USING SEMANTIC KNOWLEDGE FOR TRANSACTION PROCESSING

Ricardo A. Cordon

October, 1985

CS-TR-009-85

USING SEMANTIC KNOWLEDGE FOR TRANSACTION PROCESSING

Ricardo A. Cordon

A DISSERTATION
PRESENTED TO THE
FACULTY OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE BY THE
DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

OCTOBER 1985

Meiner lieben Grossmutter, fur den Ruckhalt den ich immer bei ihr gefunden habe.$˜ sup *$

$˜ sup *$To my dear grandmother, for the suppport that I was always able to find in her.

ACKNOWLEDGEMENTS

My almost five years at Princeton have been full of interesting experiences that have taught me many things about life. Some of those experiences were not always easy to go through, especially the ones related to health problems. Many persons gave me a helpful hand and numerous cheering up smiles during those times. I take the opportunity here to say THANK YOU to all of them, but specially to Teresita Heron, my friend from Ecuador, Mary Kay Heffern, whom I call "the only outstanding member of McCosh Infirmary", and my very good friends Daniel Barbara, Henry Fernandez, William Jones '84 and Jack Kent. Let me also express here my gratitude to Prof. Bede Liu and Gerry Pecht for always being nice and helpful during my years at Princeton.

For the many discussions on database management systems, and for their suggestions on this thesis, I want to thank all of the student members of **PDBG** (Princeton Data Base Group): Jae Chung '84, Frank Pitelli, and again, Jack Kent and Daniel Barbara. I also want to thank Prof. Bruce Arden and Prof. Susan Davidson (University of Pennsylvania) for taking their time to proofread this thesis, and to Mindy Klasky '86, for her useful English corrections and comments on several chapters.

It goes almost without saying that I very sincerely want to thank all those members of my family and friends in Guatemala, that in one way or another, with their loving support, encouraged me to pursue my academic goals. With my deepest appreciation these thanks go particularly to a very special women for her continuous care, love, and support: my mother.

My last words of thanks, but the most important ones, I have left for a person that during my years in Princeton was always able to offer me a helpful hand in academics and life. His advice, discussions, and suggestions on the topics of my thesis were tremendously valuable. His understanding of the different problems I went through, and his always smiling and friendly attitude were a relaxing response to my anxiety. I want to express here my most sincere gratitude to my advisor Prof. Hector Garcia-Molina. *A million thanks Hector!*

ABSTRACT

New methods of concurrency control that utilize the semantics of an application to improve performance have been proposed in recent years. In this thesis we study one of these mechanisms. After presenting the details of the mechanism, we compare it, via a simulation, to a conventional two phase locking strategy. The goal of such comparison is to determine the conditions under which the higher complexity and overhead of the application dependent mechanism pays off. Although the results presented are specific to the two selected mechanisms, we believe they provide insight into the operation of other application dependent mechanisms. The specifics of the semantic knowledge mechanism, and the algorithms used, are based on the original work that appears in [Garc]. We believe that those algorithms are not optimal, and we therefore also propose and study some modifications for improvement.

To take full advantage of the ideas in [Garc], it is necessary that we understand when two transactions are semantically compatible, i.e., when are they allowed to freely interleave their execution steps without violating data consistency. We also study in this thesis different aspects concerning the compatibility of transactions.

Finally, we present an application of our mechanism that will help to manage long lived transactions (LLT). This application shows that for the cases when a large percentage of the transactions in the system are compatible with the executing LLTs, the performance of the system improves greatly.

---

† This material is based upon work partially supported by the National Science Foundation under grants ECS-8303146 and ECS-8351616.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION AND OVERVIEW

A database (DB) is a collection of stored data, maintained for purposes of facilitating operations and decision making. If all the data is stored at only one computer we say that the DB is centralized. If the data is stored at several computing nodes, linked by a communication media, we say the that the DB is distributed. Among the requirements a DB must meet are:

1) Data accessibility to different users, even though the data may be stored far from where it is required.

2) Reliable storage of data for long periods of time.

3) Accessibility by concurrent users as long as its consistency is preserved. Consistency preservation means that the rules (consistency constraints) that apply to the stored data (e.g., $a + b = c$) have to be always observed.

4) Have data stored and organized so that queries and updates can be processed efficiently.

A database management system (DBMS) is an integrated collection of computer programs that provide the services required to manage a DB.

The means of interaction between users and a DB are transactions. A transaction is a collection of actions (read/write) acting over a portion of the data, transforming the DB from an old consistent state to a new consistent one. (Note that transactions that only read data will not transform the DB.) A transaction has to be atomic, i.e., either all of its actions are executed, and its changes, if any, reflected in the DB, or none of the effects of its actions are made permanent. If the changes are made we say that the transaction *commits*; or else it *aborts*.

While it would be easy to execute transactions one by one, i.e., without interference among them, this is not desirable. A serial execution does not permit transactions to exploit the advantages of parallel processing. Concurrent processing will permit greater system utilization. This will, in turn, result in a smaller average turnaround time for the transactions. To be able to maintain data consistency in spite of concurrent access to DB objects, a mechanism that regulates the access to objects has to exist. It will permit concurrent access, but will always allow users to have a consistent view of the DB. Such mechanism, which is part of the collection of programs of the DBMS, is called the concurrency control manager.

The first DBMSs, developed in the early sixties [Wied], did not have the notion of concurrency control as we know it now. They used a log (reserved space in disk) to write the changes made by a transaction to DB objects, in order to provide an historical record. Their view that concurrently executing transactions would not violate consistency was optimistic. However, if at the end of a transaction execution, one or more of these changes were found to have violated consistency, then the transaction had to be backed out (aborted) by installing in the DB the old values (for those objects) found in the log. Logs are still widely used today in crash recovery mechanisms [Gray3, Kent], but concurrency control is not left in the hands of the users.

With the advent of computerization in all areas of industry, business, education, and even households, in the past ten years, the demand for on-line databases has increased considerably, and with it the desire for efficient mechanisms that permit concurrent access to data. Using the log, as in 'old' times, hinders throughput if too many transactions have to be rolled back.

The most widely used general purpose concurrency control mechanisms today enforce the notion of *Serializability*. These mechanisms guarantee that the values of DB objects, after the execution of a schedule of transactions, are going to be equal to those produced by a schedule that executed each of the transactions sequentially, i.e, without interference from other transactions. Note that serializable schedules do permit interleavings of transactions, as long as their actions do not conflict [Eswar].

One common way to achieve serializable schedules is to use locking. It has been proved [Eswar] that three conditions are sufficient for a schedule of transactions to be serializable:

1) A transaction should never be allowed to access an object without first locking it;

2) A transaction should never be allowed to access an object that is already locked by another transaction; and

3) Once the first lock on an object is released, the transaction should never attempt to lock a new object.

Conditions (1) and (2) are standard among most actual locking concurrency control mechanisms, since without them, the consistency of the DB would be in constant danger while the system is active. A transaction that follows rule (3) is called two phased because there is a phase where it only asks for locks and then a phase where it only releases them. A locking mechanism that follows these rules is called two phased locking (2PL).

