB2I[5] [Date 90]. In this case, each such interactive SQL statement is treated as a transaction in its own right; DB2I will automatically issue a COM-

[4]DB2 is a commercial relational database product developed by IBM.
[5]DB2I is the interactive interface for DB2.

MIT on the user's behalf after the SQL statement has been executed. This kind of transaction is essential since generally a query statement does not contain a *begin* or *commit* command.

A transaction in REQUIEM is defined in terms of an RQL language statement, a file command, and an embedded query statement as described in Section 1.3.4. All the following examples are valid transactions in REQUIEM.

- A single RQL query statement such as:
  update location **from** supplier **where** supname = "Jones";
- An embedded RQL query statement such as:

  EM RQL **select** supname :Sup_name, location :Location **from** supplier
  where supname = "Dick";
- A command file such as *create_supplier* described in Section 1.3.4.

A transaction starts when the first RQL statement is encountered. REQUIEM will issue a *commit* automatically when the execution comes to the end, such as a ";" is executed at the end of the query statement or the end of a command file. The transaction will then be committed, otherwise, the transaction is aborted. The transaction ends when any result is returned. It is natural to define these as *transactions* since each of them is treated as a complete task in REQUIEM.

1.4 General Problems of Transaction Management

General problems of transaction management are considered in this section. They are concurrency and recovery problems.

1.4.1 General Concurrency Control Problems

Without any concurrency control, various problems may arise in a multi-user DBMS due to interference between different programs running on the database at the same

time. In this section, we describe the concurrency problems that may affect database consistency.

Concurrency problems are usually caused by the *interference* of transactions. Although some of the aborted transactions can cause database inconsistency, even if all the transactions are correct themselves, the interleaving of them may also damage the database. The *interleaving* of operations from transactions is the main cause of the problem. There are four problems that may result from the interference of transactions [Date 90]:

1. The *lost update* problem: Assume transaction A and transaction B read the same record R, containing a data item X with value 100, at the same time. Then transaction A increase X from 100 to 150, and after that transaction B increase X from 100 to 120 before both transactions commit. Then the update of X by transaction A is *lost* since transaction B overwrites it without even looking at it.

2. The *uncommitted dependency* problem: A transaction A updates a data item X in record R , from 100 to 150, and then transaction B reads the value of X which is 150. If transaction A is aborted at this time, the value of X read by transaction B is an *uncommitted* value and should not be allowed to be read.

3. The *inconsistent analysis* problem: Assume both transaction A and B are operating on bank account records. Transaction A is summing the account balances and transaction B is transferring from one account to another. At first, transaction A sums the balance account 1 X, say 40, and the balance of account 2 Y, say 50, to the sum of 90 to TOTAL. Then transaction B transfers the amount of 10 from account 3 to account 1. If the balance of account 3 Z was 30, the new balances of account 1 and 3 would be 50 and 20 respectively. After transaction B commits, transaction A reads Z, which is 20, and TOTAL is changed to 110. However, this is the wrong answer, since the sum is supposed to be 120. This is wrong due to the *inconsistent analysis* of transactions.

4. The *inconsistent table* problem: another problem may arise because of a transaction updating a table simultaneously with another deleting the table.

It is clear that if many users are trying to access the data at the same time, one way of guaranteeing database consistency is to make each person wait his turn, that is, to make the execution of transactions *serial* as described in Section 1.1.3. Therefore, if the transaction manager can manage transactions as *serializable*, which has the same effect as *serial* execution, then the concurrency control problems can be solved.

1.4.2 General Recovery Control Problems

Recovery control aims to *recover* the transaction from different kinds of failures which may occur and which affect the database consistency during transaction processing. In order to solve such problems, one must know precisely which types of failures are to be considered and how often will they happen. Generally, the following types of failures should be considered [Gray *et al* 81]:

- *Transaction failure.* As described in Section 1.1.2, the outcome of a transaction is either *commit* or *abort* (*rollback*). Although all of the transactions processed are expected to commit, there is always a possibility that the transactions will be aborted. This might happen due to internal conditions such as bad input, data not found, overflow, or resource limit exceeded, etc.

- *System failure.* The system failures can be caused by a bug in the DBMS code, an operating system fault, or a hardware failure. In such cases, the transactions are terminated unexpectedly, and the contents of main (or volatile) memory are lost. As a consequence, the database may remain in an inconsistent state due to the aborted transactions.

- *Media failure.* A media failure is a failure in which some portion of the database has been physically destroyed. This could be caused by a disk head crash, a disk controller failure, magnetic decay, etc.

A good transaction management should be able to recover from all the types of failures that occur in a system.

1.5 Problems Examined in the Thesis

We first point out the various problems of transaction management which will be examined in this thesis. Then we investigate the existing solutions for each problem and build up a firm foundation solution for the RTM. The aim of investigating existing implementations is to examine how various problems of transaction management can be resolved. Since each transaction management of those compared systems are different but have something in common, a set of problems are examined which allows them to be fairly compared. These problems are outlined below.

The first problem considered here is how a *transaction manager* can be merged seamlessly with the original DBMS design. A good transaction manager design should be able to hide the decisions from the main system and to avoid affecting the performance of the main system. That is, the design should be general and independent, so that one may easily employ the same design abstract on different systems. This aspect of the RTM design is consistent with the design of REQUIEM and no comparison with other systems in this aspect is made here.

The second problem considered here is that of the concurrency control mechanism to be used. Locking is the most popular mechanism to solve the concurrency control problem in most of the relational DBMSs. A transaction has to ask for the privilege (lock) to read or write a data before it is going to do so. If another transaction owns the conflict privilege (conflict lock) to the data, it has to wait until the privilege is granted by the DBMS. However, locking has a drawback of its own. Generally, *read* transactions can be arbitrarily large so considering them in concurrency control interactions may cause large overheads. Therefore read transactions which do not modify the database, should be possible to run without any interaction with concurrency control. Optimistic concurrency control is developed for this reason. In this method, the transactions can access the database without

any control. Serializability is guaranteed by a *validation* step performed at the end of each transaction. If a conflict action has occurred the transaction is aborted and restarted. However, abort and restart transactions caused by optimistic concurrency control may be very expensive. Locking is useful for solving this problem in the sense of *pessimistic concurrency control.* That is, *read* transactions should run without any locking control (optimistic), and *write* transactions should be controlled by the locking (pessimistic).

The third problem is the *granularity problem.* The granularity of data item, which refers to the size of the data to be locked, is important as far as performance is concerned. In the relational model of data, large items, like the whole tables, or small items, like individual records, can be chosen as the granularities to be locked. Large granularity locks reduce concurrency, since operations are most likely to conflict. On the other hand, overhead is decreased because fewer locks are managed. Small granularity locks involve higher locking overhead, since more locks are required, but they improve concurrency by locking only those data items the transaction accesses. Choosing the lock granularity presents a tradeoff between locking overhead and the amount of concurrency. One approach "is to attempt to design the database so that large transactions are rare" [Robinson 82] and simply use the small granularity locks. This approach is usually in the design at higher levels, and is not considered here. Another approach is *hierarchical concurrency control* ([Gray 75], [Gray *et al* 76] and [Gray 78]), also known as *multiple granularity locking* [Bernstein 89] and is used in RTM. This is to use a kind of scaling granularity most appropriate to each transaction's mode of operation. We propose a *query analysis module (QAM)* to *predict* the behaviour of the transactions as much as possible and make the granularity choosing decision appropriate to each transaction. This is not achieved by any other existing DBMS systems.

Since locking is used, *deadlock* may arise. When the data is *locked* by a transaction, another transaction has to wait until the lock is released if it wants to lock the data and update the data. Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each one waiting for one of the others to release a

lock before it can proceed.

The fourth problem examined in this thesis is that of *recovery control.* A DBMS should be able to recover database consistency in the event of failures as described in Section 1.4.2. Recovery from three types of failure, transaction, system, and media failures are all considered.

1.6 Structure of the thesis

The problems of transaction management considered here have been addressed above. The next chapter examines the solutions of these problems published in literature and discusses the methodologies related to each problem.

Chapter 3 contains a design framework for the RTM based on the features discussed in Chapter 2.

Chapter 4 details design of the system architectural of the RTM and describes the communication between components of the system.

Chapter 5 describes the transaction management mechanisms used in conventional relational database management systems, such as DB2, INGRES, and ORACLE, and presents a comparison of the transaction management of RTM with these RDBMSs.

Chapter 6 presents a detailed description of the RTM implementation.

Chapter 7 draws conclusions of the transaction manager developed in REQUIEM and discusses the direction of future work of this thesis at the end of this chapter.

Chapter 2

A Survey of Existing Work

The RTM is founded upon a mechanism which combines appealing features of existing methodologies. In the previous chapter, we briefly described the problems of transaction management concerned in this thesis. This chapter investigates published solutions to these problems . These four problems of transaction management which were described in Section 1.5 are used in this chapter for investigation. They are:

1. Concurrency control,
2. Granularity control,
3. Deadlock control, and
4. Recovery control.

Related work published in literature since the early seventies is discussed here. The solutions to the four problems listed above are examined in turn in the following sections. The discussion of the methodologies related to each problem are also presented.

2.1 Centralized vs Distributed DBMSs

This section briefly describes *centralized* and *distributed* DBMSs. Roughly speaking, a centralized system consists of a central processor, main memory, secondary storage

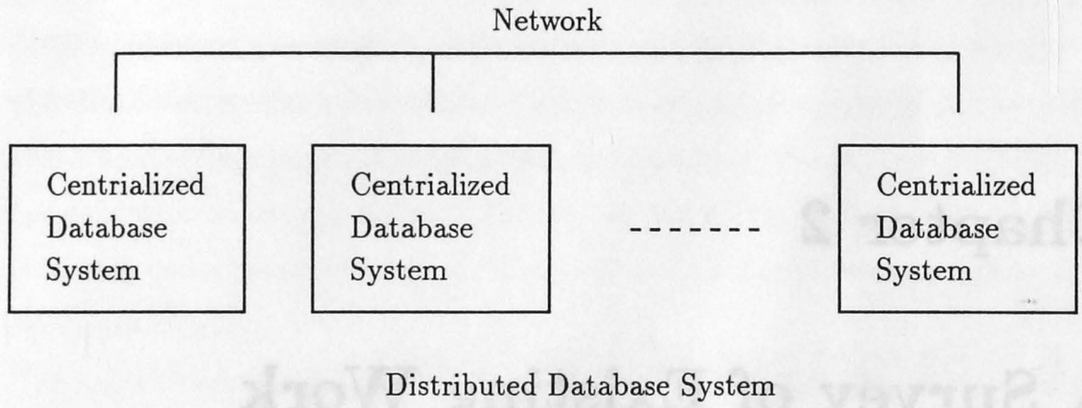


Figure 2.1: Centralized and Distributed Database Systems.

devices (usually disks), and I/O devices. In addition, a multiprocessor configuration in which each processor has direct access to all of main memory and to all I/O devices is also a centralized system. A system with two or more processors that do not have direct access to shared main memory or secondary storage devices is called a *distributed* system. Figure 2.1 shows the relationship between centralized and distributed systems.

2.2 Concurrency Control

Various concurrency control mechanisms have been invented since the early seventies. They fall into four categories: *locking*, *optimistic*, *timestamp ordering*, and *multiversion* concurrency control. However, various combinations of these mechanisms are also possible. We briefly describe them in turn in the following sections.

2.2.1 Locking

The most widely accepted mechanism used for concurrency control is *locking*. Locking is derived from the operating system method for allocating resources to tasks [Valduriez and Gardarin 89]. The data may be treated as resources that are allocated (locked) or deallocated (unlocked) to transactions. Locking oriented mechanisms for databases can be considered *pessimistic* [Menascé and Nakanishi 82] since

database resources are locked even though transactions may not conflict with other executing transactions. That is, they may prevent transactions from executing concurrently, even though no interference may occur.

For locking, sharing is usually controlled by dividing the database into locking units or granularities, and by specifying a locking policy to ensure integrity of the database. An analysis of locking policies in DBMSs can be found in [Potier and Leblanc 80]. Another comparative analysis between *two-phase* and *non-two-phase* locking algorithms for DBMSs is given in [Joergensen and Singhal 87].

Two-phase locking (2PL), one of the locking policies proposed by K.P.Eswaran, J.N.Gray, R.A.Lorie and I.L.Traiger [Eswaran *et al* 76], guarantees serializability. This mechanism has been used to deal with the concurrency control problem in most relational database management systems [Valduriez and Gardarin 89]. A brief description of two-phase locking is given below.

Theorem 2.2.1 *From P.A.Bernstein, V.Hadzilacos and N.Goodman [Bernstein 89]. Two-phase locking is achieved if all transactions obey the following rules:*

1. *before operating on any object the transaction first acquires a lock on that object; and*
2. *after releasing a lock the transaction never acquires any more locks;*

then all interleaved executions of those transactions are serializable.

A transaction that obeys conditions (1) and (2) of the theorem is said to be two phase, i.e., to satisfy the two phase locking protocol. The proof of the theorem relies on the serialization ability of two-phase locking. The proof is given in [Eswaran *et al* 76] and [Bernstein 89]. P.A.Bernstein and N.Goodman [Bernstein 89] have provided more details on two phase locking theory and some implementation issues. Serializability has been discussed in Section 1.1.3 and the related theory can be found in [Eswaran *et al* 76], [Bernstein 79], and [Papadimitriou 79].

In general, two-phase locking consists of a *growing phase*, during which locks are acquired, and a *shrinking phase*, during which they are released. If all transactions are two-phase, then all executions are serializable. The condition that all transactions be two-phase is sufficient but not necessary to guarantee serializability. The

2PL theorem provides guidelines for the safe design of transactions. The *transaction manager* ensures that all the transactions it receives follow the two-phase locking protocol.

The database is partitioned into items, such as records, which are portions of the database that may be locked. A transaction can prevent other transactions from accessing an item by locking the item until the transaction holding the lock releases the lock on the item.

Generally, two types of lock modes, *read* (or *share*) and *write* (or *exclusive*) locks are used in locking. Read locks on the same data items can be shared. Write locks on the data items can only be acquired by one transaction at a time. The compatibility matrix for the two lock types in Table 2.1 shows how different locks can be acquired in the event of conflicts. A "y" (yes) entry means that two locks of

	read	*write*
read	y	n
write	n	n

Table 2.1: The Compatibility Matrix.

the types specified by the row and column labels do not conflict. The only "y" in the table specifies that *read* lock does not conflict with itself. This means that users can acquire as many *read* locks as they like on the same data without any conflict. On the other hand, an "n" (no) entry means that the lock types conflict. The other "n" in the table shows that when a *read* or *write* lock has been acquired on one data item, no one can acquire *write* lock on the same data. The design of two phase locking is safe in the sense that only one user can change the data each time.

Two phase locking represents an overhead that is not present in the sequential case. Even read-only transactions (queries), which can not affect the integrity of the data, must in general use locking in order to guarantee that the data being read are not modified by other transactions at the same time.

Non-two-phase locking, also known as *tree locking*, is used upon the assumption that data items are stored as nodes in a tree structure, and transactions always

access data items by following paths in the tree [Bernstein 89]. Since this is not the case in REQUIEM, it is not discussed here.

2.2.2 Optimistic

Another mechanism which is called *optimistic* is proposed by Kung and Robinson [Kung and Robinson 81]. The optimistic mechanism works on the assumption that conflicts between transactions are rare. It does not consider access conflicts when they occur, as in the locking approach. Instead, detection of transaction conflicts is delayed until the transaction commit time, and upon conflict the changes made by the transaction are undone. The principles of this method are summarized in the following [Kung and Robinson 81].

1. Since reading the data from the database can never cause a loss of integrity, *reads* are completely unrestricted.

2. *Writes* are severely restricted. It is required that any transaction consist of two or three phases: a *read phase*, a *validation phase*, and a possible *write phase* (see Figure 2.2).

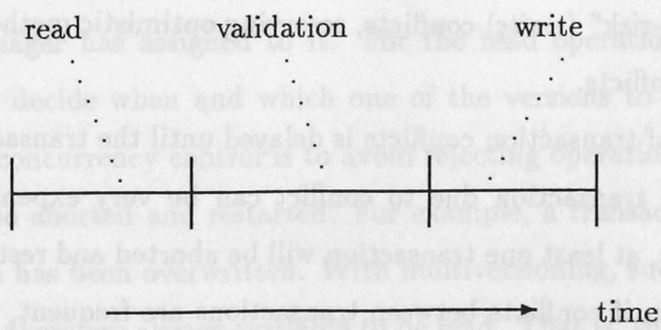


Figure 2.2: The Three Phases of a Transaction.

- read phase: During the read phase all changes take place on local copies of the objects being modified; the read phase corresponds to the execution time frame of the body of the transaction.

- validation phase: During the validation phase it is determined whether a transaction will cause a loss of integrity and whether the result of a query is still correct.
- write phase: During the write phase of a transaction the data structures of the database are actually updated, i.e. the local copies of the database are made global. In the case of a query (read-only), it must be determined that the result the query would return is actually correct.

These three phases are described below. If the validation does fail, the transaction will be undone, i.e. the local copies will be removed, and the transaction commenced again. Thus a transaction will have a write phase only if the preceding validation succeeds. Observe that queries are subject to validation, although a write phase is missing. A detailed description of an implementation of the optimistic concurrency control method can be found in [Kersten and Tebra 84].

A performance comparison of optimistic and pessimistic (locking) concurrency control mechanisms in DBMSs is presented in [Menascé and Nakanishi 82].

Optimistic techniques are suitable for resolving the infrequent conflicts between transactions, but poorly suited to the more frequent conflicts between transactions. A mixed scheme could exploit the strengths of each method by using pessimistic techniques for "high-risk" (*write*) conflicts, reserving optimistic methods to proceed "low-risk" (*read*) conflicts.

Since detection of transaction conflicts is delayed until the transaction commits, the abortion of the transaction due to conflict can be very expensive. That is, when conflict occurs, at least one transaction will be aborted and restarted as a new transaction. Further, if conflicts between transactions are frequent, the cost of the abortions is relatively increased and performance is impaired.

2.2.3 Timestamping

Timestamp ordering concepts can be found in [Ritchie and Thompson 74] and [Orji 88]. In timestamping [Bernstein 89], serialization order is determined by the unique timestamp assigned to each transaction before it starts. Each operation carries the

timestamp of its transaction. For each data item accessed, there is a *read timestamp* and a *write timestamp* to record the timestamp of the latest transaction read or write, respectively. Any write operation of timestamp which is earlier than the data's read timestamp is rejected. That is, the transaction is aborted and restarted. The restarted transactions will get new and later timestamps.

C.U.Orji, L.Lilien and J.Hyziak [Orji 88] have compared the performance of an optimistic and a basic timestamp ordering concurrency control algorithm for centralized database systems and proved that, in most cases, the optimistic is better than the timestamp-ordering mechanism in efficiency.

2.2.4 Multiple Version

A theory and algorithms of multiversion concurrency control are described in [Bernstein and Goodman 83]. In a multiversion concurrency control algorithm, each write operation on a data item produces a new version of it. The transaction manager then keeps a list of versions of the data item, which is a history of values that the transaction manager has assigned to it. For the read operation, the transaction manager has to decide when and which one of the versions to read. The benefit of multiversion concurrency control is to avoid rejecting operations and causing the transaction to be aborted and restarted. For example, a transaction may read the data item which has been overwritten. With multiversioning, such values are never overwritten and therefore always available to be read. That is, the transaction manager can simply provide the old version of the data item for the transaction to avoid rejecting the transaction.

A drawback of multiversion concurrency control is the cost of storage space since the transaction manager may have to keep many versions of the same data items. Further, if the DBMS has to keep tracing and purging the useless versions of data, it could impose additional overhead and complexity on the system.

2.2.5 Various Combined Methodologies

The concurrency control mechanisms mentioned above may also be used by mixing different methods together. M. Herlihy [Herlihy 86] proposed mixing pessimistic (locking) and optimistic methods. In [Kim and Lochovsky 89] and [Lausen 82], different ways of mixing these two methods are presented. A general description of various combinations of different concurrency control mechanisms can be found in [Bernstein 89].

Nevertheless, the compatibility of different mechanisms is a non-trivial problem, for example, two-phase locking and multiversion timestamping cannot be used together in a multi-user system [Herlihy 86].

2.2.6 Discussion

The following are the major features which have come to light from the survey of the existing concurrency control mechanisms.

1. Locking provides a safe guideline for concurrency control design, but presents an overhead that is not present in the sequential case. Read transactions, which will not possibly affect the integrity of the data, should proceed without locking to provide more concurrency in the transaction processing.

2. Optimistic concurrency control favours the read transactions to proceed without any interruption. In the case of write conflicts, the cost of the abortion and restart of transactions may be unacceptable.

3. Timestamp ordering is generally found not efficient ([Orji 88] and [Date 90]) in the DBMS especially in a centralized DBMS. Since a centralized multi-user DBMS is developed here, timestamp ordering is not considered.

4. Multiversion concurrency control also provides a better solution for read transactions. Nevertheless, the cost of storage and garbage collection may be too costly for a DBMS.

2.3 Granularity Control

The granularity of a data item refers to the size of the item. In the relational model of data, we can choose large items, like relations (or tables), or small items like individual tuples (or records). Also, items could be of an intermediate size, say 50 tuples from a relation.

The granularity of data items is important for performance but irrelevant for correctness. Suppose locking as described in the previous section is used. Large or coarse granularity locks reduce concurrency, since operations are more likely to conflict. On the other hand, overhead is decreased because fewer locks are managed. Small granularity locks involve higher locking overhead, since more locks are required, but they improve concurrency by locking only those data items the transaction accesses.

Choosing the granularity of locks presents a tradeoff between locking overhead and the amount of concurrency. It would be desirable to use different granularity according to what is most appropriate to each transaction's mode of operation. That is, large transactions, which access many tuples can lock coarse granules, say relations. Small transactions can lock fine granules, say tuples, in case only few tuples are accessed. Therefore, long transactions do not waste time setting too many locks, and short transactions do not bother others by locking those data items they do not access.

2.3.1 Multiple Granularity Locking

One solution to the granularity control is *multiple granularity locking (MGL)* which was originally proposed by J.N. Gray, R.A. Lorie and G.F. Putzolu [Gray 75][1], and are also examined by Bernstein, Hadzilacos, and Goodman [Bernstein 89]. Other related description of MGL are examined in [Gray *et al* 76], [Ries and Stonebraker 79], and [Korth 83]. The granularity of the data item may either be *logical* or *physical*. In a database, the term *logical data* refers to the data which are viewed by the ex-

[1] In [Gray 75], *multiple granularity locking* is described as *dynamic locking*.

ternal users, and *physical data* refers to the data which are actually stored in the computer system. For logical data, the most appropriate granularity to be locked are relations and tuples, i.e. files and records. Page, a unit of space in memory, is an appropriate granularity lock for physical locking. Figure 2.3 shows the relationship between these different granules. Each level of the diagram is given a node type

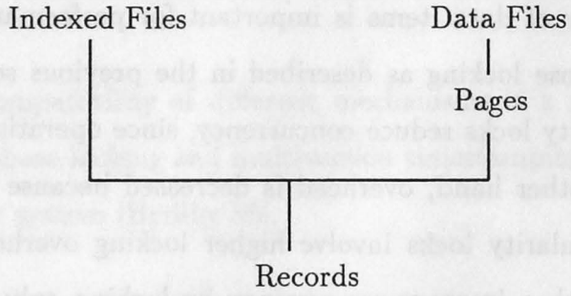


Figure 2.3: The Three Level Hierarchy.

which is a generic name for all the node instances of that type. Each node, either *indexed files, data files, pages* or *records*, of the diagram can be locked. The task of choosing coarse or fine granularity to be locked is generally achieved by the *lock escalation* which will be described in Section 2.3.2.

Multiple granularity locking (MGL) also requires the transaction manager to prevent two transactions from setting conflicting locks on two granules that overlap. If one requests a write lock to file node, then when the request is granted, the requester has write access to that node and *implicitly* to all its descendant records. If one requests a read lock to file node, then when the request is granted, the requester has read access to that node and *implicitly* to all the descendant records of that node. The two access modes lock all the records of the files.