Two phased locking has been shown to produce consistent data, but it has also been shown that a schedule of transactions does not need to be 2PL to maintain DB consistency. In fact, it is well known that for specific applications,

consistency can be maintained with more flexible mechanisms, that allow higher concurrency and do not observe two phase locking or even serializability [Eswar, Kung]. These mechanisms either operate on data with a specific structure (e.g. a tree) [Silbe]; only allow a set of simple operations on the data (e.g., operations that maintain a directory or recovery log) [Stron, Lehma]; or may take as input semantic information that specifies how transactions can be interleaved [Clark, Fisch, Garc, Lynch].

The objective of this thesis is to study a concurrency control mechanism that does not observe 2PL, but that uses the semantic information of transactions to allow higher concurrency and still preserve consistency. The original ideas, and algorithms, for such mechanism, and on which the research presented in this thesis is based, are described in [Garc]. Under the rules of this mechanism, which we will call a Semantic Knowledge (SK) Mechanism, transactions are classified into semantic types. Each type will have an associated set, called the compatibility set. This set will list the types of transactions that are compatible with transactions of the type defining the set. Compatible transactions are transactions that due to their semantic construction, can interleave their steps without violating consistency.

This thesis will be organized as follows:

A brief review of the background ideas on serializable schedules and an introduction to the concept of Semantic Knowledge (SK) applied to the processing of concurrent executing transactions will be presented in chapter 2. Examples will be given.

Chapter 3 presents the performance results of an experiment comparing a SK concurrency control mechanism to a 2PL one. These results are taken from a

simulation that measured the behavior of transactions in a two site distributed database. Of special interest are the values obtained for two different system's performance predictors. Such values could be very useful to discriminate when a SK based concurrency control mechanism should be considered for implementation, instead of a 2PL one.

In chapter 4 we propose two variations of the original SK algorithms presented in [Garc] and study their performance. The first variation is an algorithm that more effectively processes transactions with an empty compatibility set. The second variation is based on restricting the class of compatible sets that the original algorithm processes.

Since it is very important to know when two transactions are compatible, in chapter 5 we decided to explore the conditions under which some classes of transactions are compatible, and suggest aids that permit two transactions to become compatible. Three different classes of transactions are studied here: 1) Transactions whose actions are linear assignments; 2) Transactions executing set operations; and 3) Transactions where the timestamp (submission time) of the transaction determines the value of an object to be written in the database.

In chapter 6 we adapt the ideas of SK to the processing of long lived transactions (LLT), transactions whose execution even without interference from other transactions take a substantial amount of time, possibly on the order of hours or days. The implementation of a DBMS that relies on a SK based concurrency control mechanism to process LLTs is described and discussed.

A summary of the thesis, together with some conclusions and problems for future research appears in chapter 7.

Chapter 2

SERIALIZABILITY AND SEMANTIC KNOWLEDGE

2.1. Introduction

Schedules of transactions need to obey certain rules in order to transform the DB from a consistent state to another consistent state. These rules are especially needed when transactions are allowed to interleave some of their execution steps in order to obtain good system performance. Different types of concurrency control mechanisms that achieve this have been used in the past [Bern]. As mentioned in the chapter 1, probably the most widely used general purpose (GP) concurrency control mechanism is two phase locking (2PL). This mechanism is too restrictive, and for application dependent DBMSs, it may be worth considering a less restrictive, more flexible, concurrency control mechanism. The purpose of this chapter is to present some background on serial and serializable schedules (section 2.2), and to introduce the reader to the concept of semantic knowledge (SK) and a SK based concurrency control mechanism, a mechanism that takes as input the semantic information of the transactions (section 2.3).

2.2. Serializability

We will start this section by stating the concepts of a serial and of a serializable schedule. We assume the reader is familiar with serializability, and present the most important definitions and results. For those not familiar with these material we refer them to [Eswar].

Definition 2.2.1 (Serial Schedule):

Let S be a schedule of $\mathbf{T}=\{T_1, \ldots, T_n \,/\, T_i \text{ is a transaction}\}$. S is serial iff its execution is equivalent to executing $T_{\mathcal{S}(1)}, \ldots, T_{\mathcal{S}(n)}$, where & is a permutation of $1....n$. In other words, a serial schedule executes all the actions of a transaction as a block, i.e., without interference of any other actions. ○

Definition 2.2.2 (Serializable Schedule):

Let S be a schedule of $\mathbf{T}=\{T_1, \ldots, T_n\}$. S is said to be serializable iff there exists a serial schedule S' of $\mathbf{T}$, such that S and S' produce the same results. We will say that S and S' are equivalent. ○

From our first definition we can conclude that a serial schedule maintains the consistency of the database. This is because transactions are executed without interference and because transactions transform a consistent database into a new consistent one. We then can directly conclude that a serializable schedule will also maintain DB (database) consistency. It is natural that we now ask ourselves what the sufficient conditions for serializability are. The next two theorems will give us this answer. For convenience we will precede them by the following definition.

Definition 2.2.3 (Dependency Graph):

Let S be a schedule of $\mathbf{T}=\{T_1, \ldots, T_n\}$, and $T_i, T_j \in \mathbf{T}$. Let o be an object of the DB. Suppose that T_i accesses object o and the next transaction to access o is T_j. If at least one of the accesses intends to write a new value for o we say that there exist a dependency from T_i to T_j. We will represent such dependency by $T_i \rightarrow T_j$. The dependency graph of S, written $G(S)$, is the graph constructed

from the union of all dependencies produced by S. The nodes of the graph are the transaction's identifiers and the edges the dependencies between the transactions. ○

Theorem 2.2.1:

Let S be schedule of $\mathbf{T}=\{T_1, \ldots, T_n\}$ and G(S) its dependency graph. If G(S) is acyclic then S is serializable. [Eswar]

Sketch of proof: Topologically sort G(S) assigning numbers to the nodes. This new enumeration of the nodes gives us a serial schedule of the transactions. This is equivalent to S since the dependencies between the transactions have been preserved. ○

Definition 2.2.4 (Well formed transaction):

A transaction is said to be *well formed* iff it locks a DB object before it accesses the object. ○

Definition 2.2.5 (Two phase transaction):

A transaction is said to be *two phase* iff it does not ask for a new lock once it has already released a lock. ○

Definition 2.2.6 (Legal schedule):

A schedule S of transactions is said to be *legal* iff every transaction T in S does not access any object locked by a different transaction. ○

Theorem 2.2.2: [Eswar]

Let S be a schedule of $\mathbf{T}=\{T_1, \ldots, T_n\}$. If $T_1, \ldots, T_n$ are well formed and two phase, and S is legal then G(S) is acyclic.

Note: We will omit the proof of this theorem and refer the reader to [Eswar]. ○

Having stated the previous definitions and theorems, we see that acyclicity of $G(S)$, is all we need for schedule S to maintain DB consistency. This is, in fact, a positive point for serializability, since it is not difficult to implement a DB schedule manager that enforces the conditions of theorem 2.2.2, and therefore forces the dependency graph of the schedule to be acyclic. On the other hand, we can see that acyclicity is too strong a sufficient condition for consistency (see example below). This leads to inflexibility in the scheduling of transactions, decreasing the degree of transaction's concurrency and therefore increasing response times.

Example 2.2.1:

Let A be an object of the DB. Let T_1 and T_2 be two transactions such that

$$T_1\colon\ a_{11}\colon A \leftarrow A + 200$$

$$a_{12}\colon A \leftarrow A * 2 \qquad \text{; and}$$

$$T_2\colon\ a_{21}\colon A \leftarrow A * 3$$

$$a_{22}\colon A \leftarrow A + 300$$

Consider now the following execution schedule, S:

$$(T_1)\ A \leftarrow A + 200$$

$$(T_2)\ A \leftarrow A * 3$$

$$(T_1)\ A \leftarrow A * 2$$

$$(T_2)\ A \leftarrow A + 300$$

S is a serializable schedule since the serial schedule $S' = T_1, T_2 = (a_{11}, a_{12}, a_{21}, a_{22})$ would have produced the same results, for whatever initial value of A. However, G(S) is not acyclic, since we have the dependencies $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$.

The previous example showed how a schedule S could be serializable, even though its dependency graph had a cycle. This was the case because the multiplication operation has the commutative property. Such kind of properties can lead us to the discovery of many types of transactions that allow for serializable schedules with cyclic dependency graphs. Exploring more, we will also be able to show that even serializability is not necessary to achieve schedules that maintain DB consistency. The next section will introduce the reader to a concurrency control mechanism that guarantees DB consistency, but where the schedules are not forced to be serializable.

2.3. Semantic Knowledge

In this section we briefly discuss a strategy for using semantic or application knowledge in transaction processing, and illustrate how it may increase parallelism. (We refer the interested reader to [Garc] for a detailed explanation of these ideas). This method, which we call SK, is based on the nature (semantics) of the different transactions that a given DBMS processes.

Under the SK strategy, transactions are divided into *steps*. Each step is a collection of conventional database operations (e.g., read or write a record) that will be performed as an atomic unit at a single node in a distributed system. If

two transactions are *compatible*, it is possible to interleave their steps in any fashion without violating database consistency. The users of the database define what transactions are compatible by classifying them into *semantic types*; and for each type, the compatible types are given.

For example, consider an application where there are three warehouses and each one keeps its inventory on a computer. Suppose there is a transaction T_1 that examines all inventories and produces a report for the management. This transaction does not need a consistent, up to date view of the data, as long as each local inventory it examines is valid. Then T_1 can be interleaved with other transactions that update the inventory. That is, users would define semantic types "Monthly Report" (with T_1 as an instance) and "Inventory Update," and for each would give the other type as its compatible type.

The parallelism induced by the compatible sets will be especially useful in a distributed DBMS. Recall that in distributed systems, it is common for a transaction, like T_1, to visit several nodes in order to complete its required processing. In such cases, the transaction may leave locks on objects at a node N until it completes. The locks could remain for a substantial time, due to communication delays. A second transaction, T_2, requiring the object locked by T_1 will have to wait a long time. However, if T_1 and T_2 are transactions of compatible types, then with a SK concurrency control mechanism, T_2 will be able to access the object as soon as T_1 finishes processing at N, without violating consistency.

To allow compatible transactions, like T_1 and T_2 to share objects, the scheduler utilizes two types of locks: (a) Local, to ensure atomicity of steps; and (b)Global, to allow compatible transactions to run concurrently, i.e., to allow valid interleavings. Local locks are released when a step finishes; global ones are

kept at least until the transaction finishes, possibly longer. To simplify matters we will not return to discuss global lock release until after the presentation of the next set of definitions and the examples that will follow them. This set of definitions will describe our SK notions more precisely.

Definition 2.3.1 (Step):

The actions of each transaction T are grouped (by the users) into a series of *steps* $s_1 \cdots s_n \, (n \geq 1)$. Each step will be atomic and executed at just one node.○

Definition 2.3.2 (Step-Wise Serial Schedule):

All schedules where the steps of transactions are executed as atomic units, will be called step-wise serial.○

Definition 2.3.3 (Semantic Type):

According to the nature (i.e., semantics) of the transactions users will classify them into *semantic types*. That is, each transaction, T, will have its own type $ty(T) \in TYPES$, where $TYPES$ is the set of all semantic types.○

Definition 2.3.4 (Compatibility Set):

Each semantic type $Y \in TYPES$ defines a *compatibility set*, $cs(Y)$, whose elements are interleaving descriptor sets as defined below.○

Definition 2.3.5 (Interleaving Descriptor set):

An *interleaving descriptor set* of a compatibility set $cs(Y)$, h, ($h \in cs(Y)$, $Y \in TYPES$), must have the following properties:

(i) $h \subseteq TYPES$

(ii) For all transactions T_1, T_2 such that $ty(T_1)$, $ty(T_2) \in h$, any interleaving of the steps of the two transactions will not violate database consistency.○

Example 2.3.1:

Let Y_1, Y_2, ... be semantic types and consider the following compatibility sets:

1) $cs(Y_1)=\{\{Y_1, Y_2, Y_3\}, \{Y_1, Y_7\}\}$

2) $cs(Y_8)=\{\{Y_9, Y_{10}\}\}$.

These sets illustrate two different aspects of compatibility. The first set states that transactions of type Y_1 can run concurrently with transactions of type Y_2 and Y_3, or they can run concurrently with those of type Y_7. However transactions of types Y_2 and Y_7 will not be allowed to interleave their steps since they do not belong to the same interleaving descriptor. The second set illustrates the fact that a transaction of a certain type may not be allowed to interleave its steps with the ones of other transactions of its same type. Note, however, that in practice we expect most transactions to be compatible with transactions of their same type.○

The following example will illustrate the use of compatibility sets in a simple application.

Example 2.3.2:

Suppose that we have a bank with three accounts: A, B and C. Accounts A and B will be customer accounts that are charged a 10 dollar penalty fee if they ever go below 1500 during a month. C will serve for the bank's internal accounting, and records the total penalties collected. The only restriction on accounts

will be that money be accounted for, i.e.,

$$\text{Bal(A)} + \text{Bal(B)} + \text{Bal(C)} = \text{Tot},$$

where Tot is a variable that records the total amount of money in the bank, and Bal(X) is the balance of account X. The remaining two variables, PENA_COLL(A) and PENA_COLL(B) are flags that insure that the penalty fee is collected just once a month. It will, in fact, be collected the first time the account goes below 1500. A special transaction, of type RST (see below), that resets both flags to their initial value ('False'), will be run at the bank's closing time on the last day of the month. Note that this has to be the last transaction, with access to the flags, submitted to the DBMS during the course of any month.