A lock on a file node which *explicitly* locks the node also *implicitly* lock all its descendant records. On the other hand, it is also necessary to have the effect of record locking to the file that contains the records. That is, when a record is locked by a transaction, then the file node that contains the record can not be *explicitly* locked by other transactions. To achieve this, *ir* (*intention read*) *locks* and *iw* (*intention write*) *locks* with *r* (*read*) and *w* (*write*) *locks* are introduced. Before

the *transaction manager* locks the record x, it must ensure that there are no locks on its ancestor files that implicitly lock x in a conflicting mode. To do this, it sets intention locks on those files. The compatibility matrix for the four lock types is as shown in Table 2.2. The LM sets and releases locks on data granule x for each

	r	w	ir	iw
r	y	n	y	n
w	n	n	n	n
ir	y	n	y	y
iw	n	n	y	y

Table 2.2: The Compatibility Matrix for *Multiple Granularity Locking.*

transaction T according to the following *MGL* protocol [Bernstein 89]:

1. If x is a record, then to set r *lock* on x, T must have an ir or iw lock on its ancestor files.

2. If x is a record, then to set w *lock* on x, T must have an iw lock on the ancestor file of x.

3. A read (or write) lock on the record x itself is an *explicit lock* for x; a lock on the files which contain x is an *implicit lock* for x.

4. A transaction may not release an intention lock on a file x, if it is currently holding a lock on any record which belongs to x.

Rules (1) and (2) imply that to set r or w lock on a record, T must first set the appropriate intention locks on those ancestor files of the record. Rule (3) implies that by locking a file, T has implicitly locked all of its descendant records. This implicit locking frees the transaction from having to set explicit locks on all the records of a file, which is the main reason for MGL. Rule (4) shows that a transaction never owns a lock on the record without owning the corresponding intention locks on ancestor files of the record.

2.3.2 Lock Escalation

Lock escalation is used to support the MGL [Bernstein 89]. The lock escalation means that the transaction manager starts requesting small granularity locks one by one as it receives operations from the transaction. If a transaction obtains more than a previous fixed number of locks of fine granularity, then the transaction manager starts locking the high level of granularity. That is, it escalates the granularity of the locks it requests. However, lock escalation can cause a deadlock to occur. For example, suppose two transactions are holding *iw* locks on a file and are setting *w* locks on records, one by one. If they both escalate their record locking activity to a file lock, they will both try to convert their *iw* lock to a *w* lock. The result is a deadlock. This can be detected by deadlock checking[2]. When deadlock occurs, the transaction manager simply aborts the transaction which causes deadlock.

2.3.3 Fixed Granularity

Instead of using MGL methodology to choose the appropriate granularity for locking, fixed granularity is used in some other systems such as SABRINA[3] and FOCUS[4] [Valduriez and Gardarin 89]. Some DBMSs are designed to have short or long transactions all the time, thus either small or large granularity may be chosen by the systems. An example can be found in [Robinson 82]. However, this is not suitable for a general purpose multi-user database system since systems normally have both short or long transactions.

2.3.4 Discussion

For a general purpose DBMS transaction management, the MGL is clearly the better solution than the fixed granularity. However, the lock escalation which supports the

[2]The deadlock control is described in Section 2.4.

[3]SABRINA is an RDBMS developed in the SABRE project at INRIA, France in the early eighties. The SABRE project tried to attack problems not solved completely by the System R and INGRES.

[4]FOCUS is marketed by Information Builders Inc. and is a popular commercial system that includes a fourth-generation system and a database system.

MGL could cause deadlock easily and impacts the transaction management.

2.4 Deadlock Control

Whenever a transaction waits for a request to be granted, it runs the risk of waiting forever in a *deadlock* cycle, in which two or more transactions are in a simultaneous wait state, each one waiting for one of the others to release a lock before it can proceed. If a deadlock occurs, the system should detect it and break it.

2.4.1 Deadlock Detection

One way of solving the deadlock problem is *deadlock detection* [Bernstein 89]. Detecting deadlocks involves detecting a cycle in the *wait for graph (WFG)*. The nodes in WFG are labelled with transaction names. If transaction T_i is waiting for transaction T_j to release some locks, then an edge $T_i \rightarrow T_j$ from node T_i to T_j is added to the WFG. Suppose there is a cycle in WFG such as: $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. Each transaction is waiting for the next transaction in the cycle and therefore T_1 is waiting for itself. This is a deadlock since none of the locks they are waiting for will be released in the cycle. The transaction manager should be able to detect the deadlock by checking for cycles in WFG [Holt 72]. Breaking the deadlock involves choosing one of the deadlocked transactions, i.e. one of the transactions in the cycle in the WFG, as a *victim* and rolling it back, thereby releasing its locks and allowing some other transactions to proceed.

Another way of detecting a deadlock is *timeout* [Bernstein 89]. If the transaction manager finds that a transaction is waiting too long for a lock, it simply guesses that this transaction is involved in a deadlock and aborts it. Since this is only guessing, the transaction manager may be making a mistake. The process of choosing the victim is an additional cost for this method.

Searching for a deadlock in the WFG should be fast but the searching frequency must be appropriate. If deadlock checking is performed too often, the cost of the searching could be very expensive. On the other hand, if deadlock checking is less

frequent, some deadlocks may cause the transactions to wait too long for each other.

2.4.2 Deadlock Prevention

Another solution to the deadlock problem is *deadlock prevention*[5] Deadlock prevention is a scheme whereby the transaction manager aborts a transaction when it concludes that a deadlock may occur. One deadlock prevention method is to check at the time that the transaction is going to wait for another to release some locks. The check should guarantee that the deadlock will not occur if the transaction is added to the WFG. If the transaction manager finds that a deadlock will occur after the transaction is added to the WFG, it simply aborts the transaction.[6] Therefore, deadlock will never occur.

2.4.3 Discussion

Deadlock detection is not a general method because it depends on the the context of the DBMS. To invoke the detection algorithm, one should know: how often does a deadlock occur and how many transactions will be affected by the deadlock. To select the transaction involved in the deadlock to be aborted, one should know: how long the transaction has been waiting and how long does it take to complete the transaction; how many data items have been locked in the transaction; how many locks are needed in order to complete; and how many transactions will be involved in the rollback. Since a lot of uncertainty exists, the circumstances of the DBMS should be clear in order to use deadlock detection.

Deadlock prevention, on the other hand, is not affected by the DBMS environment and the performance depends only on the search algorithm in the WFG. It is a more general method than the deadlock detection.

---

[5]The *deadlock prevention* mentioned here is cited from [Bernstein 89] and may be different from those given in popular texts. [Date 83] mentioned that deadlock prevention precludes the possibility of deadlock by, for example, requiring transactions to request all locks in a predefined sequence prior to commencing. If a system employs such deadlock prevention then transactions will never be aborted.

[6]Although any transaction in the WFG can be aborted to achieve deadlock prevention, we simply abort the one which is going to make deadlock occur for the reasons of simplicity and implementation.

2.5 Recovery Control

This section describes how failures in DBMSs can be handled. The issue for *centralized* DBMSs is explored since the current development of the multi-user version of REQUIEM is a centralized DBMS. Bibliographic notes and a detailed description for both centralized and distributed DBMSs issues can be found in [Bernstein 89] and [Gray 78].

Computer systems may fail in many ways. It is hard to build a DBMS that is immune to all possible failures. However, a good DBMS should be able to recover the database contents from the most common types of failures without external intervention.

In Section 1.4.2, three types of failures were identified as the most important failures met in centralized DBMSs, known as transaction, system, and media failures.

A transaction failure occurs when a transaction aborts. A DBMS should detect the abortion of the transaction and undo all the changes made by the transaction.

A system failure refers to the loss of the contents in volatile storage. Because of this, the database must be kept on a stable storage such as disk. To recover from the system failure, the DBMS should redo all the changes made by the committed transactions and undo all the changes made by the aborted transactions. This is achieved by using the information stored on the stable storage.

A media failure occurs due to the damage of the stable storage. The solution to this problem is similar to the others except that the redundant copy of data is stored in another piece of stable storage which makes sure that the stored data is safe enough for recovery.

The concepts underlying the recovery mechanisms for different types of failures are the same. The solution is that of *redundancy*. That is, the DBMS should always keep a redundant copy of the data, possibly in a different representation[7], in another part of the storage which is more reliable. For instance, store data in volatile storage in case of transaction failures, stable storage in the case of system failures, or another

[7]For example, the way of storing the same data in stable or volatile storage is different.

piece of stable storage for media failures.

The recovery mechanism works with the data manipulation in the DBMS. First we describe about how data is manipulated in the DBMS and then detail the recovery mechanisms.

2.5.1 DBMS Data Manipulation

The database is stored in stable storage to survive system failures in a DBMS. Part of the database is stored in volatile storage for quick access and transaction failure recovery. The following describes three steps needed for updating in a DBMS:

1. Copying the appropriate page (or block) of the file from disk to a buffer in main memory.

2. Updating the buffer in main memory.

3. Copying the buffer back from main memory to disk.

The manipulation of the data in centralized DBMSs are of two kinds: *update in place*, and *shadowing* [Bernstein 89].

With update in place, only one copy of each data item is stored in stable storage. Each time a data item is overwritten, the old value is destroyed.

With shadowing, there may be more than one copy of the data items kept in the stable storage. The database is considered as a number of fixed size disk pages (or disk blocks) for recovery purposes. When a transaction begins executing, the *current page* is copied into a *shadow page*. The shadow page is then saved on disk and available for other transactions while the current page is used by the transaction.

2.5.2 Recovery Mechanisms

For transaction and system failures, the recovery relies on the stable storage. This is achieved by the *logging* mechanism which is described below.

In the case of updating in place, each time a data item is updated, both the old value and new value of the data item are recorded into the stable storage, the *log*.

Conceptually, a log is a representation of the history of the execution [Bernstein 89]. The information stored in the log is used to redo the committed changes and undo the uncommitted changes by the transactions. Using the log, two ways of updating on the database, as shown below, can be done to ensure the transaction atomicity ([Korth and Silberschatz 86] and [Gray 78]).

- *Logging with deferred updates.* During the execution of a transaction all the write operations are deferred until the transaction commits. All updates are recorded on the system log which must be kept in stable storage. When the transaction commits, the information on the log associated with the transaction is used in executing the deferred writes. Therefore, if the system crashes before the transaction completes its execution, or, if the transaction aborts, then the information on the log is simply ignored. Using the log, the system can handle any failure which does not result in the loss of information on stable storage. In other cases, after a failure has occurred the recovery scheme consults the log and each committed transaction is redone.

- *Logging with immediate updates.* This means that all updates are applied directly to the database and an incremental log of all the changes to the system state is kept in stable storage. If a crash occurs, the information in the log is used in restoring the state of the system to a previous consistent state. This is accomplished by using the *undo* and *redo* operations. As before, the log can be used to handle any failure that does not result in the loss of information on stable storage.

As transactions are performed, the space consumed by the log may fill the available stable storage. The space which stores out of date information in the log should be freed and reused. In principle, when a system failure occurs, the entire log is searched in order to determine those transactions that need to be redone and those that need to be undone. Two disadvantages caused by this approach are listed below [Korth and Silberschatz 86].

- The searching process is time-consuming.

- Most of the transactions that need to be redone have already actually written their updates into the database and do not really need to be redone. Redoing them will cause recovery to take longer with no benefit.

The solution to these problems is *checkpoints* ([Bernstein 89], [Haerder and Reuter 83], [Gray *et al* 81], and [Gray 78]). It is common to keep one page of the log file in main memory until it is filled with log information and then write it back to disk only once rather than writing it to disk every time log information is added. To construct a checkpoint, a *checkpoint* record is written into the log periodically at the point when the system writes all the committed changed data into the database. The following sequence of actions are required to take place to perform this.

- Write all log records currently residing in main memory onto stable storage.
- Write all updated buffer pages to the disk.
- Write a log record *checkpoint* onto stable storage.

Therefore, all transactions committed before the *checkpoint* do not need to redo their write operations in case of a system crash.

For shadowing, the shadow pages are stored in the stable storage. During transaction execution, the shadow pages are never modified. When an update is performed, new copies of the modified database pages are created and they become the current pages. On the other hand, the old copies of the pages, which are called the shadow pages, are *not overwritten* and can be read by other users. When a transaction commits, the shadow pages are discarded and the current pages become the new pages stored in the database. If the transaction aborts, the current pages are simply discarded. This mechanism belongs to the class of *no-undo/no-redo* algorithms since it is not necessary to redo or undo a transaction to recover from a transaction or system failure [Bernstein and Goodman 83].

Until now, we have only considered recovery from transaction and system failures. To handle the media failure, *database backup* has to be done. Database backup means that the whole database (including the log) are periodically copied onto a cheap storage medium such as magnetic tapes. In case of media failure, the latest

backup copy can be reloaded from the tape to the disk and the system is restarted. However, the changes made on the database since the last backup was made may be lost and must be redone.

2.5.3 Discussion

For the update in place method, garbage collection should be considered. Redo and undo algorithms place additional computational costs on the recovery procedure.

For shadowing, the cost of storing and the manipulating different copies of data may be expensive. However, the reduction of redo and undo algorithms is an appealing advantage of this method.

It is necessary to strike a balance between the two methods. The following are the features desired for recovery control:

- Garbage collection should be considered.
- The use of storage should be minimized.
- Redo and undo processing should be minimized.

The design of the recovery manager in the RTM attempts to achieve the above goals.

Chapter 3

Design Principles

Chapter 2 investigated existing solutions to transaction management. This provides a firm foundation for developing the RTM (REQUIEM Transaction Manager). The issues arising from the comparison of the existing solutions are used to aid the design of the RTM.

Before the design principles of the RTM are presented, we first state some important guidelines for developing the RTM:

1. *The RTM is built in a centralized DBMS fashion*: Since the original REQUIEM was a single user DBMS developed for small and medium systems, it is reasonable to start the development of the multi-user version of the RTM as a centralized system rather than a distributed system.

2. *The design architecture and data structures of the original REQUIEM are not changed*: The RTM is built on top of REQUIEM without changing its original functionality.

3. *The RTM is an automated system and is invisible to its users*: The evolution of the RTM is based on *implicit* transaction management. The user would feel no difference when he or she is running transactions under the single-user or multi-user version of REQUIEM. Once the hidden RTM is built, it can be expanded to an *explicit* RTM easily. Using an explicit RTM, the user can issue some explicit commands such as COMMIT and ROLLBACK, to deal

with the transaction management manually. All the basic facilities dealing with transaction management built for the implicit RTM can also be used for the explicit RTM version. Therefore, it is obvious that the explicit RTM would comprise a minor extension of the implicit RTM.

Chapter 2 examined the existing solutions to those transaction management problems and described some features of the solutions. This chapter describes how those features are used to develop the design principles of the RTM.

Section 3.1 proposes a hybrid mechanism to overcome the drawbacks of the concurrency control mechanisms described in the previous chapter.

Section 3.2 describes how lock types are reduced to simplify the locking mechanism and the use of the multiple granularity locking in the RTM.

Section 3.3 proposed a new methodology, *query analysis methodology*, to support the multiple granularity locking which is more powerful than the lock escalation described in Section 2.3.2.

Section 3.4 examines the deadlock prevention method used in the RTM and the deadlock checking in the WFG (Wait For Graph).

Finally, the recovery mechanism of the RTM is presented in Section 3.5.

3.1 A Hybrid Concurrency Control Mechanism

Concurrency control is the main issue considered in transaction management. According to the features discussed in Section 2.2.6, concurrency control should deal with the following situations:

- Read transactions should not be restricted. Since read transactions do not conflict with other transactions, they should run without being interrupted or interrupting the other transactions.

- Write transactions should be restricted. Write transactions should be well controlled in a serializable fashion.

- In case that locking is used, the locking overhead should be reduced as much as possible. The reduction of the number of locking operations reduces the cost of locking and improves performance.

- A single locking operation should be fast. That is, the operation of setting a lock or releasing a lock should be fast. This would reduce the cost of locking operation and speed performance.

- The use of storage space should be minimised. Normally, storage space, including volatile storage and stable storage, is limited. The transaction manager should avoid over using the storage space.

A hybrid concurrency control mechanism is proposed to deal with all the issues listed above.

3.1.1 Combining Different Mechanisms

Conflicts between two transactions are of two kinds, *read* and *write* conflicts[1]. The read conflict occurrs when there is improper interleaving between read operations of one transaction and write operations of the other transaction on the same data item. For example, a *read* transaction and a *write* transaction both are accessing the data items of a particular table at the same time. If the *write* transaction commits before the *read* transaction is finished then as the changed data are made permanent, there is a possibility that the read transaction may read post-change data where before it was reading pre-change data. This will cause read conflict and lead to loss of data integrity. Similarily, write conflict is caused by interleaving of write operations between different write transactions on the same data items.

Since reading the data from the database should never cause a loss of integrity, the optimistic concurrency control schema used must ensure that read transactions proceed in an optimistic manner without sacrifising the integrity of data. According to this schema, cases of read conflicts can be avoided. Unlike read transac-

[1] A *read* conflict is either a *read/write* or *write/read* conflict. A *write* conflict means a *write/write* conflict.

tions, write transactions are controlled by the locking mechanism (pessimistically). That is, instead of having a *validation phase*[2] to detect the conflicts and a *write phase* to perform the actual writing operations, the *pessimistic* (two phase locking) method is used to solve the conflicts between the transactions. Therefore, those worse cases which involve write transactions are taken care of by the *pessimistic* (locking) method.

Timestamp ordering is not used since it "*is almost certainly not appropriate in a nondistributed system. Indeed, there is considerable skepticism as to its practicality in distributed systems also*" [Date 90].

The ideas of *multiversion* concurrency control, also known as *shadowing*, are employed to assist *read consistency* which assures that read transactions do not need locking to proceed. When a write transaction is started, two versions, the new version and the old version, of data items which could be changed are created in the volatile storage. All the changes are made on the new version, the old version of the data items, on the other hand, is available to other transactions. A read transaction can not read the data which is being changed by the write transaction before it commits; the old version of consistent data is visible to the read transaction. This makes certain that the unstable data is always unavailable to the users. When the (write) transaction commits, the new version of the data items are written to disk (made permanent) while the old version of data is kept in the volatile storage until those read transactions, which have read the pre-change data and have not completed the transaction, commit. Transactions start after the commit of the write transaction will read the new version of data from the disk and keep at least a copy (write transactions will still keep two copies of data items read) in the volatile storage[3].

write conflicts are controlled by locking to ensure that all the transactions run are in a serializable fashion. If *write* conflicts occur rarely in a database management system [Kung and Robinson 81], little overhead would be introduced in such cases.

[2]The three phases of processing in an optimistic mechanism were introduced in Section 2.2.2.

[3]Section 6.3.1 shows how this is implemented in the RTM and a comprehensive example is also given in the section.

Locking is used to guarantee the serializability of the transactions run under REQUIEM since it is easy to understand and implement. It has also been widely accepted and used in most DBMSs, especially relational DBMSs [Valduriez and Gardarin 89].

Since *read* conflicts have been taken care of by the optimistic concurrency control, only *write* conflicts between transactions remain to be considered. This greatly reduces the overhead in lock setting.

The RTM simplifies the types of locks which reduces the frequency of locking operations and speeds the performance of the RTM. The setting of locks is described in Section 3.2.

The storage manipulation of the RTM is described in Section 3.5.

3.1.2 Discussion

The concurrency control mechanism proposed in this thesis is based on the discussion of the features which have arisen from the comparison of the existing mechanisms. Read consistency, which allows read transactions to proceed without locking, enables concurrency in the RTM transaction processing. Write conflicts are dealt with by the locking mechanism. Moreover, types of locks are reduced to simplify the operations of locking and to speed up the performance of the RTM. We will describe how this hybrid mechanism is implemented in the next chapter.

3.2 Multiple Granularity Locking (MGL)

As concluded in Section 2.3.4, multiple granularity locking is the best solution to the data locking size problem. Lock escalation is the proposed solution to support multiple granularity locking, because the behaviour of the transaction can not be predicted in advance. The identification of which data should be locked is only available at compile time. However, an increased possibility of deadlock is the main drawback of this method. A *query analysis methodology* is proposed to predict the behaviour of the query transaction as much as possible. The details of this

methodology are fully described in the next section.

Since multiple granularity locking is dealt with, the intention locks introduced in Section 2.3.1 are used. That is, when a *read* (r) or *write* (w) lock is acquired on the small granule, say a tuple, the *intention read* (ir) or *intention write* (iw) lock is first acquired on the page and table, which contain the tuple, to ensure that no other transactions can explicitly lock the whole page or table containing the tuple. However, the r or ir locks are deleted from our lock tables in order to simplify the locking subsystem in the RTM. The reasons for omitting r and ir locks are:

1. Since read transactions do not need locking, r and ir are not necessary.
2. To reduce probability of deadlock events, r and ir locks should be omitted.

Consider the example in Figure 3.1. When a *write* transaction is going to

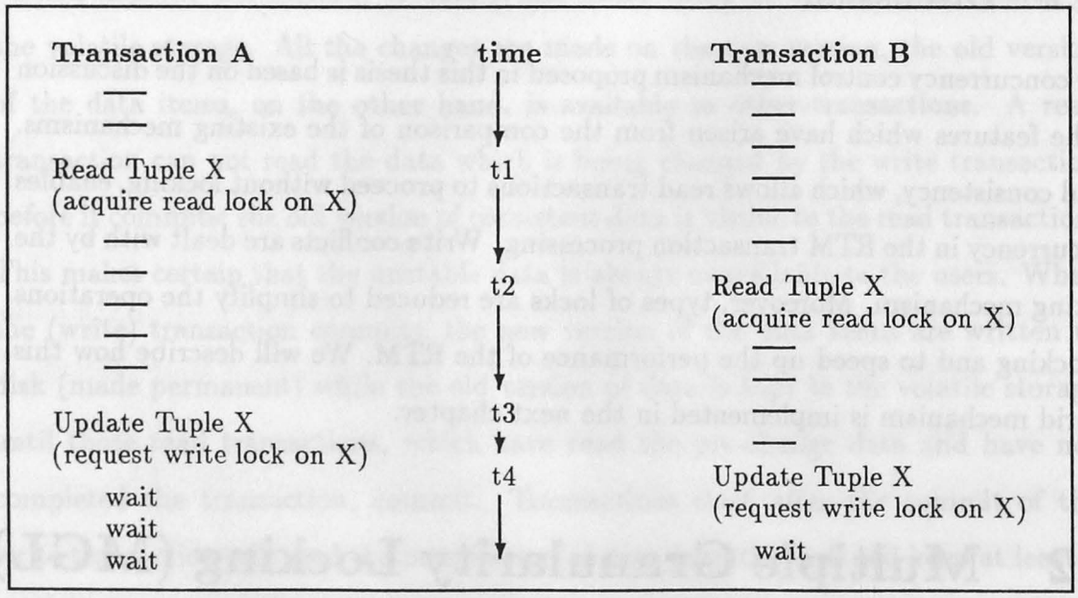


Figure 3.1: An Example of Deadlock Due to Read Lock Acquired.

update a tuple, it has to acquire the read lock and upgrade it to write lock on this specific tuple. As shown in Figure 3.1, both transaction A and B are waiting for each other to release the *read* lock on X in the final stage. This is one of the deadlock examples since none of the transactions can proceed until one or the other transaction aborts. We can rectify this problem by removing

"r" and "ir" locks, therefore avoiding deadlock occurrences in such cases.

3. To simplify the lock operations, *read* transactions do not use locks. When a *write* transaction reads a tuple, it usually intends to update the data. Instead of setting a *read* lock on the data, a *write* lock should be requested directly. This ensures that the data read by the transaction are always ready to be updated and should not be read by the other *write* transactions.

The reduction of lock types also reduces the number of locking operations since less locks are involved in transaction processing. This makes the RTM a simpler and more elegant transaction manager.

3.2.1 Discussion

MGL is not sufficient for serializability, though it prevents transactions from owning conflicting locks. To ensure serializability, a *transaction manager* that manages data items of different granularities must combine the MGL with the 2PL protocol. 2PL gives rules regarding when to lock or unlock data items. The MGL indicates how to set and release a lock on a data item which could be either an entire file or a single tuple.

The details of locking operations in REQUIEM will be discussed in the next chapter.