For this DBMS we will define the following three types of transactions:

a) D2(x)(deposit in accounts A and B the amount x). Its steps are:

   1) Bal(A) ← Bal(A) + x; Tot ← Tot + x

   2) Bal(B) ← Bal(B) + x; Tot ← Tot + x

b) W2(y)(withdraw from accounts A and B the amount y). Its steps are:

   1) Bal(A) ← Bal(A) - y; Tot ← Tot - y;

      If Bal(A) < 1500 and not PENA_COLL(A) then

        Bal(A) ← Bal(A) - 10;

        Bal(C) ← Bal(C) + 10;

        PENA_COLL(A) ← 'True'

      Endif

   2) Bal(B) ← Bal(B) - y; Tot ← Tot -y;

      If Bal(B) < 1500 and not PENA_COLL(B) then

        Bal(B) ← Bal(B) - 10;

Bal(C) ← Bal(C) + 10;

PENA_COLL(B) ← 'True'

Endif

c) RST. Its only step is:

PENA-COLL(A) ← 'False'; PENA_COLL(B) ← 'False'

Since the addition operation is commutative (i.e., x - y = -y + x), and account C will be credited 10 dollars every time the penalty is charged, then it is clear that interleavings of the steps of transactions of types D2 and W2 will not violate the restriction Bal(A) + Bal(B) + Bal(C) = Tot. Based on these facts we can define cs(W2)={{D2, W2}}. By these same facts and since transactions of type D2 do not access the variables PENA_COLL(A) and PENA_COLL(B), then the compatibility set of D2 can be defined as cs(D2)={{D2, W2}, {D2, RST}}. Running transactions of types W2 and RST concurrently could result in collecting a penalty fee twice and providing the incorrect flag value for the beginning of the next month. For this reason we define cs(RST)={{D2, RST}}.

Suppose now that we initially have Bal(A)=Bal(B)=2000 and Bal(C)=0, and that T_1 and T_2 are of types D2(500) and W2(800) respectively. Consider then the following execution schedule S:

T_1 step 1

T_2 step 1

T_2 step 2

T_1 step 2

Schedule S is a valid interleaving of the steps, since D2 and W2 are compatible. To check we can see that at the end of the execution Bal(A)=1700,

Bal(B)=1690, Bal(C)=10 and Tot=3400, which fulfills the restriction that Bal(A) + Bal(B) + Bal(C) = 3400 = Tot. Observe now, that any serial schedule of this two transactions would either have produced Bal(A)=Bal(B)=1700 or Bal(A)=Bal(B)=1690. We have thus shown in a simple example and via a simple use of the SK strategy, how non-serializable, consistent schedule may arise.

As discussed earlier, not having to enforce serializability could be an advantage. In the schedule above, T_2 can access the funds it needs without waiting for T_1 to finish. With a 2PL mechanism, T_2 would have had to wait. ○

Even though the example we have presented is extremely simple, we believe that compatibility sets can be defined in many real applications, including banking systems, airline reservation systems, system of libraries databases, and insurance company databases. However, the question remains whether the performance gains illustrated above will be significant. This is precisely the issue we intend to address in the next chapter.

Before finishing this section we return to discuss global lock release. As we can recall, local locks can be released as soon as a step finishes, but to enforce consistency, the global locks must be held until all transactions being interleaved are completed. To see this, consider two transactions T_1 and T_2 such that ty(T_1)=Y_1, ty(T_2)=Y_2, and cs(Y_1)=cs(Y_2)=$\{Y_1, Y_2\}$. After T_1 accesses o_1, T_2 also accesses it and goes on to lock and access object o_2. Suppose that T_2 finishes before T_1 does. If the global lock on object o_2 is released when T_2 finishes, a third transaction T_3, incompatible with T_1 and T_2, could produce a value read by T_1 and could read a value produced by T_2. The interleaving of the three transactions could violate consistency (i.e., there would be a cycle $T_3 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$ in the dependency graph). Therefore, the global lock on o_2 can only be released

when both T_1 and T_2 finish.

To accomplish this, each globally locked object o has a "release set" $Rel(o)$ associated with it. $Rel(o)$ contains the names of the transactions that have accessed object o. As a transaction T executes each step, it accumulates in a "wait set" $WAIT(T)$ the release sets of the objects it has accessed during the step. The names in $WAIT(T)$ will, in turn, at the end of each step, be accumulated in a "total wait set" $T\_WAIT(T)$. Transaction T reaches its termination point when it has finished and all the transactions in $T\_WAIT(T)$ have reached their termination point. The global locks on the objects accessed by T can be released only when T reaches its termination point. To insure this (see [Garc] for details), then for every object, o, such that $T \in Rel(o)$, $T\_WAIT(T)$ gets added to the $Rel(o)$, and T's name is removed from $Rel(o)$ when T finishes execution. The global lock of o will be released only when $Rel(o)$ is empty.

The release sets, as we have seen, play a very important role in the releasing of global locks, but their emptiness is not a sufficient condition to release the mentioned locks. We also have to make sure that the global lock on an object o is not released if there is still a transaction T in the system that holds a global lock on o, but has not yet accessed it. (Note that in such case, since T has not accessed o, it could easily happen that $Rel(o) = \emptyset$.) To met this requirement, the algorithms will use for each object o a set call $Pre(o)$. This is the set of transactions that have obtained a global lock on o but have not yet accessed it. The global lock cannot be safely released until this set becomes empty. To summarize, we can now say that to release the global lock on o, we need to have $Rel(o) = \emptyset$ and $Pre(o) = \emptyset$.

Chapter 3

THE SIMULATION

3.1. Introduction

A semantic knowledge (SK) based concurrency control mechanism is designed to perform better than a general purpose (GP) mechanism for a specific range of database applications. To verify this, one constructs examples where a GP scheduler produces larger delays than the SK mechanism does. However, the examples by themselves do not guarantee that performance will improve. For improvement, the number of times the "examples" arise in practice must be significant. Furthermore, the cost of running the SK mechanism must not outweigh the gains. Specifically, the SK mechanism may have one or both of these drawbacks:

(1) The overhead of the mechanism may be higher than that of an efficient GP one (e.g., two phase locking).

(2) Transactions outside the specific range of applicability (e.g., those that do not access the data in a structured fashion or that perform operations outside the allowed set) may not be able to access the data at all, or if they can, may encounter long delays (e.g., all the application transactions may have to be flushed first).

The objective of this chapter is to investigate these performance questions in more detail. We are not trying to show that one strategy is superior to another; our goal is simply to identify the conditions under which it may be advantageous to use an SK mechanism.

This is, of course, a very difficult task, given the large number of available strategies, applications, and hardware parameters. Thus, in this chapter we only study a relatively narrow portion of the spectrum, and compare this mechanism with the conventional two-phase locking (2PL) [Eswar]. The latter, as already said, is the most common GP concurrency control mechanism and is generally perceived to be efficient. We compare these mechanisms in a distributed database since the SK mechanism may be most applicable in this environment. We evaluate the strategies by postulating a relatively simple performance model and simulating it.

In section 2 we describe the simulation model and our results are presented in section 3. Finally, in section 4 we argue that the trends discovered in our case study may also be applicable in other contexts.

3.2. The simulation model

Reference [Garc] (see Appendix A) presents a concrete example of semantic knowledge concurrency control algorithms for a distributed DBMS. We use these algorithms to test the efficiency of the SK strategy in different situations, and to compare it to a GP, two phase locking (2PL) strategy [Eswar]. Our evaluations are based on a simulation model that is presented in this section. We feel that simulation is the correct approach here, since it would be difficult to capture the intricacies of the locking protocols with a mathematical analysis [Gray1], and since an implementation was not appropriate at this stage.

Although the simulation model is simple, we believe that takes into consideration the main features and parameters involved in the running of a distributed DBMS. In our search for a simple model, we made many assumptions, but

in each case we believe that they preserve the essence of the system and its performance. For instance, we only consider a system with two processing nodes because we think that this number is sufficient to model the transmission delays that transactions may encounter.

Following are the assumptions made in designing the simulation:

- There are two computing nodes interconnected via a reliable transmission line.
- Each node has its own local database. The databases are different, but of the same size.
- Transactions are of four different types: Local and nonlocal compatible (LC, NLC), and local and nonlocal incompatible (LI, NLI). Local transactions will execute just one step at the node at which they are submitted, whereas nonlocal will execute one step at each node.
- The compatibility sets are defined as follows: cs(LI)=cs(NLI)={} (i.e. no compatibility at all); and cs(LC)=cs(NLC)={{LC,NLC}}.
- The arrival of transactions to the system forms a Poisson process.
- The node that receives the transaction, and the objects it will access at each node are randomly chosen.
- The transaction's type is randomly chosen too, but it follows the distribution of probabilities for the different types given by the input parameter *Mix* (see below).
- Each step's computing time is fixed, but the locking time for the SK mechanism is larger than for the 2PL one.
- Read locks are not implemented. All locks in 2PL, and all local locks in SK,

are exclusive.

- Next we present the input parameters; in parenthesis are the range of values for the variable, and the most frequently (*typical*) used value in a simulation run.

  M: the size of the database (100 - 700 objects (half at each node); 200 objects). These are relatively small databases, but can be thought of as the high traffic portions of larger databases.

  K: the number of objects to be accessed by each transaction's step (1 - 8; 5)

  TT: the internode transmission time (5 -125 msec; 100 msec)

  λ: the mean interarrival time of transactions (100 - 400 msec; 150 msec)

  TC: the computing time of each step (held fixed at 100 msec)

  TL: the locking time of each step (held fixed at 10 msec for the SK mechanism, and at 8 msec for the 2PL one) ‡

  Timeout: time after which a transaction waiting for a locked object will be aborted (usually set to 300 msec)

  Nexttime: time that an aborted transaction has to wait to be resubmitted for processing (usually set to 300 msec)

  Mix: A four dimensional probability distribution vector, (P_{LI}, P_{NLI}, P_{LO}, P_{NLO}), where P_X is the probability of a transaction being of type X. Note that this parameter allows us to modify the degree of locality and/or compatibility of the

‡A discussion of these values will be found in section 3.3.

transactions. (Any combination of values, as long as the sum of the probabilities equals 1; most frequently used vector was (.25, .25, .25, .25))

We now describe, in a simplified manner, how the simulation proceeds:

a) **Local transactions:**

Upon submission of the transaction to the local node, it immediately tries to lock, first globally and then locally, the objects needed for the step. Once all required objects have been locally locked, simulation of the locking and computing time ($TL + TC$) of a step takes place. The transaction then releases the local locks and tries to release the global locks. (When the transaction is being interleaved with another compatible transaction, the global locks may have to remain set even after the transaction that set them completes. More on this later.) The transaction's processing is now finished. (To make local transaction processing simple, we decided to avoid local deadlock situations by requesting the locks of objects by ascending identification number.)

b) **Nonlocal transactions:**

Immediately after being submitted, the transaction tries to globally and locally lock the required objects at the receiving node. Upon successful execution of this task, it waits $TL + TC$ simulated time units and and releases the local locks. Global locks will remain, to allow valid interleavings of compatible transactions. The transaction now waits now TT (internode transmission time) units, and then tries to lock the required objects, in the same manner, at the second node. After obtaining the remote locks, the transaction waits again $TL + TC$ units. Permission is then given at the the

second node to release the local locks, and to try to release the global ones. (Our previous comment on global lock release also applies here.) The transaction returns, finally, to its node of origin, after a TT transmission time is simulated, and tries to release the global locks left here. At this point, the transaction has finished its execution.

Each time a transaction fails to get a desired lock, it will be placed on one of two waiting queues for that object, depending on which kind of lock is requested. When a lock is released, either local or global, the oldest waiting transaction, in the respective queue, will be reactivated, i.e., it will continue execution at the point it was suspended. As for deadlock situations, we know that they cannot occur within a node, but they can arise between nodes. To deal with such situations we decided to use a timeout mechanism. A timed out transaction is aborted (following [Garc]'s algorithms) and resubmitted to the system a fixed amount of time later.

Note that if one defines the compatibility sets of all types of transactions to be empty, i.e., no compatibility at all, then the semantic knowledge algorithms will enforce two phase locking. Thus the same simulator was used for both concurrency control strategies. With 2PL all transactions are made incompatible; and with SK, the larger step locking time (TL) was used to model the higher complexity of the algorithms. (The reader should bear in mind this last fact while examining the results of the simulation in the next section.)

One last important consideration that should be mentioned here: In order to get accurate statistics, each simulation was run until the results lay in a small confidence interval. The size of this confidence interval was less than 15% of the value of the mean transaction response time. The response time is the time a

transaction takes from the moment it is submitted to the moment it finishes execution, i.e., the moment the transaction completely finishes doing its work at both nodes. This time does not include the time spent for previous executions, when the transaction was aborted.

3.3. Results

Our objective in this section is to present some of the results on the use of a semantic knowledge based concurrency control mechanism. The results we present were summarized from hundreds of simulation runs using the model explained in the previous section. They will compare the performances of the SK and the 2PL methods.

In the early stages of experimenting with the simulator, it became evident (as we had suspected) that under the proper conditions a SK mechanism would significantly improve the system's response time as compared to 2PL. For example, Figure 3.3.1 shows the performance improvement of SK as the percentage of NLC (non-local compatible) transactions increases. This figure plots the average transaction response time as a quotient 2PL/SK (vertical or y axis), versus the fraction of NLC transactions simulated (horizontal or x axis). The remaining transactions were equally distributed among the LI, NLI, and LC types. All other input parameters were set to their typical values. (See Section 3.2) (In the following graphs all input parameters will be "typical," unless otherwise noted.) In Figure 3.3.1, as well as in Figures 3.3.2, 3.3.4, 3.3.5, 3.3.7 and 3.3.8, we have drawn a line at $y = 1$. This line marks the boundary between the regions where each strategy performs best. Since the average transaction response time is given as the quotient 2PL/SK, then for any value on the x axis whose image is above the

$y = 1$ axis, SK performs better, else 2PL is faster.