3.3 A Query Analysis Methodology to Support DGL

In our design, we propose a module called *the query analysis module* (*QAM*). The aim of this module is to determine the granularity of locks which will be requested by the RTM during transaction processing. This decision is based on a prediction of how many records are likely to be accessed in the transaction. The interpreter may be able to make such predictions by analyzing a transaction's context and thereby generating specific granularity lock requests that will be explicitly issued by the

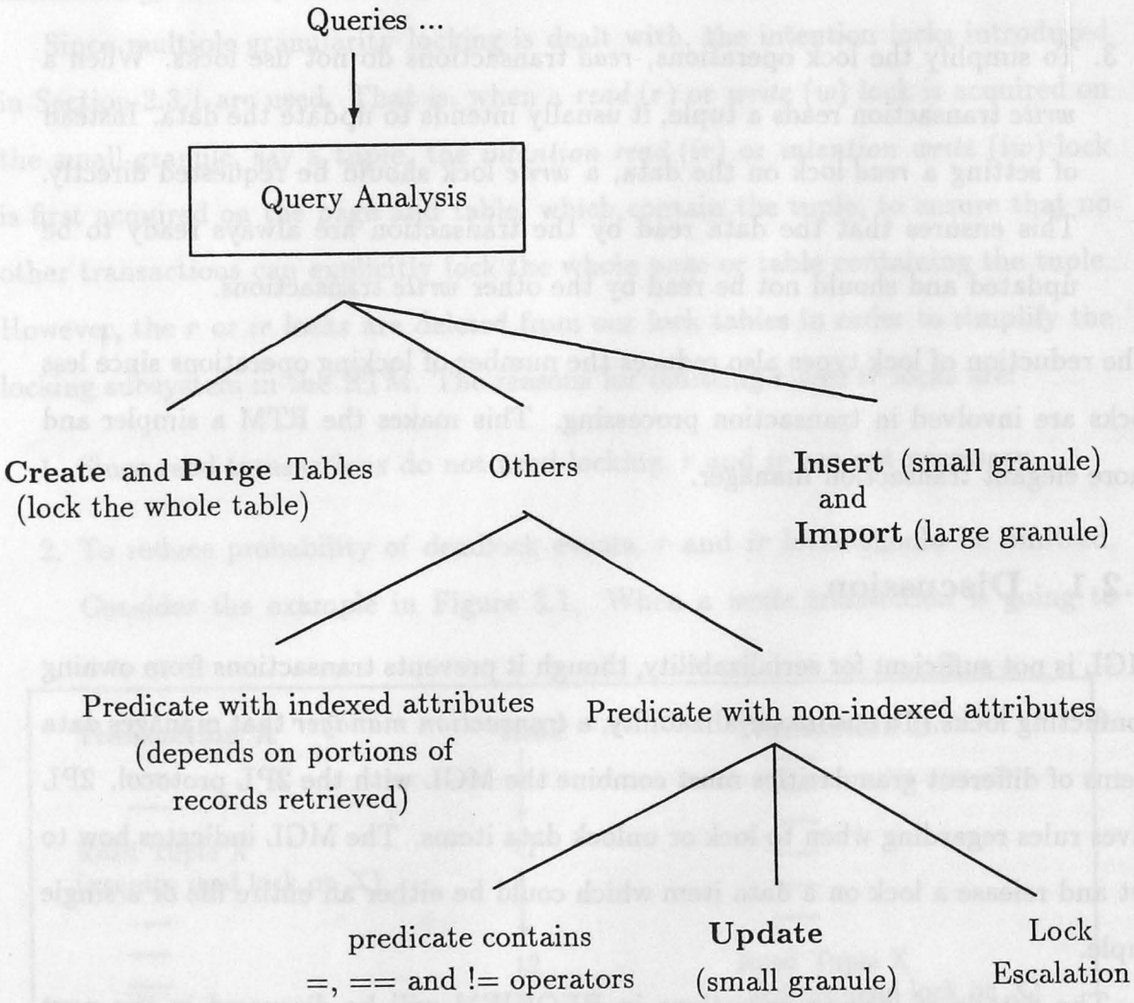


Figure 3.2: The Query Analysis Module.

transaction at run-time. If large transaction queries are sent to the RTM, the RTM should be able to tell that the query will generate many record accesses to certain files. Thus, large granularity is set to reduce the number of locks required. On the other hand, if small number of records are to be accessed, small granularity is set to increase the concurrency of transactions. This module is invoked in the *REQUIEM interface process*. Details will be explained in Section 4.3.

The various stages of *query analysis* are shown as Figure 3.2. According to this figure, query transactions may be divided into seven categories as shown below:

1. Queries including **create** or **purge** tables.

2. Queries including **insert** or **import** clauses.

3. Queries without predicates.

4. Queries with predicates containing indexed attributes.

5. Queries with predicates containing equality or non-equality.

6. Queries including **update** clauses.

7. Others.

These categories are prioritised according to their sequence. That is, if a query contains **create** or **purge** clauses, it is not necessary to check any further to determine category. Granularity decisions are made according to the category of a query. The following subsections show the processing of a query during the phase of analysis. Examples based on the *supplier-and-product* database, as shown in Table 1.1, are given for each category.

3.3.1 Create and Drop clauses

The *query analysis module* first checks if the query is a **create** or **purge** clause. **create** and **purge** clauses are used to create and delete a table, respectively. Such actions should be restricted since a table can not be created or deleted by more than one user at the same time. Therefore, the whole table is locked in such cases.

3.3.2 Insert and Import clauses

The *query analysis module* then checks if the query is an **insert** or **import** clause. The following is an example of **insert** statement.

Example 3.3.1 *Insert tuples into the supplier table.*

insert supplier

The **insert** statement is usually used for inserting small amount of tuples interactively and will not affect the existing tuples in the database, therefore it is natural to request fine granularity locks for the **insert** clauses.

Besides the **insert** statement, RQL provides an additional file manipulation statement **import** to insert data into a table. The following is an example of **import** [Papazoglou and Valder 89].

Example 3.3.2 *Insert tuples into the* supplier *table using the data contained in the file named supplier.dat.*

import supplier **into** *supplier*;

The first supplier in the statement is the name of the input file internally qualified by the suffix *.dat*. This input file must contain values of the tuple attributes, with each attribute on a separate line. Then the tuples are appended to the relation *supplier*.

import is suitable for inserting a large amount of tuples into the database from the input file. It is necessary to request coarse granularity locks for the **import** transaction since a large transaction is likely to be involved.

3.3.3 Queries without Predicates

The current version of REQUIEM does not allow for using updates/deletes without predicate. Only read transactions in REQUIEM can be expressed without any predicate. The following is an example.

Example 3.3.3 *Find all the tuples in the table supplier.*

select from supplier;

For write transactions such as **update** or **delete**, the tuples to be updated or deleted need to be specified by using a predicate (the statement with **where** clause). Therefore, queries without predicates are not considered here since read transactions are the only case and no locks are needed for read transactions in the RTM.

3.3.4 Predicates with Indexed Attributes

The queries with predicates are divided into two categories, predicates which contain indexed attributes and those without. In the former case, decision of setting types of granularity lock is made by associating indexed files to find out the portions of records to be accessed. A representative example is given below.

Example 3.3.4 *Update the location of the supplier whose name is Jones.*

update location **from** supplier **where** supname = "Jones";

If the number of records which satisfy the predicate in the table *supplier* are above a previous fixed number[4], say 30, or exceed a fixed percentage, say 30 percent, of the table, the module will set the coarse granularity locks, otherwise, fine granularity locks will be applied. That is, if it is found more than 30 tuples or more than 30 percent of tuples in the table supplier with supplier name "Jones" according to the indexed file, the granule will be set to page level automatically. Otherwise, the locks are set on the tuple level.

3.3.5 Predicates containing Equality or Non-equality Operators

With queries with predicates containing non-indexed attributes, the prediction of the portion of tuples to be updated relies on the analysis of the predicate statements. In RQL, a predicate select formula is an expression involving a linear sequence of operands, operators, and delimiters [Papazoglou and Valder 89]. These rules are similar to the rules utilized in high-level languages such as Pascal or C for creating analogous expressions.

The queries with predicates containing =, == or != operators will be analysed at this stage. In the predicate, the comparison operator = means exactly equal and != means not equal. In addition to = and !=, RQL uses the == partial comparison operator to determine whether it should select a tuple, or not

[4] 30 is the number used in our experimental test. The fixed number may be changed according to user's demand.

[Papazoglou and Valder 89]. This is the practical requirement because the user for example not remember the full name of a supplier, or a product. Consider the following query:

Example 3.3.5 *Update the names of all the supplier currently residing in cities of the names start with Lo.*

update supname **from** supplier **where** location == "Lo";

Usually, few records satisfy a predicate involving = or == operators, but on the other hand, many records satisfy a predicate with != operator. It is easy to tell how to set granularity locks for this kind of query. Again, one example is given below.

Example 3.3.6 *Delete all the records of suppliers currently residing in London.*

delete supplier **where** location = "London";

It is clear that not many tuples would satisfy this predicate under the assumption that a database usually contains diverse data values in any attribute. Hence, fine (tuple) granularity is set in this case. However exceptions are possible. When the number of records satisfy the predicate are over a fixed number, the locks are upgraded to the coarser granularity locks. This is known as *lock escalation* which is described in the next section.

3.3.6 Lock Escalation

Finally, *lock escalation* is considered for the remainder of this section. Lock escalation means that the transaction manager starts requesting small granularity locks one by one as it receives operations from the transaction. If a transaction obtains more than a previously fixed number of locks of fine granularity, then the *lock manager* starts locking the high level of granularity. That is, it escalates the granularity of the locks it requests. However, lock escalation can cause a deadlock to occur. For example, suppose two transactions are holding *iw* locks on a file and are setting *w* locks on records, one by one. If they both escalate their record locking activity to

a file lock, they will both try to convert their *iw* lock to a *w* lock. The result is a deadlock. This can be detected by deadlock checking. When deadlock occurs, the transaction manager simply aborts the transaction which causes deadlock.

3.3.7 Discussion

In QAM, those cases which can be predicted before the read or write operations are actually performed are extracted and the remainder are determined by *lock escalation*. Since the remaining cases are rare, deadlock occurs infrequently. This would be better than a transaction manager which utilizes only lock escalation to solve the multiple granularity problem.

3.4 Deadlock Prevention

Section 2.4.3 concluded that deadlock prevention is better than deadlock detection as it is simpler and transactions do not have to wait when deadlock occurs. In deadlock prevention deadlock never occurs. The following will show that a WFG constructed for deadlock prevention is simpler than one for deadlock detection, so deadlock checking is more efficient.

Many transactions may wait for one transaction to release a lock. Hence, the WFG is presented as a group of tree structured graphs. One example of a WFG for deadlock prevention is shown in Figure 3.3.(a). This graph shows that some transactions are waiting for either transactions T_1, T_2, or T_3 to release the lock. Since no deadlock occurs in the WFG, no cycles are involved. This is a typical tree structured graph with directed edges. Suppose a transaction T_1 requires a lock concurrently held by T_{11}. This means an edge $T_1 \rightarrow T_{11}$[5] should be added to the graph. If the edge is added, the deadlock cycle $T_1 \rightarrow T_{11} \rightarrow T_7 \rightarrow T_4 \rightarrow T_1$ as shown in Figure 3.3.(b) would be created.

The RTM checks if deadlocks could occur before any edge is added to the graph. To check whether the edge $T_1 \rightarrow T_{11}$ will cause a deadlock, RTM has to find out if

[5] The presentation of the directed edge has been discussed in Section 2.4.1.

there exists a directed path from T_{11} to T_1. If it does, then deadlock would occur.

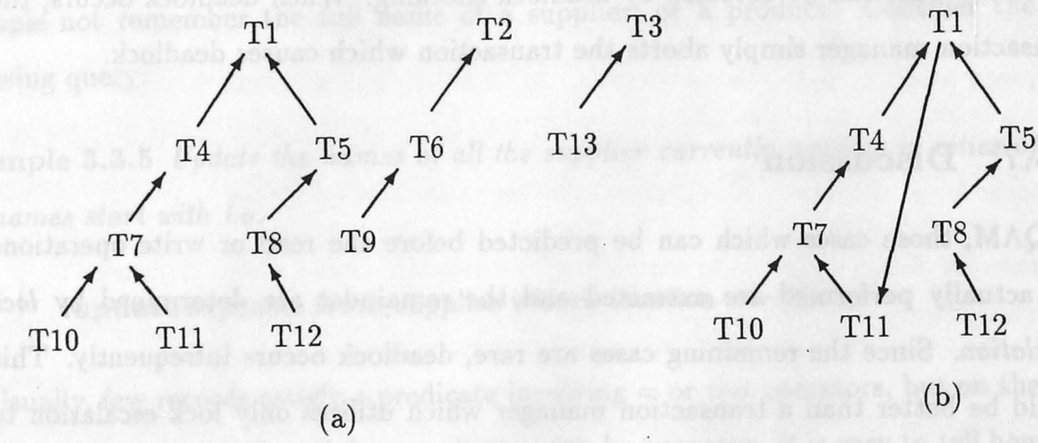


Figure 3.3: The Wait for Graph in Deadlock Prevention.

The checking can be started by searching for T_{11} in the graph. As soon as T_{11} is found, the target transaction T_1 can be easily found in the path by following the arrow. That is, a directed path exists between T_{11} and T_1, therefore deadlock will occur if the edge $T_1 \rightarrow T_{11}$ is added to the graph. The deadlock is prevented by aborting transaction T_1. Transaction T_4 and T_5 will then *wake* in order and transaction processing continues.

On the other hand, the deadlock checking for deadlock detection is more complicated. Consider the WFG example shown in Figure 3.4. Cycles may be involved in

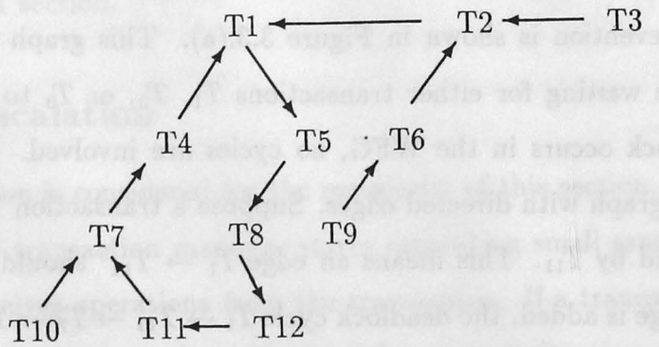


Figure 3.4: The Wait for Graph in Deadlock Detection.

the WFG in deadlock detection, that is deadlocks may occur before deadlock checking. All cycles have to be found and the candidates which are going to be aborted

for breaking the cycles have to be chosen. For these reasons, deadlock detection is not used.

3.4.1 Discussion

The deadlock problem is usually solved in terms of graph searching. Deadlock prevention detects deadlock before it occurs. This method does not only simplify the WFG for searching, but can also ease the actions involved in solving the deadlock problem.

The WFG in deadlock detection is more complicated for searching. More actions, such as when to detect the deadlock cycles in the WFG and choosing the candidates to be aborted to break the deadlock, are involved. This makes the situation more complicated and difficult.

3.5 Recovery

The recovery control in the RTM tries to strike a balance between *update in place* and *shadowing*[6]. In the RTM, update in place is used on stable storage, however, the shadowing concept is used for volatile storage. That is, exactly one copy of the database is stored on the stable storage. A shadowing (or multiversion) method is used in the sense that the current working data are stored on volatile storage.

In REQUIEM, tables are stored in files on disk (stable storage). We first describe how data is manipulated in the RTM.

- Write transaction: Figure 3.5 shows the data manipulation involved in a write transaction. When a write transaction starts, a copy of the data items (the old version) which are possibly changed by the transaction are stored on the shared volatile storage, namely the *page*[7] *cache*. The changed data (the new version) are stored in another cache[8]. The old version of the data items are kept in

[6]These two methods are described in Section 2.5

[7]The page refers to a unit of data stored in the stable storage or cache.

[8]The cache could be a tuple or page cache, depending on the type of granularity applied.

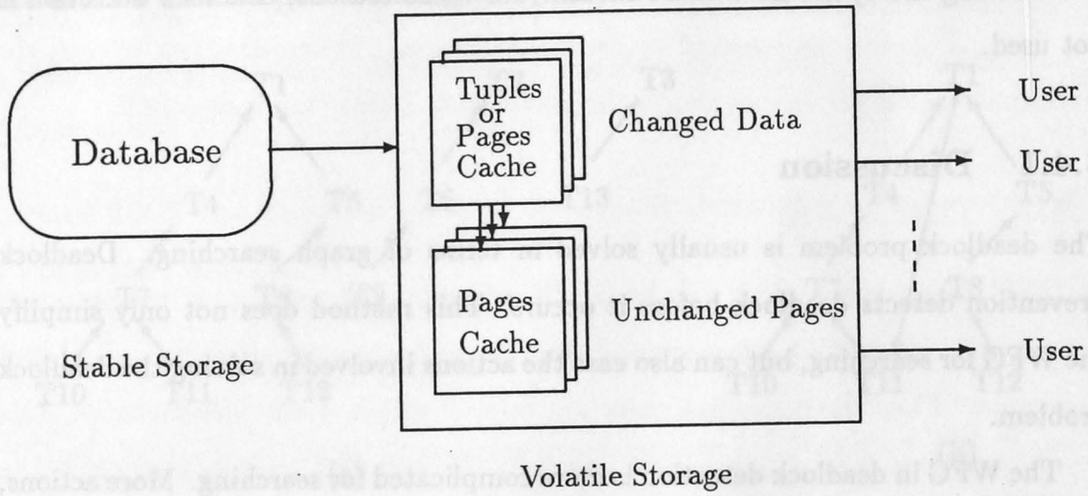


Figure 3.5: An Example of The Write Transaction Data Manipulation.

the cache until the transaction commits. Hence, before a transaction commits, the old version of data is available to users by means of the cache. Once the transaction commits the changed data in the cache are made permanent.

- Read transaction: Figure 3.6 shows the data manipulation involved in a read transaction. When a read transaction starts, it first checks if the requested

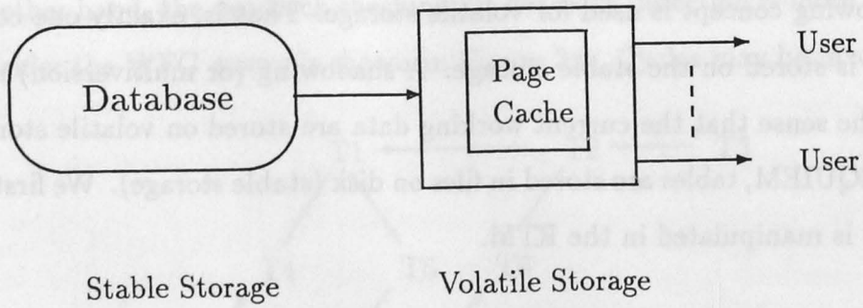


Figure 3.6: An Example of The Read Transaction Data Manipulation.

data items are available from the cache. In case the data items are changed by some uncommitted transactions, the committed data (the old version) in the cache are fetched. If no other process has read or changed the requested data, then it simply fetches the data from stable storage into the cache and reads it. However, only one *page* cache is kept for each table if no write transactions

consult the same table. Since no changes will be made for the read transaction, only one page cache is enough for transferring the data and this minizes the use of volatile storage.

No redo or undo is necessary to be made on the stable storage to recover from a transaction failure, since no changes are made to the stable storage before a transaction commits. The only task for the transaction recovery is to delete all the data which are read or written by the transaction in the cache.

Whenever a transaction commits, before the committed data are stored into the database files, the old version and new version of the data are written to the *log* on the stable storage which is separated from the database files. After the committed data are made permanent in the database files, a commit signal of the transaction is stored in the log, which indicates that the transaction is committed and no redo or undo is required. Then both versions of data stored in the cache will be discarded. If the system crashes during the transaction commit, the RTM will restart the DBMS by undoing the uncommitted changes according to information stored in the log.

Media failure problems are solved by combining the system recovery mechanism of the RTM and the UNIX backup scheme. The content of the database is saved on tape periodically by using the UNIX backup scheme. When a media failure occurs, the database files on tape are first restored to disk. When this is done, the restart of the RTM will recover the database according to the information in the log. This is done as described for the system failure recovery.

3.5.1 Discussion

In the RTM, the concepts of recovery are simple and easily implemented. Further, fewer operations are needed for recovery control than the mechanisms mentioned in Section 2.5. Redo and undo are not necessary for the transaction failure handling. To recover from the system and media failures, the information in the log is used to undo the transaction whose commit processing is not completed.

Only one uncommitted transaction could occur in our recovery model since all transactions are centrally controlled and data updated by a transaction is written

to the stable storage only when the transaction commits. Undo may be necessary for a system restart, but is rare. The reasons are:

1. Since commit is an atomic action which writes all the changed data in the database, system crash during transaction commit time would hardly happen. Because of this, no redo or undo is necessary for system recovery.

2. In the worst case, in which the system crashes during the commit processing, only one transaction needs to be undone. The uncommited data of only one transaction could be stored in the database. Therefore, the task of system recovery becomes simple. As long as the old version of such data are found in the log, the database can be recovered by undoing the transaction according to information contained in the log.

Chapter 4

System Overview

In the previous chapter, the design of the mechanisms for *concurrency control*, *granularity problem solving* (*QAM*), *deadlock prevention*, and *recovery control* were outlined. In this chapter we indicate how they fit into our RTM architecture. A detailed design of the system architecture will be presented in Section 4.1. The communication between modules will be addressed in Section 4.2, and the detailed processing of each module in the architecture will be described from Section 4.3 to Section 4.6.

4.1 Architectural Overview

Our design avoids changing the original design context of REQUIEM, since REQUIEM is a complete single-user DBMS. It would be senseless to redesign one system to build another since the effort required to redesign a system is unacceptable. Therefore, we design a transaction manager module external to REQUIEM, and a communication interface between REQUIEM and the transaction manager. The design architecture of the multi-user version of the REQUIEM is shown in Figure 4.1. Each double box represents a process which is an execution of a computer program. Every single box is the module within a specific process.

Many users may access the database simultaneously. Transactions must be controlled properly such that one user's action does not interfere with another's. The architecture is *centralized* because operation requests from different transactions are

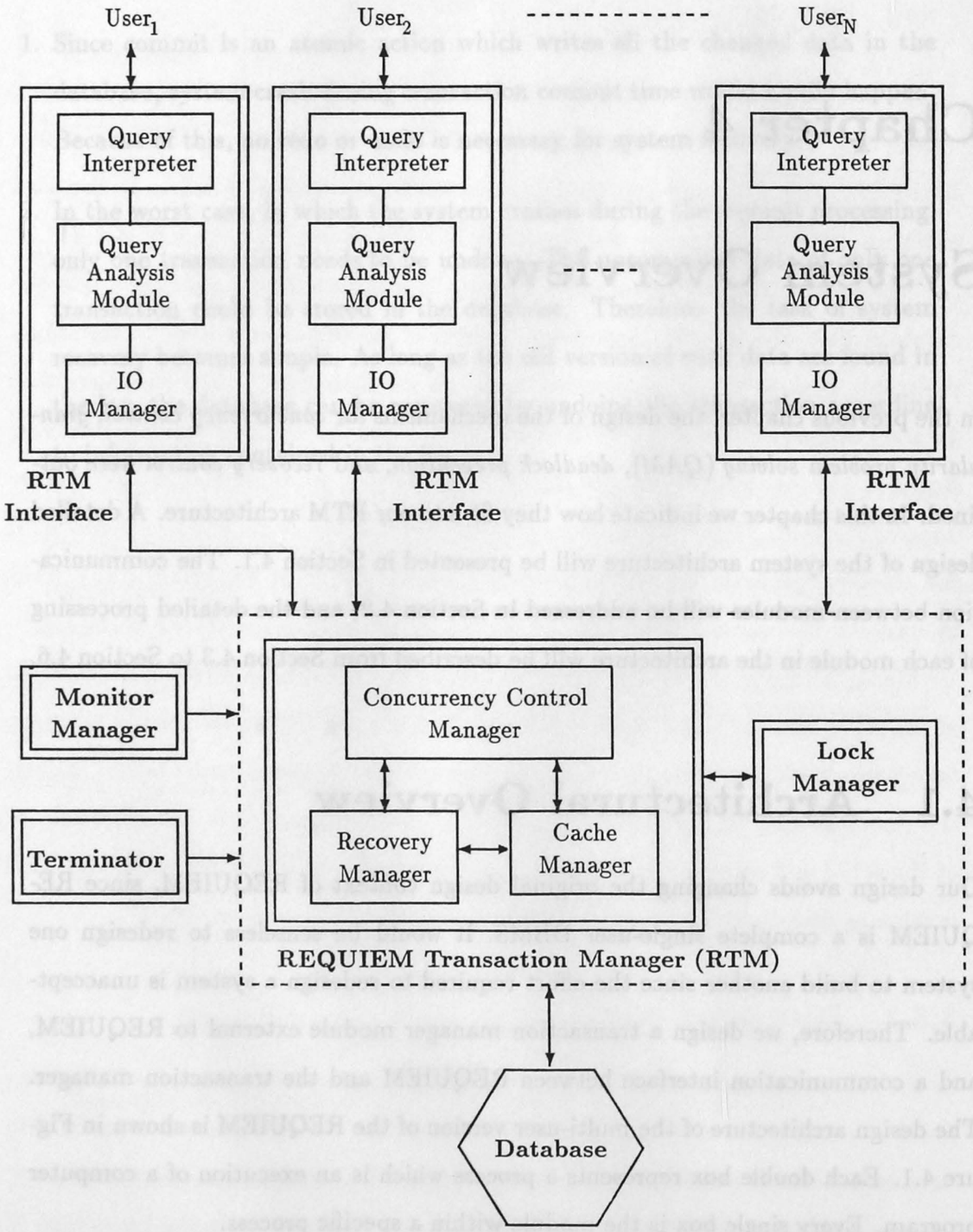


Figure 4.1: The Architecture of the multi-user REQUIEM.

passed to and controlled by the *central transaction manager*.

Some advantages of this centralized design are:

1. The *transaction* and *lock* managers, being implemented in separate processes, are protected from damage that might result from a faulty or devious *RTM interface* process.
2. The centralized process can run with a separate *user-id* under UNIX so the data can be easily protected by the security system provided by UNIX.
3. As the lock manager is the only process that can directly access the data, it can easily control concurrent access by keeping a table of locks in its own address space.

In multi-user REQUIEM, when a transaction is executed, it is first passed to the *RTM interface*, which does the compilation, and transferred to low level IO (input and output) operations, read or write[1] operation, which are sent to the RTM. The RTM then schedules all the IO operations according to the decision made by the *lock manager*. However, *read* transactions are never delayed. When the RTM receives the IO operations sent from *read* transactions, it does not pass them to the *lock manager*. It simply finds the right version of data from the cache or stable storage and returns the results to the RTM interface directly. *Write* transaction operations may be delayed if the *access primitives*, which are locks, can not be acquired. Alternatively, the RTM may send the *abort* message back to the RTM interface to abort the transaction if the transaction causes *deadlock*. If the transaction proceeds without producing any error, both the updated and the old version of data will be kept in the *volatile storage*, namely *cache* storage, by the *cache manager*. The updated data of each transaction is stored in the stable storage only when the transaction commits. Therefore, if a transaction aborts abnormally before it commits, the updated data will not be written to stable storage and the updated version kept in *cache* will be deleted. Otherwise, the changed data will be stored in the database permanently.

A *monitor manager* and a *terminator* are introduced. The *monitor manager* sends a message periodically to the *RTM* to ask it to check if any of the trans-

[1]The read and write represent the single read and write operation to read data from the stable storage to the volatile storage or write data from the volatile storage to the stable storage in the single-user version of REQUIEM. In the multi-user version of REQUIEM, these read and write operations are sent to the centralized transaction manager.

actions running have aborted. In this case, the data changed or retrieved by the transaction will be deleted from the cache and the locks related to the transaction will be released. Those transactions which are waiting for the locks will then be awakened. The *terminator* is used by the DBMS administrator to terminate those central processes such as the RTM and *lock manager*. The *monitor* process will cease automatically as soon as the RTM terminates.

The following sections describe the communication between modules and details the work of each of them.

4.2 Functional Components for Interprocess Communication

Communication between modules and processes is important because of its effect on the efficiency of the system. Communication between modules is easy, because modules are in the same *process* and share the same part of the memory space. However, it is not that easy to communicate between processes, since they do not share the same memory when they are processing. This section describes the *interprocess communication* used in the RTM.

Shared memory provides the fatest way for interprocess communication[2] [Rochkind 85]. The same memory is mapped into the address space of different processes. As soon as data is written to the shared memory, it is immediately available to other processes.

Semaphores are used to synchronize different processes. A semaphore is a flag that prevents two or more processes from accessing the same resource (shared memory) at the same time. If a process does not check a semaphore before accessing shared memory, the results are unreliable.

send and **receive** are two primitives for interprocess communication. Processes send messages to other processes to communicate with each other. When processes

[2] Nine techniques for interprocess communication are detailed in [Rochkind 85].

communicate, messages are *sent* (or written) to a *queue* in the shared memory. As soon as the message is available in the queue, it can be *received* (or read) by other processes.

Messages sent to a queue are deposited there in order of arrival. A message may consist of any arbitrary data. If the queue becomes full, *send* blocks until it empties enough to hold the message being sent.

Figure 4.2 shows the interprocess communication between two single processes.

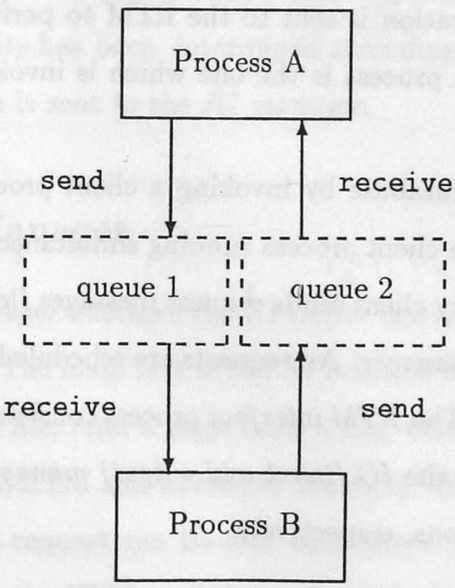


Figure 4.2: The Interprocess Communication between Two Processes.

Messages are sent to *queue 1* by *Process A*, from where they are received by *process B*. Similarly messages are sent from *process B* to *process A*. In order for two processes to communicate with each other they must know each other's receiving queue address. Each can then send messages to the other's queue.

The RTM and *lock manager* are both a single central DBMS process; a RTM interface process performs an access by sending a message to the central *transaction manager* and *lock manager* process. A message containing the response (a status code or some data record) is sent back.

To achieve good performance with the centralized approach, *interprocess commu-*

nication must be fast. We will explain how to implement more efficient *interprocess communication* in Chapter 6.

4.3 The RTM Interface

The *RTM interface* process is the same as the original single-user REQUIEM except that the IO manager is upgraded and a *Query Analysis Module* is added. Now, when RTM interface issues a read or write operation[3], instead of immediately accessing the stable storage, the operation is sent to the RTM to perform the read or write action. The RTM interface process is the one which is invoked by the user and is called the *client process.*

A user can access the database by invoking a client process. This means that there may be more than one client process running simultaneously, although there is only one central RTM. Every client sends request messages (low level IO operations) to the central *transaction manager.* All requests are scheduled centrally to guarantee transaction serializability. The *RTM interface* process consists of a *query interpreter, query analysis module,* and the *IO (input and output) manager.* They are described in the following three sections, respectively.

4.3.1 The REQUIEM Query Interpreter

When the *RTM interface* receives a transaction (i.e. a query) from a user, the transaction is first processed by the *REQUIEM query interpreter* which validates and translates the transaction into low level access operations[4]. These operations are in a suitable form for accessing the database in the stable storage.

The REQUIEM query interpreter first decomposes the query and makes sure that the requested tables exist and that the query follows the RQL syntax. If there is any error at this level, the transaction is aborted by the REQUIEM query interpreter and an appropriate error message is returned to the user. Otherwise, the processing

[3]Since the RTM is implemented under UNIX operating systems, the *read* and *write* operations refer to the `read` and `write` system call in C language as described in [Kernighan and Ritchie 88].

[4]The REQUIEM query interpreter is detailed in [Papazoglou and Valder 89].

continues and the requested operations, such as *find the retrieved tuples* and *update the tuples found*, will be passed to the *query analysis module*.

4.3.2 The Query Analysis Module

The *query analysis module* tries to predict the behaviour of write queries by analyzing them at compile time. Read operation requests from a *read* transaction are passed to the next module, *IO manager*, in the sequence of arrival. This is because read transactions do not need any locking control (optimistically) to be processed. After the lock *granularity* has been determined according to the rules described in Section 3.3, the decision is sent to the *IO manager*.

4.3.3 The IO Manager

The *REQUIEM IO manager* manages the IO (input and output) for client processes (REQUIEM Interface). The main task of the IO manager is to send the IO operation requests such as open a file, read a page from a file, write a record into a file and close a file, to the central RTM and to receive the reply[5] from it when the operation is completed. Only one request can be sent each time. The IO manager can not send a second request to the RTM until the reply of the first request from the RTM has been received. If the operation is delayed by the RTM because it is waiting for a lock to be granted, the IO manager must also wait until the lock is granted and the RTM replies.

The *REQUIEM IO manager* sends the IO requests for data to satisfy the query predicates, together with the locking requests, to the *transaction manager* via the *interprocess communication* interface.

The IO manager also manages a local cache for each client process. The original REQUIEM IO operation is based on a tuple basis. Each time, a single tuple is either read or written to the stable storage. This is not adequate for retrieving a large amount of data. If, say, 10,000 tuples are read, then 10,000 IO operations

[5]More details of different replys will be discussed in the next section.

must be performed. This is improved by upgrading the IO from the tuple basis to the page basis. Each time, an entire block of data (page) is read from the file rather than a tuple. Then the page is kept in the local cache. The client may retrieve the tuples of the page in the cache. This reduces the number of IO operations and improves the performance.

4.4 The REQUIEM Transaction Manager (RTM)

The RTM is the central process which controls all requests from the client processes and makes sure that all transactions are run without resulting in data inconsistencies in the database.

4.4.1 Concurrency Control Manager

The IO operation requests received from client processes and by the *transaction manager* are passed to the *concurrency control manager*. This module schedules the order of the IO requests based on the *two phase locking* protocol. That is, before a commit is issued, locks can be acquired but can not be released. Locks are released only after the transaction is committed and changed data thus made permanent in the database. This is achieved with the aid of the *lock manager*. When the *concurrency control manager* wants to perform *write* transactions, it sets the locks required by sending requests to the *lock manager* and asking it to do so. If the locks are acquired, it then passes the requests to the *cache manager* to perform the read or write operations.

4.4.2 Cache Manager

The *cache manager* is used to manage the space in the volatile storage. For read transactions, the cache manager always maintains a page space in the cache for each table read. The page of a table read from the stable storage is kept in the cache

until another page read or the table (or file) is closed.

With write transactions, more space is needed to keep both an old and an updated version of the data in the volatile storage. When the data is read by a write transaction, the data is first stored in the page cache and marked *uncommitted.* The data marked uncommitted can not be removed until the mark is disabled. All changed data are written to the cache[6]. When the transaction is committed, the updated data is then written to the disk and the old version of the data is discarded.

The *cache manager* finds the right version of data to pass to the client process. It also stores the updated data in cache storage until a transaction commits.

4.4.3 The Recovery Manager

When the transaction commits, the *recovery manager* writes all the changes from the updated cache to the stable storage and the old version of data stored in the cache will be deleted. If the transaction is aborted before committing, no changes are made to the stable storage and changes made in the cache will be cleared. Since the stable storage is not changed, *undo* on the stable storage is not necessary. Thus transaction failure is easily dealt with.

For system failure recovery, all updated data is written to the *log* file which is separated from the database. Each log entry comprises the old values, the new values of the changed data, and the identification of the transaction. After all the changed data are stored into the stable storage, a *commit signal entry* is recorded into the log file. When a system failure occurs, all the central processes are rerun to restart the RTM. Whenever the RTM is started, it first checks the log file. If any uncommitted transaction has been recorded according to the commit signal entry, all changes made by the transaction will be undone. If all the data recorded in the log file are committed, then the database is consistent and there is no need to make any correction.

To recover from media failure, the procedures are similar to the system failure recovery. The database and the log file are periodically archived into another safe

[6]The cache could be tuple or page cache depending on the granularity used.

storage such as tape. When the RTM is restarted, it first restore the database and the log file from the tape. The RTM then can then be restarted and according to the log, the failure can be easily rectified.

4.5 The Lock Manager

The *lock manager* maintains a set of *lock tables* that record the locks acquired by transactions. To process a lock operation, the *lock manager* tries to set the specified lock by adding an entry to the lock table. If another transaction owns a conflicting lock, then the *lock manager* adds the locks to a waiting queue list for that data item. The *unlock operation* releases the specified lock, and grants any waiting lock requests that are no longer delayed.

In addition, the *lock manager* also deals with the *deadlock* problem. Deadlock prevention is performed systematically each time a transaction must wait. The *lock manager* maintains a WFG which shows *who is waiting for whom* as described in Section 3.4. Whenever one transaction has to wait for another to release the lock, it is inserted in the WFG if it does not cause deadlock. On the other hand, if it does cause deadlock, the *lock manager* will send an abort message back to the client process via the *transaction manager* to abort the transaction. The locks owned by the transaction will be released and the transactions waiting for the released locks will be awakened. Further, the data created by the transaction in the cache space in the *transaction manager* will be removed.

4.6 Other Transaction-Specific Modules

4.6.1 Monitor Manager

Some failures may occur due to the abortion of a transaction. For example, the execution of an application program, which embeds RQL query statements, could be terminated by a user. Also, an interactive user may enter *Ctrl-C* to *break* the RTM interface (client) process during the transaction processing. If the central processes,

the *transaction manager* and the *lock manager*, are unaware of this situation, locks held by the aborted transaction may remain in the lock table and those transactions waiting for acquiring the locks will be idled until the central processes are terminated. This kind of fault should be prevented in a multi-user DBMS. This is solved in the following way.

- The *monitor manager* regularly sends a *detection signal* to the *transaction manager* at short time intervals.
- On receipt of the signal, the *transaction manager* looks through the *process IDs* of the *client* processes that have passed requests to it and checks if those processes are still running.
- If any of the client processes has terminated, the information relating to the process kept in the *cache manager* is treated as *garbage* and *collected.* That is, the information created by the aborted client process is discarded.
- The *transaction manager* then sends a message to the *lock manager* to release all locks created by the aborted client processes.

Therefore, storage used by the aborted client processes is deallocated and can be reused. Related locks are also released. Other transactions are no longer blocked, waiting for locks held by the aborted transactions.

4.6.2 Terminator

The multi-user REQUIEM has three processes[7] running at all times. They are run in the background to control all requests sent by client processes. Any of the central processes may, however, terminate abnormally due to faults in the RTM programs, inconsistent data input, or other reasons. When one of the central processes is aborted, other processes should also be aborted and restarted since the RTM can only proceed when all the central processes are running consistently. Further, the

[7]The three processes are the *transaction manager* process, *lock manager* process and the *monitor manager* process.

shared memory and semaphores[8] created for interprocess communication should be removed from the system kernel[9]

The central processes may be terminated manually by the *terminator*. It does not only terminate the *transaction manager* and the *lock manager* processes, but also discards the *shared global memory* and *semaphores* in the *UNIX kernel*.

[8] *Shared memory* and *semaphores* are used to enable *interprocess communication* in the *UNIX* operating system. We will describe more details in the following chapter.

[9] Since the RTM is implemented on UNIX System V, the system kernel means the kernel of UNIX System V.

Chapter 5

Comparison of REQUIEM with other Relational DBMSs

This chapter describes the transaction management mechanisms used in conventional relational database management systems (RDBMSs), such as DB2, INGRES, and ORACLE, and compares them with the transaction management of REQUIEM. The comparison is made to illustrate the superiority of the design principles of RTM.

Although most of these systems support both implicit (automatic) and explicit (manual) transaction management, the comparison concentrates on the implicit part of the transaction management. The design of the explicit transaction management could be very flexible and is usually based on the design of the implicit transaction management.

We study transaction management mechanisms in these relational DBMSs because:

- REQUIEM is a relational DBMS and it is desirable to compare the design of the RTM with similar systems.
- All these relational DBMSs support a multi-user database environment.
- The concepts behind transaction management in the RDBMSs are the same as in non-relational DBMSs.

These three RDBMSs are chosen as they are the most well known RDBMSs de-

veloped. The transaction management in each of them differs in many aspects. However, the mechanisms they used are quite similar. The historical background of these three DBMSs is described in Section 5.1. The four problems used in the previous chapter are also used through Section 5.2 to Section 5.5 to illustrate the comparison of transaction management between the RTM and these relational systems. A comparative table of the RDBMSs is summarized in Section 5.6.

5.1 Brief Overview of the RDBMSs

5.1.1 DB2

DB2[1] (IBM Database 2) is an IBM product. It was designed for large configurations and targeted for on-line processing applications. DB2 provides the query language SQL for access to its databases. SQL was initiated in 1975 and implemented in the prototype language System R.

System R was developed at the San Jose Research Laboratory, California, between 1975 and 1984. It is an experimental DBMS, built to demonstrate the advantages of a system based on the relational model and its feasibility in real environments.

The most significant contribution of System R was "*the design and specification of SQL, a unified language for relational database management, and efficient techniques for SQL query compilation and transaction management*" [Valduriez and Gardarin 89]. The major impact of System R on the RDBMS market was evidenced by the fact that most RDBMSs support or are committed to support the standard SQL language.

The latest IBM products, including DB2 and SQL/DS, are both derived from System R.

[1] A detailed description of DB2 can be found in [Cheng *et al* 84], [Crus 84], and [Teng and Gumaer 84].

5.1.2 INGRES

INGRES[2] (INteractive Graphics and RElational System) was developed at the University of California, Berkeley by Michael Stonebraker, Eugene Wong, and Lawrence Rowe, from 1973 to 1983 [Stonebraker 86]. It has since been developed and marketed in a commercial version by Relational Technology Inc. (recently renamed Ingres Corporation) [Date 90].

The most significant contribution of INGRES was "*the design and specification of QUEL a unified language for data description, manipulation, and control, and the development of original solid solutions to semantic data control and query optimization*" [Valduriez and Gardarin 89]. QUEL is a relational query language which is more regular and powerful than SQL ([Date 90], [Valduriez and Gardarin 89]). However, the popularity of SQL has encouraged the commercial version of INGRES to provide an SQL compatible interface.

The transaction management in the commercial INGRES product has been completely redesigned from the prototype's solutions [Date 90]. In this thesis, we examine the transaction management in the commercial version of INGRES as it is the most recent and is available commercially.

5.1.3 ORACLE

ORACLE[3], which has been marketed by Oracle Corporation since 1979, is one of the most popular RDBMSs in the computing market. It supports the SQL language as the way to access its database. ORACLE "*optimizes the execution of the relational operations using efficient storage structures, in particular, multirelation clustering*" [Valduriez and Gardarin 89]. It also comes with a set of *fourth-generation language* (4GL)[4] tools such as report generator that interface the RDBMS through SQL.

ORACLE comprises a 4GL, a form-oriented application generator, a report

[2]INGRES information is from [Stonebraker 86], [INGRES 90], [Date 90], and [Valduriez and Gardarin 89].

[3]ORACLE is described in [Oracle 88] and [Valduriez and Gardarin 89].

[4]A typical 4GL includes query language commands, control capabilities (IF, GOTO, WHILEDO, FOR), variable manipulation, and error handling. Compared with a 3GL such as COBOL, a 4GL is essentially nonprocedural [Valduriez and Gardarin 89].

writer, a database design help facility, a color graphics facility, and an easy-to-learn spread-sheet. Further, it is available to run on a range of equipment and operating systems.

The transaction management in ORACLE provides a SNAPSHOT mechanism to support *read consistency*[5]. No locking is necessary for read transaction. This is described in the following section.

5.2 Concurrency Control

The transaction management in the three DBMSs are similar in that they all use locking to solve their concurrency control problems. However, the locking mechanisms differ in significant detail.

5.2.1 DB2

DB2 uses locks to minimize interference between concurrent users and to prevent them from accessing inconsistent data. Two phase locking ensures that transactions run by different users will not interfere.

When a user requires a lock, if another user already holds a conflicting lock on the same resource in question, the user must wait. Locks are held until a transaction is committed. In practice, the installation of DB2 may specify a maximum wait time. Then if any transaction reaches this limit when waiting for a lock, the transaction is aborted and a *time out* error code is returned.

In DB2, locks are of two kinds, namely read (shared or S) locks and write (exclusive or X) locks[6].

The primary consideration in locking is the virtual storage required for each outstanding lock, which is approximately 200 bytes [Cheng *et al* 84]. If one transaction locks 10,000 pages in shared mode (read lock), roughly two megabytes of virtual

[5]Read consistency in ORACLE means that read transactions can be run without interfering with any other transactions and always in a consistent state.

[6]In fact, DB2 provides additional types of locks which are described in Section 5.3.1.

storage are taken up by the transaction. DB2 uses *lock escalation*[7] to solve the virtual storage space problem.

When a transaction reads a record, it automatically acquires a read lock on the record. When a transaction successfully updates a record, it automatically acquires a write lock on that record. If it already holds a read lock on the record, then the update *upgrades* the read lock to a write lock. Locks are released only when the transaction commits.

5.2.2 INGRES

INGRES[8] also uses locking to ensure that transactions do not interfere with each other. To ensure consistency, INGRES automatically places locks on data items when users submit queries to access data.

The locking policy in INGRES is also two phase locking. In INGRES, transactions can be committed by either issuing the *commit* statement after one or more queries, or setting *autocommit* which causes a commit to occur after every SQL query that was successfully executed during the session.

An upper limit on the total number of locks is determined when INGRES is installed [INGRES 90]. A counter is decremented by one whenever a lock is taken. This counter reflects the number of locks available. If all the locks have been taken when INGRES receives a lock request, the request will be blocked (i.e. wait) until a lock is freed. This indicates that the locks in INGRES consume a notable amount of space in the volatile storage.

When a user issues a read transaction such as a **select** statement to access some data, a *read lock*[9] is required. When a user issues a write transaction that writes to the database, such as an **update**, **insert**, or **delete** statement, INGRES then requires a *write lock* for it. In such case, other transactions can not access (neither read nor write) the data locked by the transaction. Locks are released when the

[7]DB2 lock escalation is described in Section 5.3.1.

[8]The commercial version of INGRES is discussed here.

[9]In INGRES, *shared locks* and *exclusive locks* are the term used for *read locks* and *write locks*, respectively. However, read and write locks are used here for consistency.

transaction commits. The modes (types) of locks are detailed in Section 5.3.2.

5.2.3 ORACLE

The concurrency control algorithm in ORACLE is also based on two-phase locking. A transaction in ORACLE is defined as a set of SQL statements or a single SQL statement. Transaction commitment or abortion must be specified by *commit* or *rollback* statements within a transaction if an *autocommit* is not issued. Several *commit* statements may be specified within the same long transaction. If an *autocommit* is issued, the transaction management is controlled automatically. Users do not specify any explicit commands to commit a transaction in such case. Like INGRES, ORACLE's *autocommit* command enables the commit action as soon as the execution of a query statement is completed.

Read consistency is the significant consistent model supported by the ORACLE RDBMS. It means that the read transaction in ORACLE never needs locking control. The ORACLE RDBMS always establishes a read consistent snapshot of the data at the time the read transaction is executed. Read consistency [Oracle 88]:

- guarantees that the data seen by a query statement (such as a read transaction) will not change during statement execution.
- assures that read transactions do not wait for write transactions with the same data.

ORACLE's read consistency model is a "multi-version" consistency model because multiple versions of the same table may appear to exist simultaneously. The characteristics of the read transaction in ORACLE are [Oracle 88]:

- A read transaction requires no locks.
- A read transaction never waits for any data locks to be released; it can always proceed.
- A read transaction always returns a read consistent view of the data.

- Other transactions can access the table being queried by a read transaction, including the particular rows being queried.

That is, a read transaction never has to wait for the data resources, even when the underlying data is being accessed or altered by other transactions. No data locks are required for read transactions.