The reason for the improved SK performance as P_{NLC} grows is, of course, the parallelism induced by compatible transactions, as illustrated in Section 2. The increase in the percentage of NLC transactions implies an increase in the average size (number of objects accessed) of transactions. (Remember that non local transactions access twice as many objects as locals do.) When using 2PL this size increase implies a quadratic increase in the probability that a transaction will ever have to wait (find an object locked) [Gray1]. When using SK, the increase in the percentage of NLC transactions implies, not only an average transaction size increase but also an increase in the number of sharable objects, avoiding then many "waits" that would have occurred with 2PL. Therefore, we intuitively do not expect the number of "waits" to go up in this case. In summary, we expect that as the percentage of NLC transactions increases, the transaction's response time with 2PL will increase with the second power of the transaction's size, but we do not expect this to happen when using SK. It should therefore, be no surprise that the 2PL/SK transaction response time, which shows the better performance of SK over 2PL, increases approximately in a quadratic fashion as the percentage of NLC transactions increases.

The same behavior was observed as the mean interarrival time of the transactions (λ) was varied (See Figure 3.3.2). As the value of λ decreases, a higher load is placed on the system, and SK pays off. However, for light loads, the smaller locking time of 2PL outweights the increased parallelism of SK.

From our understanding of semantic knowledge, it also seemed reasonable to expect the other parameters, especially the internode transmission time (TT), the size of the database (M), and of the number of objects to be locked by each

transaction's step (K), to have similar effects on performance. That is, increasing TT, decreasing M or increasing K, should show the same performance improvements (of SK over 2PL) as decreasing λ, because changing these variables in the mentioned directions will increase transaction "traffic" in the system: decreasing M or increasing K, increases the number of objects locked at any moment relative to the size of the database; increasing TT causes objects accessed by non-local transactions to remain locked for longer periods of time. These ideas were confirmed by running additional tests and obtaining graphs (not shown here) similar to the one of Figure 3.3.2. (Another result not shown here, is that with high "traffic" the number of aborted transactions was less with SK than with 2PL.)

An increase in the relative number of database objects locked, or an increase in the average period that an object remains locked, represents a higher probability that an object is locked when a requesting transaction tries to access it. It then seems reasonable that we express our performance results in terms of this variable. This probability, P, can be computed by the simulator simply by counting the number of conflicts (i.e., times a requested lock is not immediately available) and dividing by the total number of lock requests made during a 2PL simulation run.

In a moment we will see that P is indeed a good predictor of system performance. However, we first present a simple way of estimating P without having to resort to a complete simulation. If

t= average transaction response time in normal 2PL processing, and

λ= mean interarrival time of transactions,

then the expected number of transactions in the system is λ/t (Little's result [Klein]). The fraction of locked objects at any instant of time, that is P, can therefore be approximated by

$$PRE = \frac{(t/\lambda)\ K^*}{M}$$

where

M= number of objects in the database; and

K^*=average number of objects that a transaction is holding locked during its lifetime. Local transactions, we already know, access K objects, whereas non locals will access 2K objects (K at each node), but they will not hold locks during their complete lifetime. K^* has to be, therefore, a function of K and the probability vector *Mix*. Its calculation follows.

Our simulation model forces a local transaction to have locked all its needed objects during its lifetime. Non local ones hold locks in the first node for all their execution time, whereas the objects in the second node are locked just during the time that the transaction is processing in that node. Let θ_l and θ_{nl} be the probabilities that an object required by a local, or a non local, transaction be locked at any moment of the transaction's lifetime. Certainly θ_l=1.

To calculate the value of θ_{nl} we take the *typical* values (see section 3.2) for *TT*, *TC*, and *TL* (*TL* for the 2PL case since we want to estimate the number of conflicts in a non application dependent environment). Due to our simulation design, a non local transaction will hold locks on its first node for all of its lifetime, i.e., for 416 msecs. (416 msecs is the total of the locking and computing times (TL + TC) at the first node, the transmission time (TT) to the second node, the locking and computing times at the second node, and the transmission

time back to the first node.) The locks on the second node will remain just for 108 msecs (locking and computing times at second node). Therefore, half of a non local transaction's required objects will be locked for 416 msecs and the other half for just 108. This implies that,

$$\theta_{nl} = \frac{(\frac{1}{2} * 416) + (\frac{1}{2} * 108)}{416} \approx 0.629$$

Having the values for θ_l and θ_{nl} we can now calculate K^*:

$$K^* = (\textit{fraction of locals}) * \theta_l * K + (\textit{fraction of nonlocals}) * \theta_{nl} * 2K$$

$$= (P_{LC} + P_{LI}) * \theta_l * K + (P_{NLC} + P_{NLI}) * \theta_{nl} * 2K$$

$$= K * \Big[(P_{LC} + P_{LI}) * \theta_l + (P_{NLC} + P_{NLI}) * \theta_{nl} * 2 \Big]$$

$$= K * \Big[(P_{LC} + P_{LI}) * 1 + (1 - (P_{LC} + P_{LI})) * 0.629 * 2 \Big]$$

$$= K * \Big[(P_{LC} + P_{LI}) + 1.258 - 1.258 * (P_{LC} + P_{LI}) \Big]$$

$$= K * \Big[1.258 - 0.258 * (P_{LC} + P_{LI}) \Big] \quad \bigcirc$$

The advantage of the formula for *PRE* over the one for P is that it involves system parameters that can be estimated for a given application. If the application is already implemented on a 2PL system (and we are considering switching to a SK mechanism), the value of t can be easily measured without having to instrument concurrency control, as would be necessary if P were measured. Of course, *PRE* is only a approximation to P, but Figure 3.3.3 shows that (at least for our model) it is a fairly good one.

To confirm our hypothesis that P and PRE are good predictors of performance, we graphed our previous results as a function of PRE. (Since P and PRE are roughly equivalent we will only deal with PRE.) Figure 3.3.4 exhibits four different curves, each of them showing the system's performance as four parameters (λ, K, TT, and M) were varied. The fact that the performance curves for the different parameters varied follow almost the same path shows that the performance of the system depends very much on PRE and not so much on the parameter varied. This an indication that PRE is a good system predictor. This figure is for the case when all types of transactions are equally distributed. Similar graphs, that reenforce our hypothesis, can be obtained for different transaction mixes.

A graph like the one in Figure 3.3.4 can be useful for determining, for a given transaction mix, the cases where the use of semantic knowledge pays off. For instance, in the particular case of Figure 3.3.4, if the probability of an object causing a conflict is less than roughly 0.02, then 2PL is a good choice, but if PRE is greater than about 0.035, then SK works better. If PRE is between the two values, we are unable to predict (without more detailed information) which mechanism is superior, since the curves cut the $y = 1$ axis at different places in this range.