For write transactions, locks are required for the modified data. When write transactions are started, the *write locks* on the tuples to be changed are acquired. Other transactions can read the tuples to be modified (supported by a snapshot of the data), but can not alter them until the current transaction has released the locks. Other transactions can either read or write other tuples in the table without being affected by the locks. The locks are released when the transaction commits.

5.2.4 Discussion

A comparison of concurrency mechanisms between the RDBMSs and the RTM is shown in Table 5.1. *Two phase locking* is used in all RDBMSs and the RTM to

RDBMS / categories	DB2	INGRES	ORACLE	REQUIEM
Concurrency Control	2PL	2PL	2PL	2PL
Read Consistency	No	No	Yes	Yes
Locks Limitation	Yes	Yes	Yes	Flexible

Table 5.1: The Comparison of Concurrency Control Between RDBMSs

control multi-user concurrent access. Most RDBMSs store their locks in virtual storage which makes the locking operations (acquire and release a lock) more efficient. However, the number of locks are limited by the virtual storage. For experimental

reasons, the RTM records the locks in the database which provides more space for more locks, but affects the performance. A new structure is proposed in the next chapter which stores locks in virtual storage, but allows a large number.

The RTM also provides *read consistent* data as described for ORACLE concurrency control. However, there are no *read* locks which simplifies the locking operations and reduces the number of deadlocks. In the RTM, write locks are enough for dealing with concurrency control for transactions. A detailed comparison is made in Table 5.1.

5.3 Granularity Control

5.3.1 DB2

The implicit locking mechanism of DB2 is defined in terms of record granularity locking. However, the definition is a *logical* definition. *Physically*, DB2 locks data in terms of either page or tablespace[10]. That is, when a given transaction logically locks some individual record, DB2 physically locks either the page or the tablespace that contains the record depending on the granularity determined by DB2 at the time of table creation. Table 5.2 shows the types of locks provided by DB2.

Read (shared or S) locks and write (exclusive or X) locks are provided to restrict access to data. Intention read (IS) and intention write (IX) locks are provided to make possible the use of multiple granularity locks. That is, before an S or X lock is applied to a data item, an IS or IX is first acquired on the item of the next larger granularity which contains the smaller item.

By default, DB2 will decide the appropriate physical unit of locking. DB2 may acquire page locks initially, and then trade all those locks in for a single tablespace lock, if the number of page locks reaches some specified threshold (lock escalation[11]).

[10]Tablespace is a DB2 object in which one or more tables are stored.
[11]Lock escalation has been described in Section 2.3.2.

Mode of Lock	Description
X	*write* (or *exclusive*) locks. A transaction can not tolerate any concurrent access to the data locked by itself. The transaction, however, may update the data locked.
S	*read* (or *share*) locks. A transaction can tolerate concurrent readers, but not concurrent writers, on the data locked by itself. The transaction will not update the data locked by the read lock.
IX,IS	*intention write* and *intention read* locks. Whenever a write (*X*) or read (*S*) lock is acquired on an item, say X, DB2 acquires an intention write (*IX*) or intention read (*IS*) lock on the item of next greater granularity which contains X. They are used to guarantee the stability of X while it is being processed.
SIX	*intention read write* (or *shared intention exclusive*) locks. This combines read (S) and intention write (IX) locks. That is, a transaction can tolerate concurrent readers, but not concurrent writers, on the locked data, say Y. The transaction may update the individual data in Y and will therefore set write locks on the data granularity.

Table 5.2: The DB2 Locks Table.

Mode of Lock	Description
X	*write* (or *exclusive*) locks. Only one user may hold a write lock on a resource at any given time. (A "resource" is a data page or table on which a lock can be taken.)
S	*read* (or *share*) locks. Multiple users may hold read locks on the same resource at the same time. No user can update an object which is read locked.
IX,IS	*intention write* and *intention read* locks. Whenever INGRES takes a write (*X*) or read (*S*) lock on a page within a table, INGRES takes an intention write (*IX*) or intention read (*IS*) lock on the table. INGRES uses these locks to determine whether it is possible to take a lock on the table. These locks do not actually lock users out of the table.
SIX	*intention read write* (or *shared intention exclusive*) locks. The INGRES DBMS server's buffer manager uses these locks to manage its page cache.
N	*Null* locks. INGRES locks that do not block any action but preserve the number in the value block of the locks.

Table 5.3: The INGRES Locks Table.

5.3.2 INGRES

INGRES locks can be of various modes. Table 5.3 [INGRES 90] shows the modes of locks in INGRES and describes the usage of them. Basically, *read* and *write* locks are provided for users to acquire access privileges. Read locks ensure that no one can change the data locked by the transaction, although it can be read by other transactions. Write locks ensure that no other transactions can read or write the data locked.

Locks on different granularities are available in INGRES. This is shown in Table 5.4 [INGRES 90]. By default, locks in INGRES are page locks. The intention locks are similar to those (*ir* and *iw*) described in Section 3.2. Lock escalation is

SQL Operation	Comment	Lock Mode	Lock Level
Select	For each table involved in the **select**:	*IS* and *S*	Table Lock Page lock(s) on pages in table
	If query touches > **maxlocks** pages, INGRES takes a read table lock rather than page locks	*S*	Table lock
Update, Insert, or **Delete**	Table **update, insert**, or **delete**	*IX* and *X*	Table lock Page lock(s) on pages in table
	If query touches > **maxlocks** pages	*X*	Table lock
	For other tables used in query but not being changed	*S*	See lock for **select** statement
Create table	On table	*X*	Table lock
Drop table	On table	*X*	Table lock

Table 5.4: The INGRES Locking Levels Table.

also provided in INGRES for which a *maxlocks* parameter is used. The default for *maxlocks* is 10. This number is changeable depending on the user's need. The use of *maxlocks* is described below.

If INGRES estimates that a transaction will be touching more than the number of pages to which *maxlocks* is set, the transaction will start out with a table lock. For example, in a query that is not restrictive and does not use a key to locate affected tuples, where INGRES decides that the scanning the entire table is required, it takes a table lock at the beginning of query execution.

Further, if the transaction locks more than *maxlocks* pages within a table, INGRES will then escalate locks to table locks for the transaction. In such case, three

steps will be done for escalating the locks [INGRES 90]:

1. Stop accumulating page locks
2. Escalate to a table lock
3. Drop all page locks that have been accumulated

INGRES also escalates to table locks in an attempt to complete a query if it has run out of locks.

The *intention read* (*IS*) and *intention write* (*IX*) locks on the table are used to determine whether a table lock can be taken on that table as follows:

- An *IS* lock on the tables means that a read lock has been taken on at least one page of the table; nevertheless, a read lock, if available, can still be taken at either the page or table.
- An *IX* lock on the table means that a write lock has been taken on at least one page of the table; no table lock can be taken on the table by another user until the current page lock(s) has (have) been released.

5.3.3 ORACLE

Two types of granularities are provided in ORACLE: *tuple* and *table*[12]. All modes of locks are described in Table 5.5 [Oracle 88]. Read and write locks are the same as the ones described in the previous section.

However, multiple granularity is not supported implicitly (automatically). Instead, users have to issue explicit commands to specify levels of locks if they want to change the default granularity locks. The usage of locks are listed in Table 5.6 [Oracle 88].

The default locking granularity is on tuple level. Tuple granularity locks increase concurrency, but also increase locking overheads. If a large number of tuples, say 10000, are going to be changed (or deleted), then 10000 locks will be acquired by

[12]The locks provided by ORACLE are ROW (tuple) and TABLE granularity. We standardize the names of granularities in terms of the terminology used in this thesis for comparison reasons.

Mode of Lock	Description
X	*write* (or *exclusive*) locks. Write locks allow queries on the locked resource but prohibit any other activity on the resource (table or tuple).
S	*read* (or *share*) locks. Read locks allow queries but prohibit updates to the table.
RS	*intention read* locks. Intensive read locks allow concurrent access to a table. They prohibit other users from locking the entire table for exclusive access.
RX	*intention write* locks. Intensive write locks are the same as intention read locks, but also prohibit locking in read read (*S*) mode. These locks are obtained when updating, inserting, or deleting.
SRX	*intention read write* (or *intention tuple exclusive*) locks. Intensive read write locks are used to look at a whole table to do selective updates and to allow other users to look at tuples in the table but not to lock the table in read mode or to update tuples..

Table 5.5: The ORACLE Locks Table.

SQL Operation	Tuple Lock	Table Lock
SELECT	none	none
SELECT FOR UPDATE	*X*	*RS*
INSERT	*X*	*RX*
UPDATE	*X*	*RX*
DELETE	*X*	*RX*
CREATE	none	*X*
DROP	none	*X*

Table 5.6: The ORACLE Locking Levels Table.

the transaction. The locks require more space for setting and, since there are more locking operations, performance is reduced.

The ORACLE RDBMS does not escalate locks, but may upgrade read locks to write locks.

5.3.4 Discussion

As explained in the previous chapter, three types of granularities are provided in the RTM: *tuple*, *page*, and *table*. All modes of locks for the RTM are described in Table 5.7. No read locks are needed since read consistency is supported by the RTM.

Mode of Lock	Description
w	*write* locks. Only one user may hold a write lock on a resource at any given time.
iw	*intention write* locks. Whenever the RTM takes a write (*w*) on a tuple within page X and table Y, the RTM takes an intention write (*iw*) lock on both page X and table Y. The RTM uses these locks to determine whether it is possible to take a lock on the page or table. These locks do not actually lock users out of the page or table.

Table 5.7: The REQUIEM Lock Mode Table.

Moreover, a *query analysis module* is provided to support the multiple granularity locking.

A comparison of granularity control between the previously described RDBMSs and the RTM is shown in Table 5.8. From this table it can be seen that DB2 and INGRES both provide lock escalation and the default granularity of locks is *page*. The disadvantage of the lock escalation is that it may result in deadlock. By default, ORACLE locks the *tuple* to allow more concurrency, but requiring more space for setting the locks.

The RTM balances the granularity trade-offs by employing the *query analysis module* to determine the granularity for each transaction.

RDBMS / categories	DB2	INGRES	ORACLE	REQUIEM
Concurrency granule	Variable	Variable	Relation or tuple	Variable
Lock Escalation	Yes	Yes	No	Yes

Table 5.8: The Comparison of Granularity Control Between RDBMSs

5.4 Deadlock Control

5.4.1 DB2

Since locking is employed to provide concurrency control in DB2, deadlock may occur. DB2 has been designed to check for deadlock situations when transactions are going to be blocked and to abort and rollback[13] one or some transactions (candidates[14]) if deadlock is detected in advance. This known as deadlock prevention. Each time a transaction waits, a matrix (representing a WFG) of who is waiting for whom is examined, and deadlock cycles (if any) are detected. The selection of a victim is based on the relative ages of transactions in each deadlock cycle. In general, the DB2 RDBMS selects the youngest transaction as the victim since youngest transaction usually holds less locks and is more easily recovered from.

5.4.2 INGRES

When a potential deadlock is discovered in INGRES, the locking system in INGRES aborts the one transaction, allowing others to continue. The user whose transaction was aborted receives an INGRES error [INGRES 90]. The transaction will then be

[13]The rollback operation in DB2 is described in Section 5.5.

[14]The candidates are one or more of the transactions involved in a deadlock cycle which are going to be aborted.

rolled back[15]. This is also the deadlock prevention.

5.4.3 ORACLE

Deadlock prevention is used in ORACLE to solve the deadlock problem. Whenever a transaction is about to go into the waiting state (waiting for other transactions to release a lock), ORACLE checks if deadlock will occur. If so, ORACLE will roll it back[16].

5.4.4 Discussion

A comparison of deadlock control between the RDBMSs and the RTM is shown in Table 5.9. Deadlock prevention is used by the RTM. A WFG structure is maintained

RDBMS / categories	DB2	INGRES	ORACLE	REQUIEM
Deadlock prevention	Yes	Yes	Yes	Yes

Table 5.9: The Comparison of Deadlock Control Between RDBMSs

in the lock manager in the RTM. Before a transaction is added to the WFG, it is determined whether a deadlock would be caused. If so, the transaction is aborted, and all locks held by the transaction are released and granted to other transactions. Otherwise, the transaction is added to the WFG.

Deadlock prevention is used for all the RDBMSs and the RTM because it is simple, efficient, and can be easily implemented.

[15]INGRES rollback operation is described in Section 5.5.2.
[16]We describe the roll back operation of ORACLE in Section 5.5.3.

5.5 Recovery Control

5.5.1 DB2

The description of the data recovery in DB2 is detailed in [Crus 84]. This section briefly describes the recovery control in DB2. DB2 [Crus 84] uses the DB2 recovery log to record both system status information and information describing changes that are made to the database. The DB2 recovery log consists of an *active* portion, an *archive* portion, and a special data set called *Bootstrap Data Set* (BSDS). The active log resides in volatile storage[17] so that when recovering from transaction and system failures, only the information stored in the active log (in main memory for transaction failure and in disks for system failures) need be referenced.

The archive log is allocated when an active log becomes full, and the data in the active log is stored in the archive log. After this is completed, the active log is available for further use.

During recovery, DB2 must determine which active or archive log contains the needed data. This information is kept in the BSDS along with information used to help restart the system after it has determined whether terminated it normally or abnormally. The BSDS is archived by an archive operation along with the active log.

DB2 always begins the restart process at the last complete checkpoint, a known point in the log.

In DB2, media recovery consists of starting with a copy of the data (called an image copy) which was made at a checkpoint in the log, and redoing all committed updates using the appropriate redo information on the log up to the point of failure.

5.5.2 INGRES

The INGRES logging system is used to recover the databases. It keeps track of all database transactions automatically. It is comprised of the *logging facility*, a

[17]The detailed buffer management is described in [Teng and Gumaer 84].

recovery process and an *archiver*.

- The logging facility includes a transaction log file and shared memory that contains the logging database. INGRES has one global transaction log file that keeps track of all INGRES transactions for all users. The logging facility logs INGRES transactions and manages the logging file. It ensures that log records are written in a way that makes them accessible to the recovery and archiver processes. These processes manipulate the data in the transaction log file in case that certain events occur. For example, after a transaction is committed, the logging facility moves the log buffer in the shared memory to the transaction log.

- The recovery process handles on-line recovery from system failures and transaction aborts caused by user actions. INGRES writes consistency points (checkpoints) into the transaction log file to indicate that all data in databases are consistent up to that point and to allow on-line recovery to take place when a failure is detected.

- The archiver process deals with the media failures. It removes completed transactions from the transaction log file and writes them to the corresponding journal files. These contain records of all the changes made to the database since the last checkpoint was marked. The archiver process idles until sufficient portions of the transaction log file are ready to be archived or until the database is removed from the logging system. The archived data is used for recovering the databases from media failures.

5.5.3 ORACLE

A temporary workspace called the *database buffer pool* is used for updating the data rather than directly changing the actual database. The most recently changed data are stored in the database buffers, whether or not the data have been committed. This allows users to rollback the transaction easily and allow the database to support many concurrent read consistent views of the same tables.

To reduce I/O, buffers are written to disk only when necessary, such as when the buffer is full, using the *least recently used* (LRU) algorithm. Buffers that have not been used for the longest time are written to the database. Some data which is not committed but resides in the least recently used buffers, may be written to the database. Therefore, the database may contain some uncommitted data at any given time. Two types of structures are used to solve the resulting inconsistent data problems: *rollback segment* and *redo log.*

- Redo log is used for redoing transactions. The redo log is a set of operating system files external to the database that record all changes, including uncommitted, rollbacked, or committed data, made to the database. During recovery, all changes in the redo log are applied to the database. After redo, the database contains all committed changes as well as any uncommitted changes from transactions in progress at the time of the failure. That is, the redo is only half of recovery, since the uncommitted changes after redo should be undone.
- Rollback segments are used for undoing transactions. After the redo log files have been applied and all changes made to the database, the rollback segments are used to undo the uncommitted changes. The rollback segments are used to identify and undo transactions that were active at the time of a failure.

Both actions of redo and undo are necessary for ORACLE to recover from system failures.

Media failure involves the same steps as system recovery (redo and undo). The primary difference is that media recovery may need to start at an earlier point in time and thus may require older redo log files. This is made possible by keeping the copies of the database and redo log files on tapes.

In addition, whenever the database is not used (no data of the tables is in any of the database buffers), it is saved to tapes. To recover from a media failure, one can simply restore the database when it is not used and then restart the transactions.

Checkpoints are also employed to decrease the time required by system recovery (if it is required) and to allow a redo log file to be reused. All changes prior to the

checkpoint are *safe*, so recovery can resume with only those changes following the checkpoint. Shorter periods between performing checkpoints reduce the time necessary for recovering from system failure, but increase the frequency of checkpoints being taken and have a negative impact on performance. When a redo log fills, a checkpoint should be made so that the redo log files may be reused.

5.5.4 Discussion

A comparison of recovery control between the RDBMSs and the RTM is shown in Table 5.10. The storage management is important as far as recovery control is

RDBMS / categories	DB2	INGRES	ORACLE	REQUIEM
Implementation of updates	In place	In place	In place	In place
Redo Algorithm	Yes	Yes	Yes	No
Undo Algorithm	Yes	Yes	Yes	Yes

Table 5.10: The Comparison of Recovery Control Between RDBMSs

concerned. If the buffers in the virtual memory can be efficiently used, the recovery control could be more efficient and save storage space.

Most RDBMSs store redo and undo information in the log either in volatile storage or in stable storage. After a transaction or system failure occur, the redo and undo operations are required to recover the database. In the RTM, no redo or undo are required for transaction failure recovery. No changed data are written to the database until the transaction commits.

With system failure in the RTM, at most one transaction needs to be recovered, because all transactions are controlled centrally and only one commit operation of a transaction could be interrupted. In general, if system failure does not occur in

any commit operation, no actions are required for system recovery. Recovery merely restarts the RTM and clears the locking tables.

5.6 Comparative Table

Table 5.11 summarizes the transaction management support in the RDBMSs and the multi-user REQUIEM. All systems support concurrency control by two-phase locking with deadlock prevention. ORACLE and the RTM both support read consistency. The number of locks available from RTM is more flexible than the other RDBMSs. The concurrency granule could be variable (relation, tuple, set of qualified tuples) or fixed. Lock escalation is not provided by ORACLE . All systems implement updates in place and provide a mechanism for recovering after media failure, as well as after transaction and system failures. Further, no redo algorithm is required for the RTM recovery. The undo algorithm provided for the RTM is provided only necessary for recovery from system and media failures. Even in the case of these failures it is rarely required.

RDBMS / categories	DB2	INGRES	ORACLE	REQUIEM
Concurrency Control	2PL	2PL	2PL	2PL
Read Consistency	No	No	Yes	Yes
Locks Limitation	Yes	Yes	Yes	Flexible
Concurrency granule	Variable	Variable	Relation or tuple	Variable
Lock Escalation	Yes	Yes	No	Yes
Deadlock prevention	Yes	Yes	Yes	Yes
Implementation of updates	In place	In place	In place	In place
Redo Algorithm	Yes	Yes	Yes	No
Undo Algorithm	Yes	Yes	Yes	Yes

Table 5.11: The Comparison of Transaction Management Between RDBMSs

Chapter 6

Implementation

Multi-user REQUIEM runs on Sun-4 computers under the Unix operating system[1]. The single-user REQUIEM source code is written in the C programming language, as is the RTM. The RTM comprises the following components: the interprocess communication interface, the central DBMS processes, and the terminator manager. The central processes include the transaction manager, the lock manager, and the monitor manager. Some minor changes to single-user REQUIEM were required to support the RTM.

The previous chapters have outlined the design principles and system architecture for the RTM. This chapter details the implementation of the RTM. The procedure for building a kernel which can provide enough interprocess communication facilities[2] to suit the RTM is shown in Appendix A. Appendix B illustrates the installation of multi-user REQUIEM. Several processes are invoked when multi-user REQUIEM is started. The interprocess communication interface provides the communication interface between these processes. Section 6.1 outlines the types of processes which run in the multi-user REQUIEM.

Two primitives *send* and *receive* which are necessary for message communication between processes are described in Section 6.2.

The major components of the RTM, the central DBMS processes, are described

[1]The source code of the single-user version of REQUIEM is also available for Apple Macintosh, IBM-PC compatible systems, and Digital Equipment VAX (ULTRIX).

[2]The implementation of interprocess communication is detailed in Section 6.2.

in Section 6.3. A comprehensive example is also given in this section to demonstrate how the RTM actually handles transaction management by utilizing its data data structure.

Section 6.4 describes the manual way of terminating the multi-user REQUIEM processes. A sample run of *process terminator* is shown in Appendix C.

Some changes made to the original REQUIEM for achieving the multi-user version are described in Section 6.5.

6.1 Types of Processes in the multi-user REQUIEM

In the multi-user REQUIEM, there are two types of processes:

- User Processes: User processes are invoked by the users to connect to the database.

- Server Processes: Server processes are invoked in the background and are used to perform functions on behalf of user processes. They are the transaction manager process, lock manager process, and monitor process. Server processes are not owned by any one user process, but perform functions for all users of a database system. Server processes are invoked when the multi-user REQUIEM is started.

In sum, each user invokes a *user process* to connect to the database. Many user processes may be invoked by different users to access the database. Only one set of the central server processes are run in the background to perform functions on user's behalf of the user process.

6.2 Interprocess Communication Interface

This section outlines how the interprocess communication between user and server processes is implemented[3]. Some important UNIX commands relating to interprocess communication can be found in Appendix D. The C code of the interprocess communication interface for the multi-user REQUIEM is listed in Appendix E.

The following two headers refer to the *send* and *receive* primitives which implement message communication between processes.

```
BOOLEAN send(dstkey, buf, nbytes)        /* send message */
long dstkey;                             /* destination key */
char *buf;                               /* pointer to message data */
int nbytes;                              /* number of bytes in message */
/* return TRUE on success or FALSE on error */

BOOLEAN receive(srckey, buf, nbytes)     /* receive message */
long srckey;                             /* source key */
char *buf;                               /* pointer to message data */
int nbytes;                              /* number of bytes in message */
/* return TRUE on success or FALSE on error */
```

Messages are sent to a *queue*, identified by a long integer called the key. buf[4] points to the message, which is nbytes bytes in length.

The queue is implemented in terms of a shared memory segment with a segment *identifier*. That is, every shared memory segment acts like a message queue. For each such queue, there are two semaphores, one for sending and one for receiving. To send a message, a process acquires the sending semaphore and when it is available, copies the message to the shared segment. It then releases the receiving semaphore to indicate that the message sending is completed. To receive, a process should release

[3]The implementation of the communication in this thesis is specialized for Unix System V. Interprocess communication implementations on other UNIX systems are detailed in [Rochkind 85] and [Stevens 90].

[4]Pointer structure of the C language is detailed in [Kernighan and Ritchie 88].

the sending semaphore to indicate that the shared memory segment is available for sending. It then attempts to acquire the receiving semaphore. If the receiving semaphore is acquired, the process copies the message from the shared segment to its buffer. When a semaphore is not released, a process can not acquire it and must wait until it is available. This is similar to locking operations but different in that there is only one kind of semaphore and no conflict occurs between semaphores.

UNIX System V, the operating system on Sun-4 machines, provides shared memory and semaphores facilities to assist interprocess communication. The default number of semaphores is 10 and the default number of shared memory segments is 100 [Sun 90]. To install the multi-user REQUIEM into the system, the kernel has to be rebuilt to increase the number of semaphores available.

To calculate the number of shared memory segments and semaphores needed, one has to know how it is related to the multi-user REQUIEM processes. In the multi-user REQUIEM, each process needs a message queue to deliver messages. Further, each queue demands a shared memory segment and two semaphores in the kernel. Suppose there are 50 users. Then 52 message queues, which include 50 queues for user processes and 2 for transaction manager and lock manager processes, are essential. Therefore, 52 shared memory segments and 104 semaphores need to be provided by the kernel. Before the multi-user REQUIEM is installed, the number of the shared memory segments and semaphores need to be determined according to the expected number of maximum users. The kernel can be rebuilt according to the instructions listed in Appendix B.

Shared memory is fast because of two reasons [Rochkind 85]:

1. During the transmission, the kernel is involved only in handling semaphores, a very quick operation. The overhead of a message queue is avoided.

2. Data need not be copied from a process's data space to the kernel space and back to the other process. It is only copied entirely within user space by ordinary machine instructions.

For the central DBMS processes, the process queue key of each of them is established in advance and told to each client; an environment variable DBMSKEY

is used for this purpose. That is, each process can get the value of the environment variable DBMSKEY to know the address of the central process. In order to do this, the value of DBMSKEY has to be setup in advance[5].

Each client makes up its own key, which is simply its process number, since that guarantees uniqueness. A client passes its key to the central process with each request for service so the central process knows on which queue to respond. The client key is used by the central process to send responses back to its clients.

One example of the interprocess communication between clients and centralized process is shown in Figure 6.1. There is only one central queue to receive requests from client processes, one by one and in the order of FIFO (first in first out). After

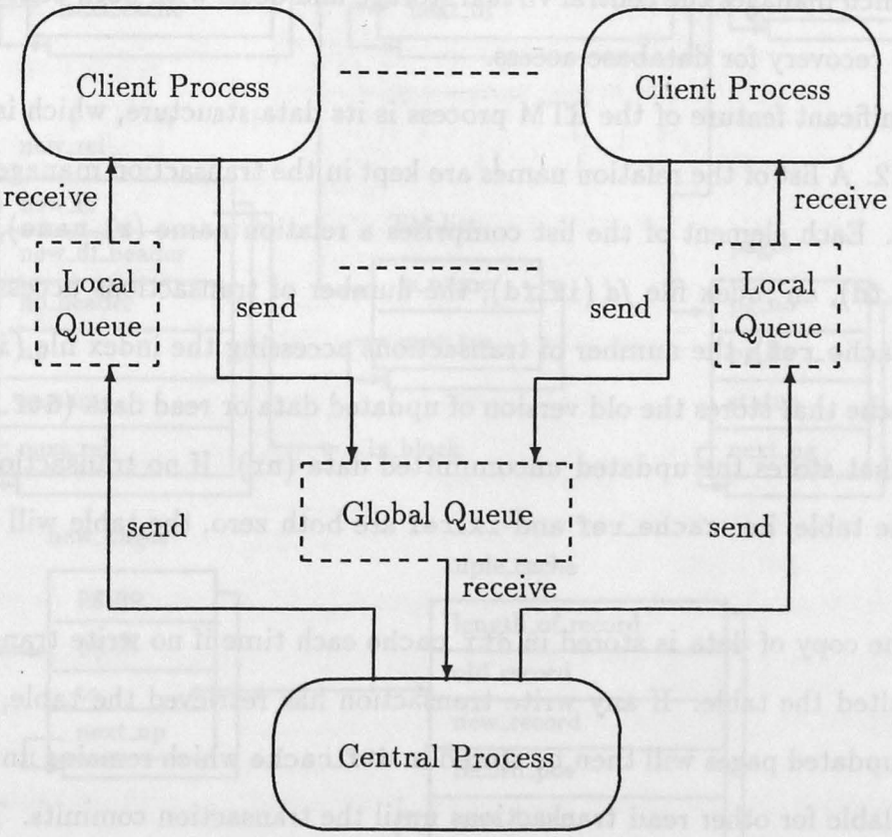


Figure 6.1: The IPC between Clients and Centralized DBMS processes.

processing the operations sent by a client process, the central process of the DBMS returns the result to the client process which requested it.

[5]For further details, see Appendix A.

6.3 Central DBMS Processes

Before a user process starts, the central processes should be invoked. Three central processes, the REQUIEM transaction manager process, the lock manager process, and the monitor process, are run simultaneously in the background. The central processes act as servers for user processes. A discussion on the central processes follows.

6.3.1 The REQUIEM Transaction Manager

The REQUIEM transaction manager process is the main part of the multi-user REQUIEM, which manages the central virtual storage and deals with both concurrency control and recovery for database access.

One significant feature of the RTM process is its data structure, which is shown in Figure 6.2. A list of the relation names are kept in the transaction manager cache (`Tm_cache`). Each element of the list comprises a relation name (`rl_name`), a data file *fd*[6] (`df_fd`), an index file *fd* (`ix_fd`), the number of transactions accessing the data file (`cache_ref`), the number of transactions accessing the index file (`ix_ref`), the page cache that stores the old version of updated data or read data (`dtf_cache`), the cache that stores the updated uncommitted data (`nr`). If no transactions have accessed the table, i.e. `cache_ref` and `ix_ref` are both zero, the table will then be closed.

Only one copy of data is stored in `dtf_cache` each time if no write transactions have consulted the table. If any write transaction has retrieved the table, the old version of updated pages will then be stored in `dtf_cache` which remains unchanged and is available for other read transactions until the transaction commits. The new version of updated data is stored in `nr`. Both versions of data are recorded in the log and the new version is written to the database when the transaction commits. Old version of data is not deleted until transactions which have accessed the data

[6] An *fd* is a small non-negative integer called a *file descriptor*. It is analogous to the file pointer used by the standard C library, and is used instead of the name to identify the file [Kernighan and Ritchie 88].

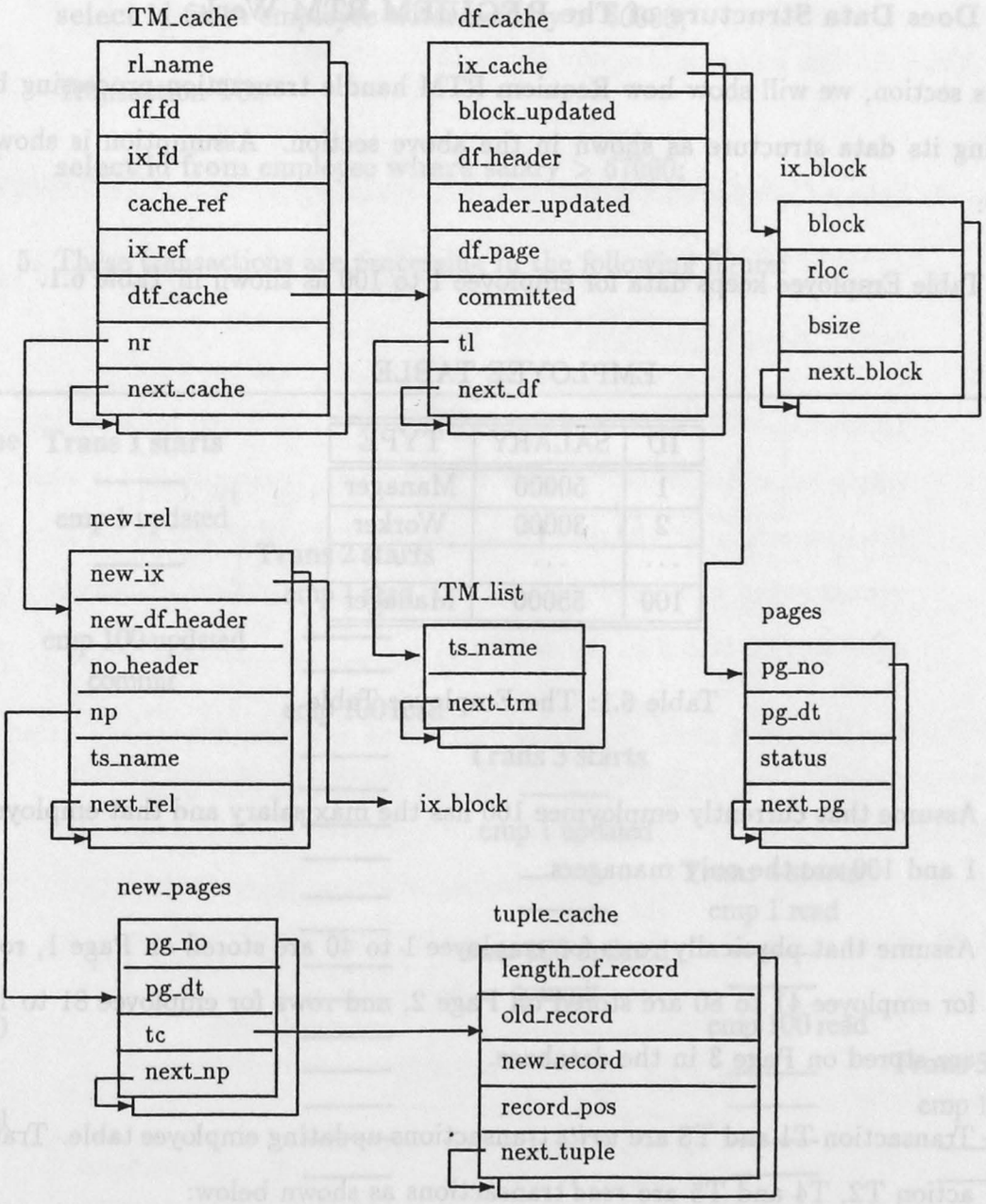


Figure 6.2: The Data Structure of the RTM Process.

have all committed.

The following shows an example of how the REQUIEM RTM actually handle transaction management by using its data structure.

How Does Data Structure of The REQUIEM RTM Work

In this section, we will show how Requiem RTM handle transaction processing by utilizing its data structure as shown in the above section. Assumption is shown below:

1. Table Employee keeps data for employee 1 to 100 as shown in Table 6.1.

EMPLOYEE TABLE

ID	SALARY	TYPE
1	50000	Manager
2	30000	Worker
...	...	...
100	55000	Manager

Table 6.1: The Employee Table

2. Assume that currently employmee 100 has the max salary and that employees 1 and 100 are the only managers.

3. Assume that physically rows for employee 1 to 40 are stored on Page 1, rows for employee 41 to 80 are stored on Page 2, and rows for employee 81 to 100 are stored on Page 3 in the database.

4. Transaction T1 and T3 are *write* transactions updating employee table. Transaction T2, T4 and T5 are *read* transactions as shown below:

   Transaction T1:

   update employee **set** salary = 75000 **where** type = 'Manager';

   Transaction T2:

   select id **from** employee **where** salary > 70000;

Transaction T3:

update employee set salary = 80000 where type = 'Manager';

Transaction T4:

select id from employee where salary > 60000;

Transaction T5:

select id from employee where salary > 67000;

5. These transactions are processing in the following figure:

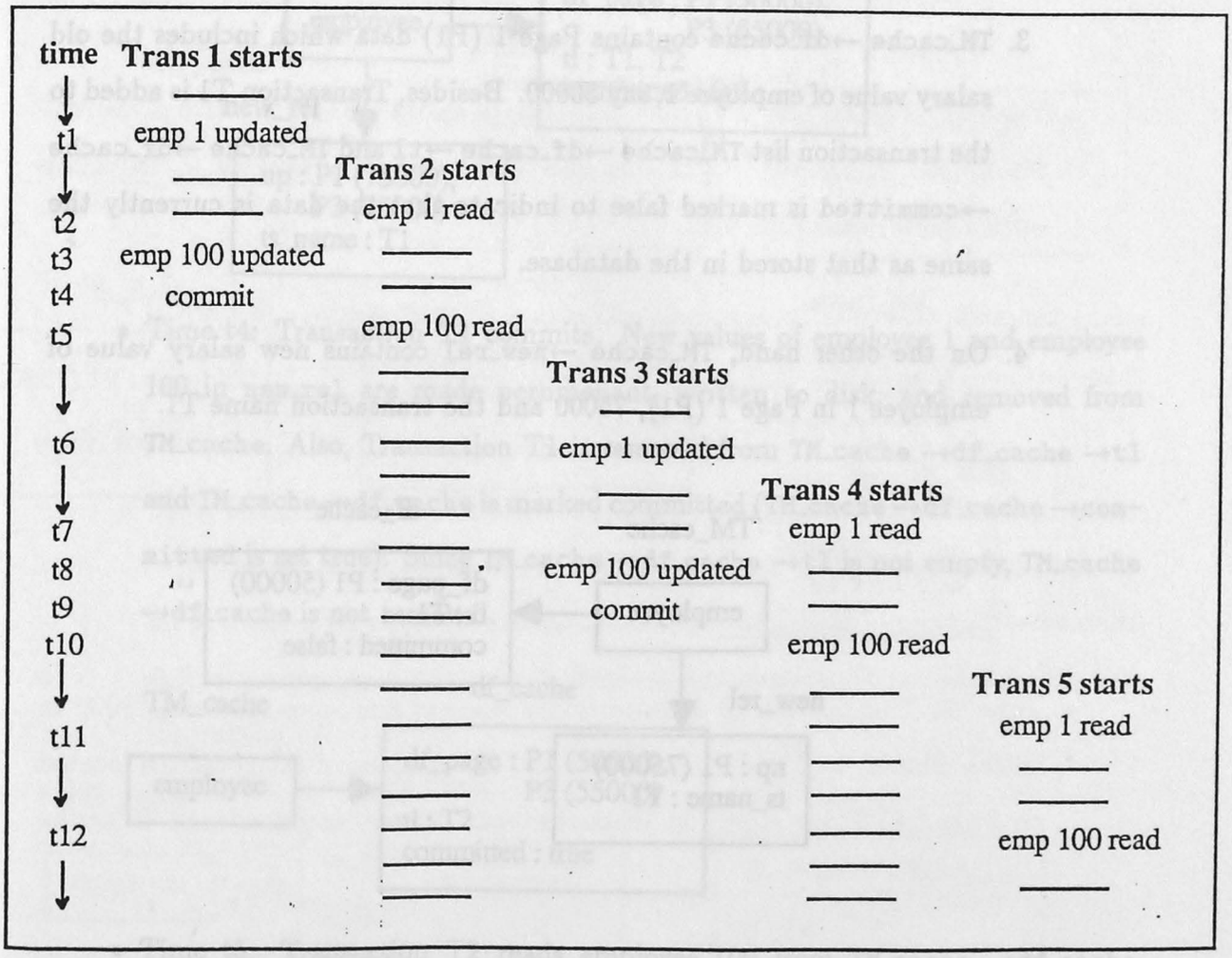


Figure 6.3: The Transactions Timing Sequence.

Time t1 to t12 listed above indicate the time points that data in the RTM data structure changes regards to each transaction processing from Transaction T1 to T5.

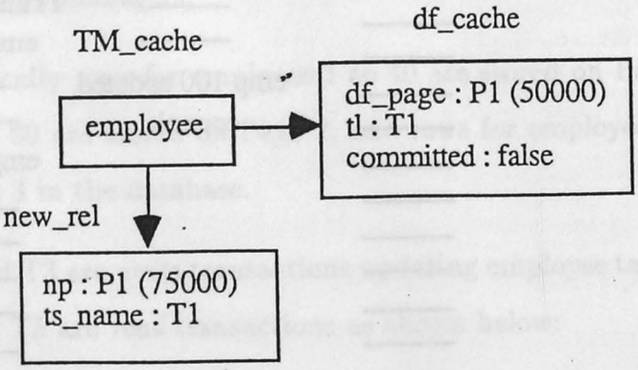
- Time t1: The following diagram shows that:

  1. Salary of employee 1 is updated by Transaction T1 from 50000 to 75000.

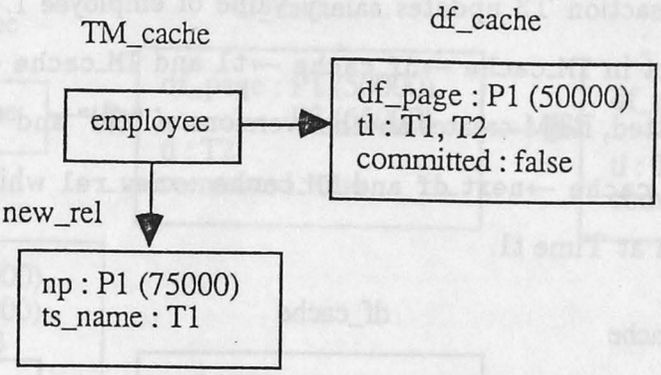
  2. TM_cache stores both old version data in TM_cache →df_cache and new version data in TM_cache →new_rel.

  3. TM_cache →df_cache contains Page 1 (P1) data which includes the old salary value of employee 1, say 50000. Besides, Transaction T1 is added to the transaction list TM_cache →df_cache →t1 and TM_cache →df_cache →committed is marked false to indicate that the data is currently the same as that stored in the database.

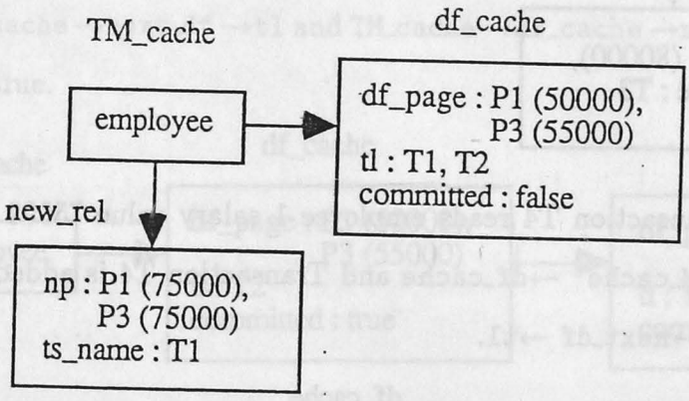
  4. On the other hand, TM_cache →new_rel contains new salary value of employee 1 in Page 1 (P1), 75000 and the transaction name T1.



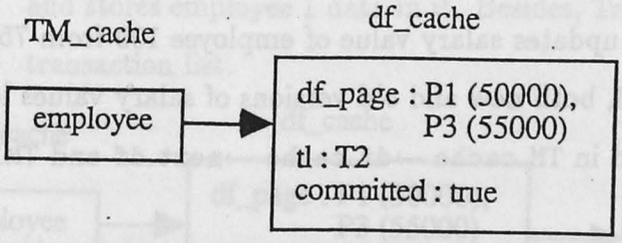
- Time t2: Since employee 1 salary 50000 stored on TM_cache →df_cache is marked uncommitted, it is read by Transaction T2. Moreover, Transaction T2 is added to the transaction list TM_cache →df_cache →t1.



- Time t3: Similar to the situation in C1, both old and new versions of employee 100 salary are stored in TM_cache →df_cache as shown below.

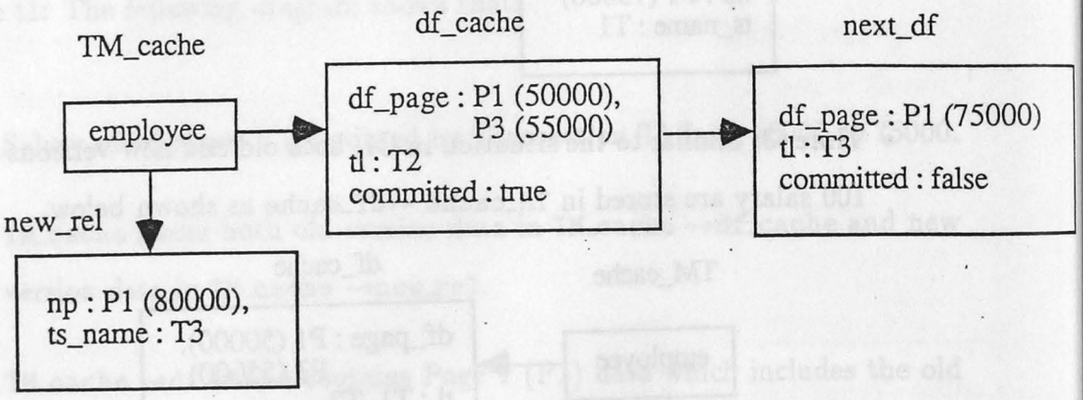


- Time t4: Transaction T1 commits. New values of employee 1 and employee 100 in new_rel are made pernmenant, written to disk, and removed from TM_cache. Also, Transaction T1 is removed from TM_cache →df_cache →tl and TM_cache →df_cache is marked committed (TM_cache →df_cache →committed is set true). Since TM_cache →df_cache →tl is not empty, TM_cache →df_cache is not removed.

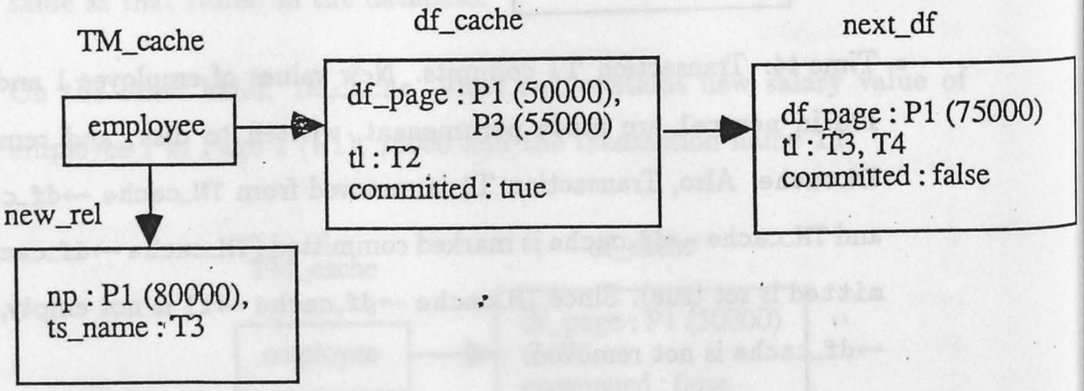


- Time t5: Transaction T2 reads employee 100 from TM_cache →df_cache →df_page since Transaction T2 is in the transaction list TM_cache →df_cache →tl.

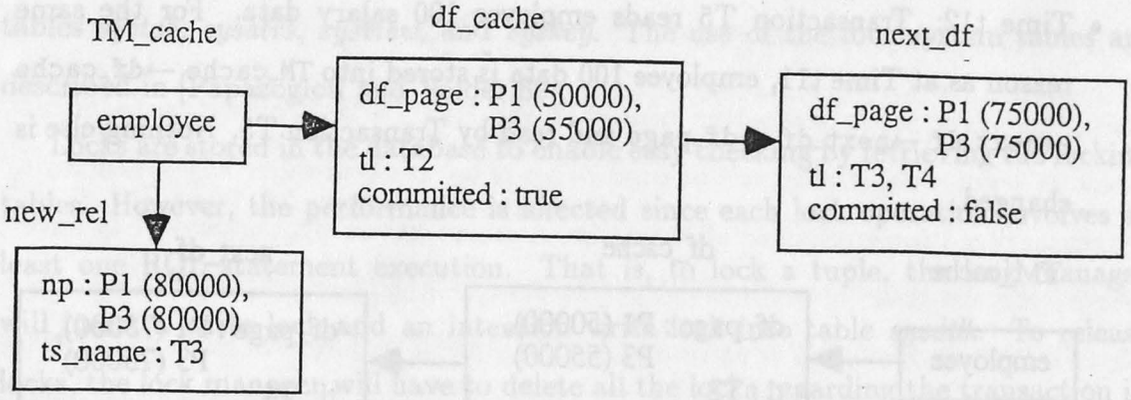
- Time t6: Transaction T3 updates salary value of employee 1. Since Transaction T3 is not in TM_cache →df_cache →tl and TM_cache →df_cache is marked committed, RTM creates another versions of "old" and "new" data on TM_cache →df_cache →next_df and TM_cache →new_rel which are similar to the situation at Time t1.



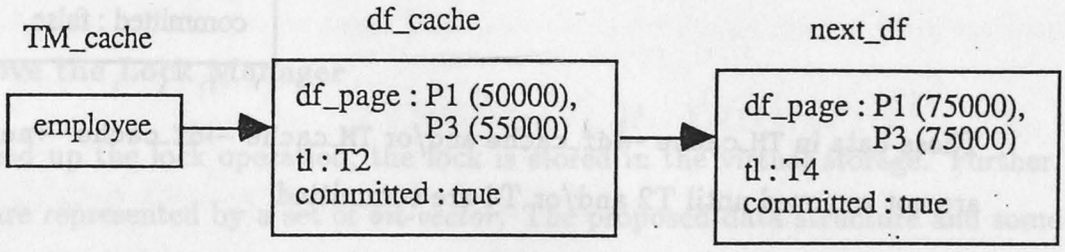
- Time t7: Transaction T4 reads employee 1 salary value 75000 from the uncommitted TM_cache →df_cache and Transaction T4 is added to TM_cache →df_cache →next_df →tl.



- Time t8: Transaction T3 updates salary value of employee 100 from 75000 to 80000. Similar to Time t3, both new and old versions of salary values located on Page 3 (P3) are stored in TM_cache →df_cache →next_df and TM_cache →new_rel.