For a given transaction mix the relative performance of SK over 2PL is driven by the probability that a requested object is locked. But how does the mix affect this performance? If we have very few conflicts, 2PL should be superior, regardless of the mix. However, if we have a high number of conflicts, *and* if there are sufficient compatible transactions, the SK may save enough conflicts and be worthwhile. These considerations led us to consider a second predictor,

the *probability of saved conflicts* or *PSC*. Predictor *PSC* will estimate the probability, when using SK, that a requested object is locked by a transaction compatible with the requesting transaction, i.e., the probability of "saving a conflict". Intuitively we would expect SK to outperform 2PL as *PSC* grows.

To calculate *PSC* we proceed as follows:

Let T_1 be any transaction that holds a lock on an object;

T_2 a transaction wanting to access the object locked by T_1; and

P(X): Probability of event X;

then

$$PSC = PRE * \mathrm{P}(T_1 \text{ and } T_2 \text{ are compatible})$$

$$= PRE * \mathrm{P}(T_1 \text{ and } T_2 \text{ are both of type LC or NLC})$$

$$= PRE * \{\mathrm{P}(T_1 \text{ of type LC}) + \mathrm{P}(T_1 \text{ of type NLC})\}$$

$$* \{\mathrm{P}(T_2 \text{ of type LC}) + \mathrm{P}(T_2 \text{ of type NLC})\}$$

Since the transaction probabilities are given by the input parameter $Mix = (P_{LI}, P_{NLI}, P_{LC}, P_{NLC})$, then:

$$PSC = PRE * (P_{LC} + P_{NLC}) * (P_{LC} + P_{NLC})$$

$$= PRE * (P_{LC} + P_{NLC})^2$$

Figure 3.3.5 exhibits four curves that show the compared performance of 2PL and SK as a function of *PSC*. Each different curve shows the system's performance as the percentage of the given transaction type is changing. The curves were obtained by varying from zero to one the probability of the given type, and distributing equally the remaining probability to the other types. The fifth curve

is the same curve that appeared in Figure 3.3.4 showing how M influences the system's performance, but here it is plotted versus *PSC*. It is included for comparison purposes.

The curves of this figure do not bunch together as the ones of Figure 3.3.4, since for almost every value of *PSC*, the transaction mix for each curve is different. (In Figure 3.3.4, *PRE* was predicting the system's performance, but only for a fixed transaction mix.) However, the curves of Figure 3.3.5 cross the $y = 1$ axis in a relatively small interval, and this means that *PSC* may be useful, as we suspected, to decide if SK is advisable. Specifically, if *PSC* is greater than 0.02, we can be fairly confident that SK will outperform 2PL. Similarly, if *PSC* is less than 0.005, chances are SK is not advisable. For intermediate values, a more detailed analysis is necessary. We discuss the implications of these results in the conclusions section.

Now that the significance and potential value of the two predictors, *PRE* and *PSC*, is well understood, we are going to present one more result that compares the performances of the two different locking mechanisms. This result refers to the system's total throughput, i.e., the number of transactions that are fully processed by the system per unit of time. Figure 3.3.6 shows the total throughput as a function of *PRE* and of *PSC* at the same time. To make this possible we have drawn just one curve, but two scales, corresponding to the two different predictors, on the x axis. The y axis gives the the total throughput as a ratio SK/2PL. We have again drawn a line at y=1 to tell us when SK performs better than 2PL. The graph shows clearly how SK outperforms 2PL as the values of *PRE* and *PSC* increase. It is interesting to notice that again the performance curve cuts the 1-axis at values in the intervals already established in figures 3.3.4

and 3.3.5 ([0.02, 0.035] for *PRE*, and [0.005, 0.02] for *PSC*).

Our next result, supported by figures 3.3.7 and 3.3.8, addresses the choice of SK/2PL locking ratio in our simulation. This ratio was taken as 10/8 because we estimate (from studying the code) that the time complexity of SK is about 25% higher than that of 2PL. The value of 8 msecs for 2PL locking time, or 8% of the CPU processing time for a step, roughly agrees with published values [Gray3]. (Incidentally, it is because the SK overhead is not much larger than the 2PL overhead that 2PL does not beat SK by a wide margin when the load is light in figure 3.3.2.)

However, our results are not highly sensitive to actual locking time values used, as figures 3.3.7 and 3.3.8 show. Both figures show how little the system's performance, as a function of λ and *PRE*, changes when the locking ratio is changed to 16/8 and to 40/20 (This last being an extreme case). The curves for these last two values are not as steep as for the 10/8 ratio, meaning that the performance of SK degrades a little compared to that of 2PL, but their behavior is certainly similar. Specifically, note that in figure 3.3.8, the curve for the 40/20 ratio cuts the y=1 axis just 1% to the right of the curve for the 10/8 ratio. This means that our prediction of when it is worth considering the use of semantic knowledge varies little, even in the case when there is quite a difference in the locking ratios.

We now turn our attention to the second potential drawback of an application dependent concurrency control mechanism: it may delay transactions outside the intended application. (See Section 3.1.) In the case of a SK mechanism, the problem arises when a string of interleaved compatible transactions prevent an

incompatible transaction from accessing the data it needs. To fully understand this potential problem, we ask the reader to review the details of the global lock release mechanism in section 2.3.

From the discussion of global lock release in section 2.3, it is not difficult to see why incompatible transactions could suffer significant delays: a long string of compatible transactions could cause the release sets to grow and the global lock release could be postponed indefinitely. The compatible transactions (i.e., those within the application) are not affected, and if there are many of them, the average system performance could still be good. We should point out in passing that, even though the details given are specific to the SK mechanism, the problem of delayed transactions is not. In any application dependent (AD) system, transactions that cannot cope with the non-2PL interleavings, must be postponed until they can be executed in a conventional way.

To study the seriousness of this problem, we can study the response time of incompatible transactions, or more directly, the size of the release sets. In both cases, we discovered that the problem is not serious, even when the degree of compatibility, i.e., the percentage of compatible transactions ($P_{LC} + P_{NLC}$), is large. Evidently, the probability that a large release set forms is negligible.

Some of our results on release sets are shown in Figure 3.3.9. They show that (a) the average size of the release sets grows very slowly as *PRE* increases; and (b) the average size of these sets is relatively independent of the degree of compatibility. The figure exhibits three different curves plotting the average size of the release sets versus the degree of compatibility. All other input parameters are typical, but each curve has a different value of λ. Varying λ, as we already know, varies the value of *PRE*, whose influence in the size of the release sets we

wanted to measure. Therefore each curve has a different mean value of *PRE*. Note in the graph that the mean sizes of the release sets increase, but very slowly, as *PRE* increases. (For a value of *PRE* of 0.218, representing a very large number of conflicts, the release sets are still very small.)

Even though these results are positive, it may still be advisable to place a limit on the number of compatible transactions that can be interleaved at a time to avoid very rare but intolerable delays on incompatible transactions. These limits are easy to implement, and as our results show, will rarely affect the performance of compatible transactions. Similar limits can be placed on other AD mechanisms.