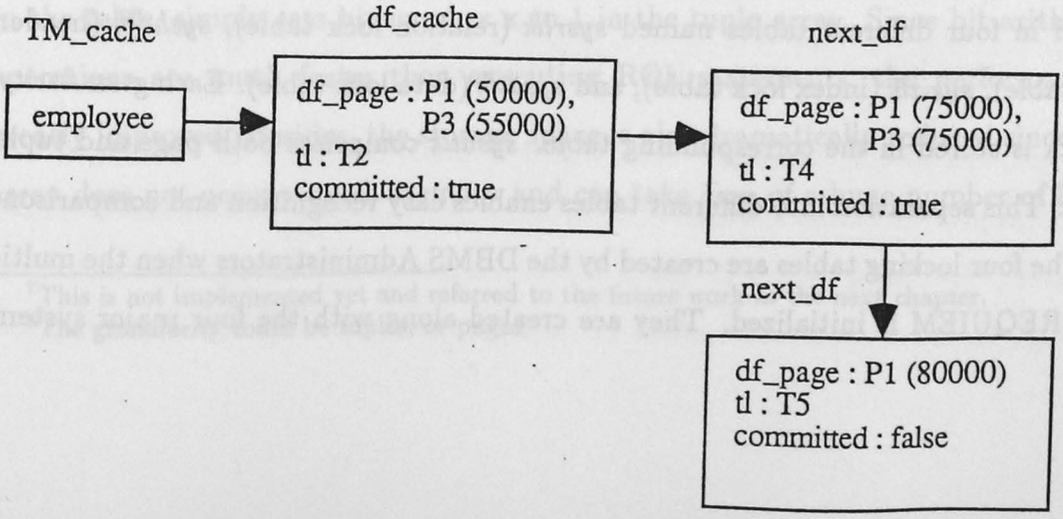


- Time t9: Transaction T3 commits. TM_cache →new_rel data are written to disk and removed from TM_cache. Transaction T3 is removed from TM_cache →df_cache →next_df →tl and TM_cache →df_cache →next_df →committed is set true.

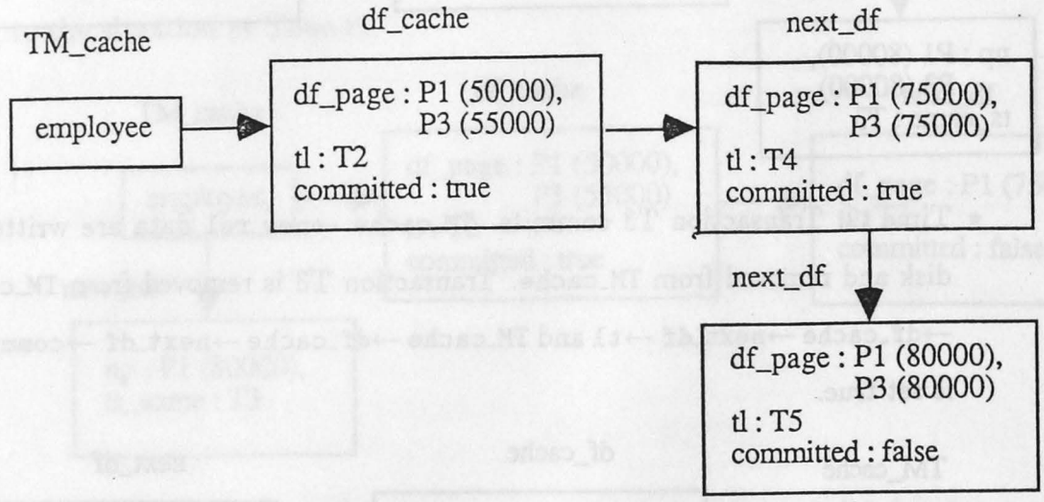


- Time t10: Transaction T4 reads Page 3 employee 100 data from TM_cache →df_cache →next_df since Transaction T4 is in TM_cache →df_cache →next_df →tl.

- Time t11: Transaction T5 reads employee 1 salary data. Since both TM_cache →df_cache and TM_cache →df_cache →next_df are marked committed, the RTM transaction manager creates TM_cache →df_cache →next_df →next_df and stores employee 1 data in it. Besides, Transaction T5 is added to the new transaction list.



- Time t12: Transaction T5 reads employee 100 salary data. For the same reason as at Time t11, employee 100 data is stored into TM_cache →df_cache →next_df →next_df →df_page and read by Transaction T5. Nothing else is changed.



Those data in TM_cache →df_cache and/or TM_cache →df_cache →next_df are not removed until T2 and/or T4 are committed.

In the above example, each transaction always reads the consistent data of employee 1 and 100. Interference between Transaction T1 to T5 is avoided. That is, the RTM guarrantees that read transactions can proceed without being interrupted by any other write transactions.

6.3.2 Lock Manager

Lock manager deals with the locking operations in the multi-user REQUIEM. The current version of lock manager stores locks in the REQUIEM database. Locks are stored in four different tables named *sysrllk* (relation lock table), *syshdlk* (header lock table), *sysixlk* (index lock table), and *sysdtlk* (data lock table). Each granularity of lock is stored in the corresponding table. *sysdtlk* comprises both page and tuple locks. This separation into different tables enables easy recognition and comparison.

The four locking tables are created by the DBMS Administrators when the multi-user REQUIEM is initialized. They are created along with the four major system

tables *systab*, *sysatrs*, *sysview*, and *syskey*. The use of the four system tables are described in [Papazoglou and Valder 89].

Locks are stored in the database to enable easy checking by retrieving the locking tables. However, the performance is affected since each lock operation involves at least one RQL statement execution. That is, to lock a tuple, the lock manager will insert a write lock and an intention write lock into table *sysdtlk*. To release locks, the lock manager will have to delete all the locks regarding the transaction in the locking tables. A method for improving both storage space and performance of locking is proposed here[7].

Deadlock prevention is also performed in the lock manager. Details of deadlock prevention has been described in Section 4.5.

Improve the Lock Manager

To speed up the lock operation, the lock is stored in the virtual storage. Further, locks are represented by a set of *bit-vector*. The proposed data structure and some manipulation functions are listed in Appendix F. To understand what the data structure is, `setword` is first introduced. A setword is a chunk of memory containing 32-bits on UNIX system V. A `set` means a subset of {0,1,...,n-1}, where n is the number of items of a certain granularitie[8] in a table, and is represented by an array of `setwords`. Bit number x in the `set` is 1 if and only if tuple number x in the table is locked. Each array represents a locking table for a relation. Three arrays, representing a tuple locking table, a page locking table and a page intention locking table are needed for one table. Four booleans are used for index locking, header locking, table locking and table intention locking. To lock tuple number x in the table, simply sets bit number x to 1 in the tuple array. Since bit arithmetic operations are much faster than executing RQL statements, the performance is greatly improved. Besides, the storage space is also dramatically reduced since each array does not occupy much memory and can take care of a huge number of locks.

[7]This is not implemented yet and referred to the future work in the next chapter.

[8]The granularity could be tuples, or pages.

6.3.3 Monitor Manager

Monitor manager sends a *monitor request* to the REQUIEM transaction manager process at regular short intervals of time. This is to check whether any uncommitted transaction has terminated abnormally. A name list of transactions that have retrieved the database is maintained in the REQUIEM transaction manager. As soon as the monitor request is received, the REQUIEM transaction manager checks whether any of the transactions has aborted. If so, the cache data that has been read or written by the write transaction is discarded and locks held by the transaction are released. The cache data which has been read by the read transaction but not been read by other transactions is also discarded. That is, resources used by the aborted transaction are deallocated and reused.

6.4 Terminator Manager

A user process may be aborted during transaction processing. Server processes may also be aborted due to inconsistent data input or faults in the RTM. The terminator manager is invoked to terminate the processes and to clean up the obsolete interprocess communication facilities, i.e. shared memory segments and semaphores created by these processes, in the kernel. Examples are shown in Appendix C.

6.5 Overview of Changes To REQUIEM

The implementation of multi-user REQUIEM required a number of changes to REQUIEM. The following sections give an overview of the changes.

6.5.1 The Changes to IO Operations

A user process of the multi-user REQUIEM is much the same as the single-user REQUIEM process except that the IO operations, such as `open`, `close`, `read`, `write`, and `lseek`, have been modified. They are implemented by sending requests to and receiving responses from the central processes rather than directly accessing the

database files. In this way, all IO operations can be centrally controlled by the transaction manager and the concurrency control and recovery can be managed.

6.5.2 Local Memory Manager

As mentioned in Section 4.3, the original REQUIEM IO operations are tuple-based and are considered to be inadequate for the purposes of the multi-user system. To resolve this problem we upgraded the IO operations from a tuple basis to a page basis. This is achieved by changing the local memory manager in the user process. Data is retrieved from the database a page at a time into a cache in the user process local memory. Each page is read into the cache where it remains until access to some tuple not in the cache. In that case, the page containing the tuple is read into cache to replace the previous page. This policy reduces the number of IO operations and improves performance.

6.5.3 The Query Analysis Module

The query analysis module is added to the original REQUIEM. When a query is compiled, the syntax of the query is analysed to determine the proper granularity for the transaction according to the rules in Section 3.3. Since read transactions do not need any locking operations, no granularity should be determined for these types of transactions.

6.5.4 Conclusion

The changes made to the original REQUIEM are minimal and most of the features and functions of the single-user REQUIEM remain intact. A user running the multi-user REQUIEM would not notice any difference from the original single-user system.

Chapter 7

Conclusion

REQUIEM is a research prototype which offers complete relational functionality in a single-user environment. The aim of designing a transaction manager for REQUIEM was to enable databases developed in this system to be accessed by multiple users simultaneously, while maintaining the internal database consistency. The multi-user REQUIEM achieved this goal while by building on top of the existing single-user system and preserving thus its full functionality.

In our view the RTM is a modern transaction manager with advanced features comparable, if not more sophisticated, than those available for commercial systems such as DB2, INGRES and ORACLE[1]:

- The read consistency model is supported by the RTM which causes no interruption to read transactions.
- The types of locks are kept to a minimum to reduce the number of locking operations required.
- The RTM allows a greater number of locks to be in use at any time than the other RDBMSs.
- The support of the RTM *query analysis module* enables full use of the multiple granularity locking mechanism.

[1]A comparison has been made in Chapter 5.

- The task of recovery in the RTM is simple and kept to a minimum. No redo algorithm for failures is required. Undo is only necessary for some cases of system or media failure.

Moreover, a local IO manager for user processes is built into multi-user REQUIEM to upgrade retrieval from tuple basis, as used with the original version, to page basis. This results in conducting less IO operations thereby improving performance.

Recovery is an integral part of multi-user REQUIEM. The same recovery mechanism could be used in the single-user REQUIEM by building a local cache manager to work with the log files system.

Although the multi-user REQUIEM is a working system dealing with concurrency control and recovery, there is still work which needs to be done on the RTM. Future work is described in some detail in the next section.

7.1 Future Work

In this thesis, we have designed and implemented a transaction manager for REQUIEM to enable multiple users to access a database simultaneously without interference. Users can simply run the multi-user REQUIEM in the same way that they run the single-user DBMS without noticing any difference. However, there is some work to be done on the RTM to achieve a more complete multi-user DBMS functionality. Future work may be performed in the areas of database security, explicit transaction management, improvement of the locking mechanisms, and the development of the RTM from a centralized system to a distributed system.

7.1.1 Security of the RTM

Database security is considered with the protection of the contents and logical structure of the database against unauthorized disclosure, accidental modification, or destruction [Papazoglou and Valder 89]. A security mechanism is necessary to provide protection and thus to permit data to be manipulated only by authorized users.

To perform any viable RQL operation the user should hold the appropriate *access privilege* for the operation and tables contents under question.

At system installation time, a specific user should be granted the *database administrator* (DBA) privileges, which means unrestricted control on the contents and the structure of the database. Any other user creating a relation should be automatically granted all applicable privileges on that relation. Any specific user holding the access privilege should be able to grant that privilege on a group of other users, or can cancel a privilege already granted to other users.

Each time when a user attempts to *grant* or *cancel* a privilege, REQUIEM should confirm that the user has the right to the specified relation. If so, the DBMS may add the access privilege to the ones already possessed.

Database security is necessary for a DBMS, especially for a multi-user system. Since data has to be shared by many users, the security control of a multi-user DBMS needs to be integrated. That is, the RTM should ensure that users are allowed to do the things they are trying to do, in order to maintain a more valuable database.

7.1.2 Explicit Transaction Management

Explicit commands to enable multiple transaction processing are also appealing, due to the following reasons as pointed out in [Oracle 88]:

- Users may want to see multiple tables in a consistent state. An example would be one master table and several detailed tables. This can only achieved through explicit transaction management.

- Users may want no one else to be able to change data they have looked at until they have committed their transaction, no matter whether they change the data or not.

- Users may not want to wait for any other transaction to complete. They should be able to request an explicit write lock to lock the relation they want until they complete the tasks.

- Users may want to read the same relation repeatedly in one transaction knowing no data has changed.

These tasks can only be achieved by specifying explicit transaction management commands to the DBMS.

Further, the task of a transaction may not be completed in one query statement. For example, consider the task of transfering funds from one bank account to another. Records from both accounts must be retrieved and changed. This task requires a series of queries to be completed. Therefore, explicit commands such as *start*, *commit*, *rollback*, and *undo*, should be available to users to achieve more complicated tasks.

This is not provided in the current version of REQUIEM.

7.1.3 Improvement of Lock Manager

Section 6.3.2 proposed a data structure focused around storing locks in virtual storage. The advantage of using that data structure is reduction in space required to store locks. Compared with other RDBMSs, more locks can be provided.

By improving the performance of locking operations, the overall performance of the RTM could also be improved.

7.1.4 Distributed Database Systems

The term *distributing processing* means that multiple machines can be connected together by means of a common communications network such that a single transaction can span several machines in the network [Date 90].

Due to increasing commercial demand, transaction processing over networks is rapidly becoming a necessity. A system with this capability is referred to as *a distributed database system.* To be more specific, a distributed database system implies that "a single application should be able to operate transparently on data that is spread across a variety of different databases, managed by a variety of different DBMSs, running on a variety of different machines, supported by a variety of different operating systems, and connected together by a variety of different communi-

cation networks - where the term *transparently* means that the application operates from a logical point of view as if the data were all managed by a single DBMS on a single machine" [Date 90].

Transaction management of distributed database systems is a serious topic in its own right. Although the design of the current version of RTM is of centralized nature, the expansion of the current design to a distributed database system is quite feasible. The mechanisms used in the RTM, such as locking, may be expanded and be used to develop a full-fledged distributed database system.

Appendix A

Building a Kernel for RTM

To increase the shared memory and semaphore facility on a given CPU one must build a new kernel for it which supports these facilities. To do this, the kernel builder must:

1. "cd /usr/sys/sun4c/conf"
2. "cp GENERIC NEW_KERNEL_NAME"
3. Edit NEW_KERNEL_NAME and after the lines

```
#
# The following options are for various System V IPC facilities.
# Most standard software does not need them, although they are
# used by SunGKS and some third-party software.
#
options IPCMESSAGE          # System V IPC message facility
options IPCSEMAPHORE        # System V IPC semaphore facility
options IPCSHMEM            # System V IPC shared-memory facility
```

add the line:

```
options SEMMNI=60           # Maximum of 60 semaphores
```

4. "config NEW_KERNEL_NAME"
5. "cd ../NEW_KERNEL_NAME"
6. "make"
7. Install the "./vmunix" file that results from the make, and reboot the machine.

On the Sun-4 machine, the default number of semaphores is 10 and the default number of shared memory segments is 100 according to the *Advanced Administration*

Topics, Section 22.5 *Tuning IPC System Parameters*, p700-704 in *Sun System & Network Administration Guide* [Sun 90].

Appendix B

A Sample Session

In this appendix we will describe what steps are necessary to install multi-user REQUIEM. For this purpose we have recorded some sample sessions that illustrate how the multi-user REQUIEM is compiled and linked and how the system is initialized. In addition, some information necessary for installing a multi-user REQUIEM DBMS is introduced.

B.1 Environment Variables

Each process on a Sun Operating System (SunOS) is created with an "environment" consisting of a set of named variables. The process may access these variables and obtain their values during its execution.

REQUIEM_DIR, DBMSKEY and LKMGKEY are the key environment variables used by the multi-user REQUIEM. They uniquely identify each central process (including the transaction manager and lock manager processes) and the REQUIEM root directory, respectively.

To see the value of a variable, use the **echo** command. For example:

```
% echo $REQUIEM_DIR
```

To set the value of a variable in the C shell, use the syntax:

setenv *VARIABLE* "*value*"

For example:

```
% setenv REQUIEM_DIR "/tmp/requiem"
```

The variable REQUIEM_DIR represents the full path name of the REQUIEM system home directory. This home directory is used to locate REQUIEM system files, among them:

- the REQUIEM catalogue file (*$REQUIEM_DIR/admin*): This directory also includes some other auxiliary files for system administrator.
- the REQUIEM executable files (*$REQUIEM_DIR/bin*): One file named RTM is created beforehand for running all background processes.
- the sample files (*$REQUIEM_DIR/doc*): This directory contains some sample files which illustrate the running of the REQUIEM DBMS.
- the database files (*$REQUIEM_DIR/database*): All data and index files are created in this directory.
- the REQUIEM source files (*$REQUIEM_DIR/src*).

B.2 The REQUIEM Makefile

We illustrate the REQUIEM component files in various directories, which must be assembled into different running packages by means of the **make** facility. To **make** the multi-user REQUIEM DBMS, simply change to the `src` directory under *$REQUIEM_DIR* directory and type **make**.

In the following, the working directory is $REQUIEM_DIR. It shows how the system executable files can be produced. The file names with a / are directories.

```
%ls
total 5
   1 admin/        1 database/     1 src/
   1 bin/          1 doc/
%ls bin
total 1
   1 RTM*
%cd src
%ls
```

```
total 7
   1 Makefile*          1 requiem_src/          1 user_process/
   1 lock_manager/      1 rtm/
   1 monitor/           1 terminator/
%make
(cd user_process; make)
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/access.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/btree.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/cat.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/create.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/err.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/hash.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/iex.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/impl.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/ipc.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/io.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/io_lib.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/lexan.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/misc.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/parser.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/pgm_int.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/pred.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/quedcmp.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/tabl.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/tmm.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/view.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/xcom.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/xinterp.c
ar rv requ.a access.o  btree.o  cat.o  create.o  err.o  hash.o  iex.o  impl.o
ipc.o  io.o  io_lib.o  lexan.o  misc.o  parser.o  pgm_int.o  pred.o  quedcmp.o
tabl.o  tmm.o  view.o  xcom.o  xinterp.o
r - access.o
r - btree.o
r - cat.o
r - create.o
r - err.o
r - hash.o
r - iex.o
r - impl.o
r - ipc.o
r - io.o
r - io_lib.o
r - lexan.o
r - misc.o
r - parser.o
r - pgm_int.o
r - pred.o
r - quedcmp.o
r - tabl.o
r - tmm.o
r - view.o
r - xcom.o
r - xinterp.o
ranlib requ.a
cc -g -DUNIX -DO_BINARY=0 -c requ.c
```

```
cc -g -DUNIX -DO_BINARY=0 -o ../../bin/requiem requ.o requ.a
(cd lock_manager; make)
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/access.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/btree.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/cat.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/create.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/err.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/hash.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/iex.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/impl.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/ipc.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/io.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/io_lib.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/lexan.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/misc.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/parser.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/pgm_int.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/pred.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/quedcmp.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/tabl.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/tmm.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/view.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/xcom.c
cc -g -DUNIX -DO_BINARY=0 -I../lock_manager -c ../requiem_src/xinterp.c
cc -g -DUNIX -DO_BINARY=0 -I../requiem_src -c lkmg.c
cc -g -DUNIX -DO_BINARY=0 -I../requiem_src -c lk.c
cc -g -DUNIX -DO_BINARY=0 -o ../../bin/lockmg lkmg.o lk.o access.o  btree.o
cat.o  create.o  err.o  hash.o  iex.o  impl.o  ipc.o  io.o  io_lib.o  lexan.o
misc.o  parser.o  pgm_int.o  pred.o  quedcmp.o  tabl.o  tmm.o  view.o  xcom.o
xinterp.o
(cd rtm; make)
cc -g -DUNIX -DO_BINARY=0 -I../user_process -I../requiem_src -c dbms.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -I../requiem_src -c Tm_io.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -I../requiem_src -c ../requiem_src/ipc.c
cc -g -DUNIX -DO_BINARY=0 -o ../../bin/rtm dbms.o  Tm_io.o  ipc.o
(cd terminator; make)
cc -g -DUNIX -DO_BINARY=0 -I../user_process -I../requiem_src -c termin.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/ipc.c
cc -g -DUNIX -DO_BINARY=0 -o ../../bin/terminator termin.o  ipc.o
(cd monitor; make)
cc -g -DUNIX -DO_BINARY=0 -I../user_process -I../requiem_src -c monitor.c
cc -g -DUNIX -DO_BINARY=0 -I../user_process -c ../requiem_src/ipc.c
cc -g -DUNIX -DO_BINARY=0 -o ../../bin/monitor monitor.o  ipc.o
rm */*.o
%
```

B.3 Database Schema Definitions

Here is the contents of file **cre_cat** which is used to create and initialize the multi-user REQUIEM database system:

```
create systab (relname  (char(12), UNIQUE),
               creator (char(20), SECONDARY),
               perms (num(5)),
               deriv (char(5)),
               natrs  (num(8)));

create sysatrs (atrname (char(20), UNIQUE),
               relname  (char(12), SECONDARY),
               type (char(5)),
               length (num(6)),
               index  (char(7)));

create sysview (viewname  (char(12), UNIQUE),
               base (char(12)),
               text  (char(132)));

create sysrllk (rlname (char(21), SECONDARY),
               transid (num(10)),
               mode (char(2)));

create syshdlk (hdname  (char(21), UNIQUE),
               transid (num(10)));

create sysixlk (ixname (char(21), UNIQUE),
               transid (num(10)));

create sysdtlk (dtname (char(21), SECONDARY),
               transid (num(10)),
               mode (char(2)));

create syskey (fgnkey  (char(12), UNIQUE),
               relname (char(12)),
               qualifier  (char(12)),
               primary  (char(12)));
```

B.4 Start A Multi-User REQUIEM

The file **cre_cat** is stored in *$REQUIEM_DIR/admin* directory and the system administrator should change the working directory to this directory to start up the REQUIEM. In addition, the background processes can be invoked from any directory.

Note: the three environment variables mentioned above should be setup before any of the multi-user REQUIEM processes starts. Here is the demonstration of the system start.

```
%ls
```

```
total 5
  1 admin/        1 database/      1 src/
  1 bin/          1 doc/
%ls bin
total 1073
  1 RTM*                40 monitor*                88 rtm*
456 lockmg*            448 requiem*                40 terminator*
%cd admin
%ls
total 19
  1 build_person        1 customer.dat             2 person.dat
  1 build_schema        2 female.dat               1 product.dat
  1 cre_cat             1 imports                  1 supplier.dat
  1 cre_per             1 invoice.dat              1 supply.dat
  1 cre_reminder        2 male.dat
  1 cre_schema          1 marriage.dat

%setenv REQUIEM_DIR "/tmp/requiem"
%setenv DBMSKEY 10000
%setenv LKMGKEY 10500

%RTM
************************************************************
REQUIEM Transaction Manager is now running in the background!
************************************************************
  REQUIEM Lock Manager is now running in the background!
************************************************************
 REQUIEM Monitor Manager is now running in the background!
************************************************************

************************************************************
You may start REQUIEM user process now! (Simply type requiem.)
************************************************************
%requiem

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+  REQUIEM: Relational Query & Update Interactive System   +
+  VERSION 2.2                                             +
////////////////////////////////////////////////////////////


+++++++++++++++++++++++++++++++++
+   Relation systab created     +
+                               +
/////////////////////////////////


+++++++++++++++++++++++++++++++++
+   Relation sysatrs created    +
+                               +
/////////////////////////////////
```

```
++++++++++++++++++++++++++++++++
+   Relation sysview created   +
+                              +
////////////////////////////////


++++++++++++++++++++++++++++++++
+   Relation sysrllk created   +
+                              +
////////////////////////////////


++++++++++++++++++++++++++++++++
+   Relation syshdlk created   +
+                              +
////////////////////////////////


++++++++++++++++++++++++++++++++
+   Relation sysixlk created   +
+                              +
////////////////////////////////


++++++++++++++++++++++++++++++++
+   Relation sysdtlk created   +
+                              +
////////////////////////////////


++++++++++++++++++++++++++++++++
+   Relation syskey created    +
+                              +
////////////////////////////////

REQUIEM: quit

 ***   You have left the database    ***

%
```

B.5 Single-User REQUIEM Compilation

To make the single-user REQUIEM, change directory to *$REQUIEM_DIR/src* and type `make single_requiem`. An executable file for the single-user REQUIEM

named *single_requiem* will then be created in *$REQUIEM_DIR/bin*. Detailed information about REQUIEM can be found in [Papazoglou and Valder 89].

Appendix C

Terminating Processes of The Multi-User REQUIEM

The following demonstrates how the central processes are terminated by means of **terminator**. To delete the shared memory and semaphores created by a user process, the process number of the process has to be known and input.

```
%terminator

Command (? for help)??
p remove semaphores and shared memory
q terminate the transaction manager
e terminate the lock manager
s exit

Command (? for help)?p

Input Transaction Process Number:?14493
OK

Command (? for help)?q
OK

Command (? for help)?e
OK

Command (? for help)?s
OK
%monitor terminated
```

Appendix D

Interprocess Communication Commands on UNIX System V

Here we illustrate how shared memory and semaphores created in the kernel can be deleted by means of UNIX commands.

```
%ipcs
IPC status from barnard as of Wed Mar 11 16:31:32 1992
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x00002710 --rw-rw-rw-      lan     staff
m    101 0x00002904 --rw-rw-rw-      lan     staff
Semaphores:
s     60 0x00002710 --ra-ra-ra-      lan     staff
s     61 0x00004e20 --ra-ra-ra-      lan     staff
s     62 0x00002904 --ra-ra-ra-      lan     staff
s     63 0x00005014 --ra-ra-ra-
%ipcrm -m 100 -m 101 -s 60 -s 61 -s 62 -s 63
%ipcs
IPC status from barnard as of Wed Mar 11 16:35:38 1992
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
%
```

For more information on SunOS shared memory operations, refer to **ipcs**(1), **ipcrm**(1), **shmget**(2), **shmctl**(2), **shmop**(2), in the *Commands Reference Manual.*

Appendix E

IPC code

```
/* the bridge between DBMS and it's clients using sephormore
   and shared memory */

#include "requ.h"

#define MSGSIZE sizeof(MESSAGE)+1                /* size of message to be sent */
#define DIFF 10000
#define MAXOPEN 20                               /* limit no. of processes */
#define BADMEM (char *)(-1)                      /* failure from shmat */

/* FAIL and MALLOC(x) have been defined in the header file requ.h
  #define FAIL -1
  #define MALLOC(x)  ((x *)malloc(sizeof(x)))  / allocate memory /
*/

/* table of created semaphores and shared memory lists */
static struct ipctable {
        int key;
        int sem1;
        int sem2;
        char *shmem;
        int shmid;
        struct ipctable *next;
}; /* ipctable */

static struct ipctable *ipclist = NULL;  /* the storage table */
extern int errno;  /* error message */

/* create the ipc of the key in the UNIX kernal */
static
int create_ipc(key, sem1, sem2, shmid, shmem)
int key;
int *sem1, *sem2, *shmid;
char **shmem;
{
        static struct ipctable *ipc1, *ipc2 = NULL;
        char *shmat();
```

```
        int counter = 0;

        /* check if ipc been created */
        for (ipc1 = ipclist; ipc1 != NULL && counter < MAXOPEN;
                    ipc1 = ipc1->next) {
                if (ipc1->key == key) {
                        *sem1 = ipc1->sem1;
                        *sem2 = ipc1->sem2;
                        *shmid = ipc1->shmid;
                        *shmem = ipc1->shmem;
                        return(TRUE);
                }
                ipc2 = ipc1;
                counter++;
        }
        /* if over MAXOPEN clients, return FAIL */
        if (counter == MAXOPEN) {
                errno = 0;
                return(FAIL);
        }
        /* if not created, create */
        *sem1 = semtran(key);
        *sem2 = semtran(key+DIFF);
        if ((*shmid = shmget((key_t)key, MSGSIZE, 0666 | IPC_CREAT)) == -1)
                return(FALSE);
        if ((*shmem = shmat(*shmid, 0, 0)) == BADMEM)
                return(FALSE);
        /* put ipc in table list */
        /* if table was empty */
        if (ipc2 == NULL) {
                if ((ipclist = MALLOC(struct ipctable)) == NULL)
                        return(FAIL);
                ipclist->key = key;
                ipclist->sem1 = *sem1;
                ipclist->sem2 = *sem2;
                ipclist->shmid = *shmid;
                ipclist->shmem = *shmem;
                ipclist->next = NULL;
        } else {
                if ((ipc2->next = MALLOC(struct ipctable)) == NULL)
                        return(FAIL);
                ipc2->next->key = key;
                ipc2->next->sem1 = *sem1;
                ipc2->next->sem2 = *sem2;
                ipc2->next->shmid = *shmid;
                ipc2->next->shmem = *shmem;
                ipc2->next = NULL;
        }
        return(TRUE);
} /* create_ipc */

/* send the message to the shared memory */
int send(dstkey, buf, nbytes)
int dstkey;
char *buf;
```

```
int nbytes;
{
        int sem1, sem2, shmid;
        char *shmem;

        if (!create_ipc(dstkey, &sem1, &sem2, &shmid, &shmem))
                syserr("create_ipc");
        p(sem1);
        memcpy(shmem, buf, nbytes);
        v(sem2);
        return(TRUE);
} /* send */

/* receive the message from the shared memory */
int receive(srckey, buf, nbytes)
int srckey;
char *buf;
int nbytes;
{
        int sem1, sem2, shmid;
        char *shmem;

        if (!create_ipc(srckey, &sem1, &sem2, &shmid, &shmem))
                syserr("create_ipc");
        v(sem1);
        p(sem2);
        memcpy(buf, shmem, nbytes);
        return(TRUE);
} /* receive */

/* Remove the ipc of the key in the kernal */
void rmqueue(key)
int key;
{
        int sem1, sem2, shmid;
        char *shmem;

        if (!create_ipc(key, &sem1, &sem2, &shmid, &shmem))
                syserr("create_ipc");
        (void)semctl(sem1, 0, IPC_RMID, 0);
        (void)semctl(sem2, 0, IPC_RMID, 0);
        (void)shmdt(shmem);
        (void)shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);
} /* rmqueue */

/* this is the function to delete quitted process's ipc from *
 * ipc table. This is used in dbms.c file. (Not included in ass2) */
int rmclient(key)
int key;
{
        static struct ipctable *ipc1, *ipc2 = NULL;

        for (ipc1 = ipclist; ipc1 != NULL ; ipc1 = ipc1->next) {
                if (ipc1->key == key) {
                        if (ipc2 == NULL) {
```

```
                        ipclist = ipclist->next;
                        free((char *) ipc1);
                } else {
                        ipc2->next = ipc1->next;
                        free((char *) ipc1);
                }
                return(TRUE);
            }
            ipc2 = ipc1;
        }
        return(FALSE);
} /* rmclient */

/* print system call error message and terminate */
void syserr(msg)
char *msg;
{
        extern int errno, sys_nerr;
        extern char *sys_errlist[];

        fprintf(stderr, "ERROR: %s (%d", msg, errno);
        if (errno > 0 && errno < sys_nerr)
                fprintf(stderr, "; %s)\n", sys_errlist[errno]);
        else
                fprintf(stderr, ")\n");
} /* syserr */

/* get the semaphore */
int semtran(key)
int key;
{
        int sid;

        if ((sid = semget((key_t)key, 1, 0666 | IPC_CREAT)) == -1)
            syserr("semget");
        return(sid);
} /* semtran */

/* perform the semaphore operation */
static void semcall(sid, op)
int sid;
int op;
{
        struct sembuf sb;

        sb.sem_num = 0;
        sb.sem_op = op;
        sb.sem_flg = 0;
        if (semop(sid, &sb, 1) == -1)
                syserr("semop");
} /* semcall */

/* acquire the semaphore */
p(sid)
int sid;
```

```
{
        semcall(sid, -1);
} /* p */

/* release the semaphore */
v(sid)
int sid;
{
        semcall(sid, 1);
} /* v */

/* print the error message */
void fatal(msg)
char *msg;
{
        fprintf(stderr, "ERROR : %s\n", msg);
        exit(1);
} /* fatal */

/* ----------------------- END of IPC code -------------------------- */
```

Appendix F

Data Structures of the RTM

```
#include "requ.h"
#include "btree.h"
#include "tmm.h"

/* Cache data structure */

struct ix_block {                    /* index file cache */
   long rloc;                        /* location of the block */
   char *cblock;                     /* block contents */
   int bsize;
   struct ix_block *next_block;      /* next block cache */
};

struct tuple_cache {
   int length_of_record;             /* the length of the record */
   char *old_record;                 /* old record buffer */
   char *new_record;                 /* new record buffer */
   long record_pos;                  /* the position number in file */
   struct tuple_cache *next_tuple;/* next record changed */
};

struct new_pages {                   /* contents of uncommitted data file */
   int pg_no;                        /* page number */
   char pg_dt[PAGE_SIZE];            /* page contents */
   struct tuple_cache *tc;           /* contents of changed tuples */
   struct new_pages *next_np;        /* next page of the file */
};

struct new_rel {
   struct ix_block *new_ix;          /* index file cache */
   struct header new_df_header;      /* new data file header */
   BOOL no_header;
   struct new_pages *np;             /* uncommitted version of data file */
   long ts_name;                     /* modified by which transaction */
   struct new_rel *next_rel;         /* more new version of data file */
};
```

```
struct pages {
   int pg_no;                      /* page number */
   char pg_dt[PAGE_SIZE];          /* page contents */
   BOOL status;                    /* if the data been changed */
   struct pages *next_pg;          /* next page in cache */
};

struct df_cache {                  /* committed version of data file cache */
   struct ix_block *ix_cache;      /* index file cache */
   BOOL block_updated;             /* if the index cache been updated */
   struct header df_header;        /* header of data file in cache */
   BOOL no_header;
   BOOL header_updated;            /* if header been updated */
   struct pages *df_page;          /* the data in cache */
   BOOL committed;                 /* if this cache been update committed */
   struct TM_list *tl;             /* transaction list in cache */
   struct df_cache *next_df;       /* next data file cache */
};

struct TM_list {
   long ts_name;                   /* transaction id */
   struct TM_list *next_tm;        /* next transaction in the list */
};

struct TM_cache {                  /* transaction cache */
   char rl_name[RNSIZE+1];         /* relation name */
   int df_fd;                      /* file descriptor of the data file */
   int ix_fd;                      /* file descriptor of the indexed file */
   int cache_ref;                  /* number of the relation referenced */
   int ix_ref;                     /* number of the relation referenced */
   struct df_cache *dtf_cache;     /* data file in cache */
   struct new_rel *nr;             /* new version of relations */
   struct TM_cache *next_cache;    /* next cache relation */
};

struct rel_names {
   char rl_name[RNSIZE+1];         /* relation name */
   struct rel_names *next_rels;    /* next relation name */
};

struct rel_list {
   long ts_name;                   /* transaction id */
   struct rel_names *rels;         /* relations list */
   struct rel_list *next_ts;       /* next relations list */
};
```

Appendix G

Proposed Data Structures for the Lock Manager

```
#define WORDSIZE 32

/* WORDSIZE */

#define SETWD(pos) ((pos)>>5)
#define SETBT(pos) ((pos)&037)

#define TIMESWORDSIZE(w) ((w)<<5)

#define ADDELEM1(setadd,pos)  (*(setadd) |= bit[pos])
#define DELELEM1(setadd,pos)  (*(setadd) &= ~bit[pos])
#define ISELEM1(setadd,pos)   ((*(setadd) & bit[pos]) != 0)

#define ADDELEMENT(setadd,pos) ((setadd)[SETWD(pos)] |= bit[SETBT(pos)])
#define DELELEMENT(setadd,pos) ((setadd)[SETWD(pos)] &= ~bit[SETBT(pos)])
#define ISELEMENT(setadd,pos)  (((setadd)[SETWD(pos)] & bit[SETBT(pos)]) != 0)

#define EMPTYSET(setadd,m) \
    {register setword *es; \
    for (es = (setword*)(setadd)+(m); --es >= (setword*)(setadd);) *es=0;}

#define MAKEEMPTY(setadd,m,i) for(i=m;--i>=0;)setadd[i]=0

#define NOTSUBSET(word1,word2) ((word1) & ~(word2))  /* test if the 1-bits
                    in setword word1 do not form a subset of those in word2  */
#define INTERSECT(word1,word2) ((word1) &= (word2))  /* AND word2 into word1 */
#define UNION(word1,word2)     ((word1) |= (word2))  /* OR word2 into word1 */
#define SETDIFF(word1,word2)   ((word1) &= ~(word2)) /* - word2 into word1 */
#define XOR(word1,word2)       ((word1) ^= (word2))  /* XOR word2 into word1 */
#define ZAPBIT(word,x) ((word) &= ~bit[x])  /* delete bit x in setword */

#define POPCOUNT(x) (bytecount[(x)>>24 & 0377] + bytecount[(x)>>16 & 0377] \
                    + bytecount[(x)>>8 & 0377] + bytecount[(x) & 0377])
#define FIRSTBIT(x) ((x) & 037777600000 ? ((x) & 037700000000 ? \
```

```
                    leftbit[((x)>>24) & 0377] : 8+leftbit[(x)>>16]) \
                   : ((x) & 0177400 ? 16+leftbit[(x)>>8] : 24+leftbit[x]))
#define BITMASK(x)  (017777777777 >> (x)) /* setword whose rightmost
  WORDSIZE-x-1 (numbered) bits are 1 and the rest 0 (0 <= x < WORDSIZE) */


typedef unsigned long setword;


typedef setword set;

#define INFINITY 077777          /* positive short int greater than MAXN+2 */

extern setword bit[];
extern int bytecount[];
extern int leftbit[];
```

References

[Agrawal and Dewitt 85]

R.Agrawal and D.J.Dewitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", *ACM Transactions on Database Systems*, Vol. 10, No. 4, June 1985, pp.529-564.

[Agrawal 87]

R.Agrawal, M.J.Carey and M.Livny, "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems*, Vol. 12, No. 4, June 1987, pp.609-654.

[Bernstein 79]

P.A.Bernstein, D.W.Shipman and W.S.Wong, "Formal Aspects of Serializability in Database Concurrency Control", *IEEE Transaction on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp.203-216.

[Bernstein 83]

P.A.Bernstein, N.Goodman and V.Hadzilacos, "Recover Algorithms for Database Systems", *Proceedings IFIP 9th World Computer Congress*, North-Holland, Paris, September, 1983, pp. 799-807.

[Bernstein and Goodman 83]

P.A.Bernstein and N.Goodman, "Multiversion Concurrency Control — Theory and Algorithms", *ACM Transactions on Database Systems*, Vol. 8, No. 4, December 1983, pp. 465-483.

[Bernstein 89]

P.A.Bernstein, V.Hadzilacos and N.Goodman, *Concurrency Control and Recovery in Database Systems,* Addison Wesley, 1989.

[Blasgen *et al* 81]

M.W.Blasgen, M.M.Astrahan, D.D.Chamberlin, J.G.Gray, W.F.King, B.G.Lindsay, R.A.Lorie, J.W.Mehl, T.G.Price, G.R.Putzolu, M.Schkolnick, P.G.Selinger, D.R.Slutz, H.R.Strong, I.L.Traiger, B.W.Wade, and R.A.Yost, "System R: An Architecture overview", *IBM Systems Journal*, Vol. 20, No. 1, 1981, pp. 41-62.

[Cheng *et al* 84]

J.M.Cheng, C.R.Loosley, A.Shibamiya, and P.S.Worthington, "IBM Database 2 Performance: Design, Implementation, and Tuning", *IBM Systems Journal*, Vol. 23, No. 2, 1984, pp. 189-210.

[Coffman 71]

E.G.Coffman, M.J.Elphick and A.Shoshani, "System Deadlocks", *Computing Surveys*, Vol. 3, No. 2, June 1971, pp.67-78.

[Crus 84]

R.A.Crus, "Data Recovery in IBM Database2", *IBM Systems Journal*, Vol. 23, No. 2, 1984, pp. 178-188.

[Date 83]

C.J.Date, *An Introduction to Database Systems, Volume II,* Addison Wesley, 1983.

[Date 90]

C.J.Date, *An Introduction to Database Systems, Volume I, Fifth Edition,* Addison Wesley, 1990.

[Elhardt and Bayer 84]

K.Elhardt and R.Bayer, "A Database Cache for High Performance and Fast

Restart in Database Systems", *ACM Transactions on Database Systems*, Vol. 9, No. 4, June 1984, pp.503-525.

[Eswaran *et al* 76]

K.P.Eswaran, J.N.Gray, R.A.Lorie and I.L.Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Vol. 19, No. 11, Nov. 1976, pp.624-634.

[Gray 75]

J.N.Gray, R.A.Lorie and G.F.Putzolu, "Granularity of Locks in a Large Shared Data Base", *Proc. 1st International Conference on Very Large Data Bases*, Sep. 1975.

[Gray *et al* 76]

J.N.Gray, R.A.Lorie, G.F.Putzolu and I.L.Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", *Modeling in Data Base Management Systems*, G.M.Nijssen editor, North Holland, 1976, pp.365-394.

[Gray 78]

J.N.Gray, "Notes on Data Base Operating Systems", *Lecture notes in Computer Science: Operating Systems*, Springer-Verlag, Berlin, Vol. 60, 1978, pp.393-481.

[Gray 80]

J.N.Gray, "A Transaction Model", *Lecture notes in Computer Science*, Springer-Verlag, Berlin, Vol. 85, 1980, pp.282-298.

[Gray *et al* 81]

J.N.Gray, P.McJones, M.W.Blasgen, B.G.Lindsay, R.A.Lorie, T.G.Price, G.R.Putzolu and I.L.Traiger, "The Recovery Manager of the System R Database Manager", *Computing Surveys*, Vol. 13(2), 1981, pp.223-242.

[Haerder and Reuter 83]

T.Haerder and A.Reuter, "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, Vol. 13(4), Dec. 1983, pp.287-317.

[Haerder and Rothermel 87]

T.Haerder and K.Rothermel, "Concepts for Transaction Recovery in Nested Transactions", *Proceedings of ACM SIGMOD 1987 Anunal Conference*, San Francisco, Vol. 16, No. 3, Dec. 1987.

[Härder and Petry 87]

T.Härder and E.Petry, "Evaluation of a Multiple Version Scheme for Concurrency Control", *Information Systems*, Vol. 12, No. 1, 1987, pp.83-98.

[Harbison and Steele 87]

S.P.Harbison and G.L.Steele Jr., *C: A Reference Manual, Second Edition,* Prentice Hall, 1987.

[Herlihy 86]

M.Herlihy, "Optimistic Concurrency Control for Abstract Data Types", *Proceedings 2nd International Conference on Data Engineering*, 1986, pp.366-386.

[Holt 72]

R.C.Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4, No. 3, Sep. 1972, pp.179-196.

[INGRES 90]

INGRES Database Adiminstrator's Guide for the UNIX Operating System, Release 6.3, November 1990.

[Joergensen and Singhal 87]

G.Joergensen and M.Singhal, "A Comparative Analysis of Two-Phase and Non-Two-Phase Locking Algorithms for Database Systems", *IEEE*, 1987, 568-574.

[Kernighan and Ritchie 88]

B.W.Kernighan and D.M.Ritchie, *The C Programming Language, Second Edition,* Prentice Hall, 1988.

[Kersten and Tebra 84]

M.L.Kersten and H.Tebra, "Application of an Optimistic Concurrency Control Method", *Software Practice and Experience*, Vol 14(2), Feb. 1984, pp.153-168.

[Kim and Lochovsky 89]

W.Kim and F.H.Lochovsky, *Objected-Oriented Concepts, Databases, and Applications,* ACM Press, 1989.

[Korth 82]

H.F.Korth, "Deadlock Freedom Using Edge Locks", *ACM Transaction on Database Systems*, Vol. 7, No. 4, Dec. 1982, pp.632-652.

[Korth 83]

H.F.Korth, "Locking Primitives in a Database System", *Journal of the ACM*, Vol. 30, No. 1, Jan. 1983, pp.55-79.

[Korth and Silberschatz 86]

H.F.Korth and A.Silberschatz, *Database System Concepts,* McGraw-Hill, 1986.

[Kung and Robinson 81]

H.T.Kung and J.T.Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6(2), Jun. 1981, pp.213-226.

[Lausen 82]

G.Lausen, "Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking", *Proceedings of the ACM'82 Conference*, 1982, pp.64-68.

[Menascé and Nakanishi 82]

D.A.Menascé and T.Nakanishi, "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems", *Information Systems*, Vol. 7, No. 1, 1982, pp. 13-27.

[Oracle 88]

ORACLE RDBMS Database Administrator's Guide, Version 6.0, ORACLE Corporation, November 1988.

[Orji 88]

C.U.Orji, L.Lilien and J.Hyziak, "A Performance Analysis of an Optimistic and a Basic Timestamp-Ordering Concurrency Control Algorithms for Centralized Database Systems", *IEEE*, Feb. 1988, pp.64-71.

[Papazoglou 89]

M.P.Papazoglou, "An Extensible DBMS for Small-Medium Scale Systems", *IEEE Micro*, April 1989.

[Papazoglou and Valder 89]

M.P.Papazoglou and W.Valder, *Relational Database Management A System Programming Approach,* Prentice Hall, 1989.

[Papadimitriou 79]

C.H.Papadimitriou, "The Serializability of Concurrency Control Database Updates", *Journal of the ACM*, Vol. 26, No. 4, October 1979, pp. 631-653.

[Potier and Leblanc 80]

D.Potier and Ph.Leblanc, "Analysis of Locking Policies in Database Management Systems", *Communications of the ACM*, Vol. 23, No. 10, October 1980, pp. 584-593.

[Power 91]

R.Power, *The Development of a Program Interface for the Relational Database Management System*, Honours Thesis, Australian National University, 1991.

[Reuter 84]

A.Reuter, "Performance Analysis of Recovery Techniques", *ACM Transactions on Database Systems*, Vol. 9, No. 4, June 1984, pp.526-559.

[Ries and Stonebraker 79]

D.R.Ries and M.R.Stonebraker, "Locking Granularity Revisited", *ACM Transactions on Database Systems*, Vol. 4, No. 2, June 1979, pp.210-227.

[Ritchie and Thompson 74]

D.M.Ritchie and K.Thompson, "The Unix Time-Sharing System", *Communications of ACM*, Vol. 17, No. 7, July, 1974, pp. 365-375.

[Robinson 82]

J.T. Robinson, *Design of Concurrency Controls for transaction processing systems,* Ph.D. Thesis, Carnegie-Mellon University, 1982.

[Rochkind 85]

M.J.Rochkind, *Advanced Unix Programming,* Prentice-Hall Software Series, 1985.

[Stevens 90]

W.R.Stevens, *Unix Network Programming,* Prentice-Hall Software Series, 1990.

[Stonebraker 86]

M.Stonebraker, *The INGRES Papers: Anatomy of a Relational Database System,* Addison-Wesley Publishing Co., Reading, Mass., 1986.

[Sun 90]

Sun System & Network Administration Guide, Sun Microsystems Inc., 1990.

[Tay 85]

Y.C.Tay, N.Goodman and R.Suri, "Locking Performance in Centralized Databases", *ACM Transactions on Database Systems*, Vol. 10, No. 4, June 1985, pp.415-462.

[Teng and Gumaer 84]

J.Z.Teng and R.A.Gumaer, "Managing IBM Database 2 buffers to maximize Performance", *IBM Systems Journal*, Vol. 23, No. 2, 1984, pp. 211.

[Valduriez and Gardarin 89]

P.Valduriez and G.Gardarin, *Analysis and Comparison of Relational Database Systems,* Addison Wesley, 1989.