Before closing this section we will summarize results that show how semantic knowledge can aid in processing Long Lived Transactions (LLT) [Gray2]. We consider the subject of LLTs, and the use of a SK concurrency control mechanism to help manage LLTs, to be very relevant to this thesis. We will therefore devote a full chapter (chapter 7) to the discussion of this subject. For the purpose of this section we will limit the discussion of LLTs to a very simplified overview. A LLT, as its name indicates, will take a very long time to process because it uses considerably more distributed DBMS resources and/or accesses many more database objects than normal transactions. This can result in long delays for transactions trying to access objects already locked by the LLT. However, if a LLT is compatible with many other transactions, then a SK mechanism could reduce the delays.

In figure 3.3.10 we have considered a case where half of the transactions are compatible and where all transactions have one step, except for the compatible LLT's that have two very large steps (they will access ten times as many objects

as normal steps). These LLT's arrive every 500 normal ones. The graph exhibits the average response time of transactions, averaged over very short time intervals (every 20 transactions). Most of the time, these averages fall within a small range, and this is indicated by the two horizontal lines. Every 500 transactions, a LLT arrives and the average response time for the next period is outside the normal range. These values appear as dots (for 2PL) and asterisks (for SK) above the horizontal lines. Note that in most of those cases, the system responded better when using SK than when using 2PL.

3.4. Conclusions

In this chapter we have studied a specific instance of an application dependent (AD) transaction processing mechanism and compared it to conventional two-phase locking. We observed that the probability of conflict (P, PRE) was the dominant performance factor, and that the probability of saved conflict (PSC) could be useful in identifying the situations where the SK mechanism was advisable. We found that large release sets (caused by long strings of interleaved compatible transactions) did not represent a problem for non-compatible transactions. Finally, we found that even if the AD mechanism does not improve average performance, it may improve the performance when long lived transactions appear.

Strictly speaking, our results are only valid for the model and algorithms used. However, since PSC appears to be so good in determining the usefulness of the SK mechanism, and since it has intuitive appeal, we conjecture that PSC will also be a good predictor in many other cases. That is, we hypothesize that if PSC (i.e., the probability that a requested object is locked, but a wait is avoided)

for any AD mechanism in a given application is on the order of magnitude of 0.01 or more, then the mechanism may outperform a conventional mechanism; if it is less than this value, in all likelyhood one should use the conventional strategy. (PSC can be estimated, as we did here, from the probability of conflict and the transaction mix.) The intuitive reason behind our statement, as discussed earlier, is quite natural: if less than roughly one percent of the conflicts are avoided, the gains will be insignificant. This rule is only applicable to average performance; an AD mechanism may still be advantageous for certain transactions (e.g., LLTs). Our conjecture will have to be tested for other mechanisms.

If this conjecture holds (as it does for SK), what does it say about AD mechanisms in general? For many large databases, references are so dispersed that the probability of conflict (and consequently PSC) will be much less than 0.01. Clearly, for these systems a conventional mechanism is best. However, if the database is small or if there are frequently accessed portions (e.g., the root of a B-tree), and if the transaction rate is high, then an AD mechanism may be worthwhile.

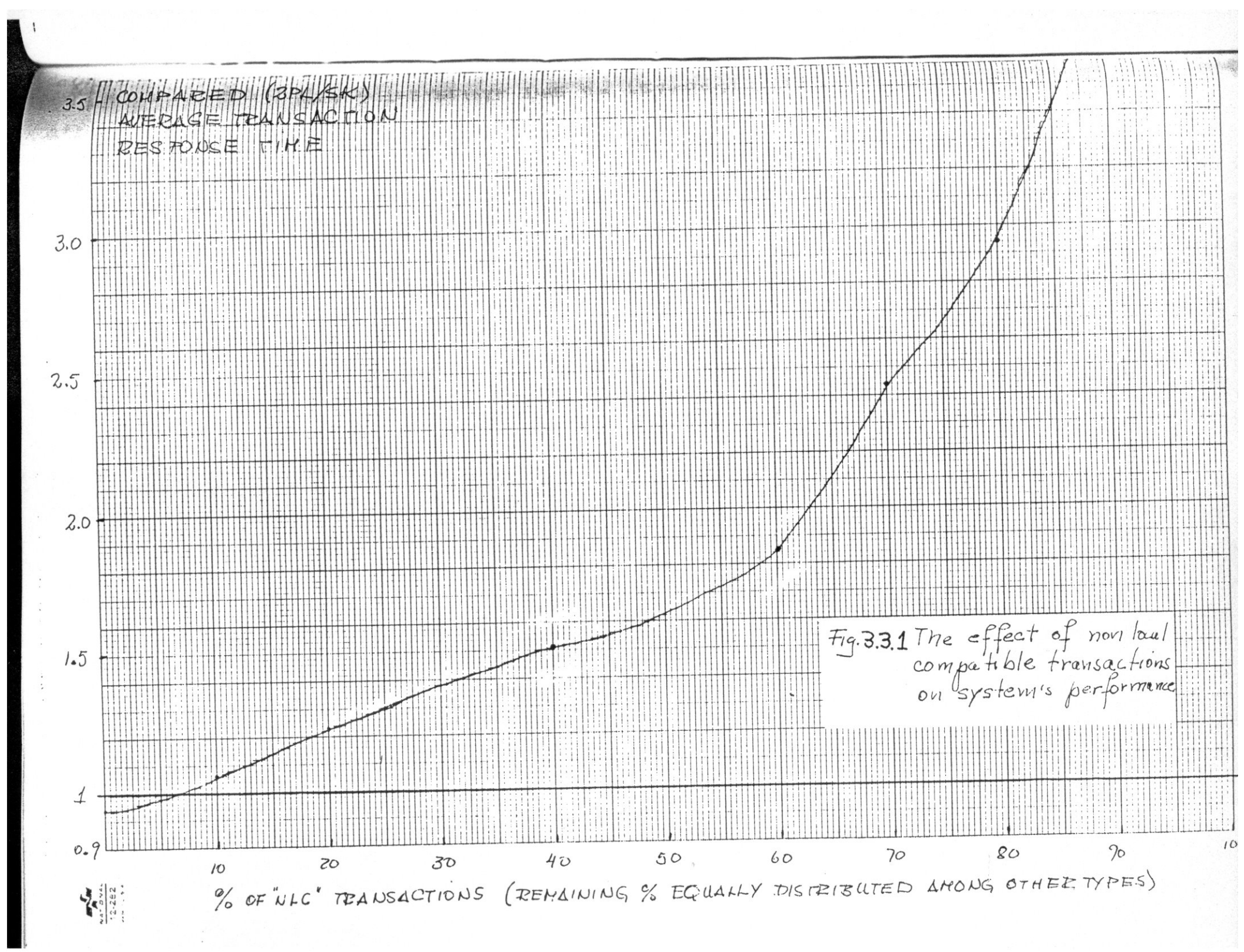

Fig. 3.3.1 The effect of non local compatible transactions on system's performance

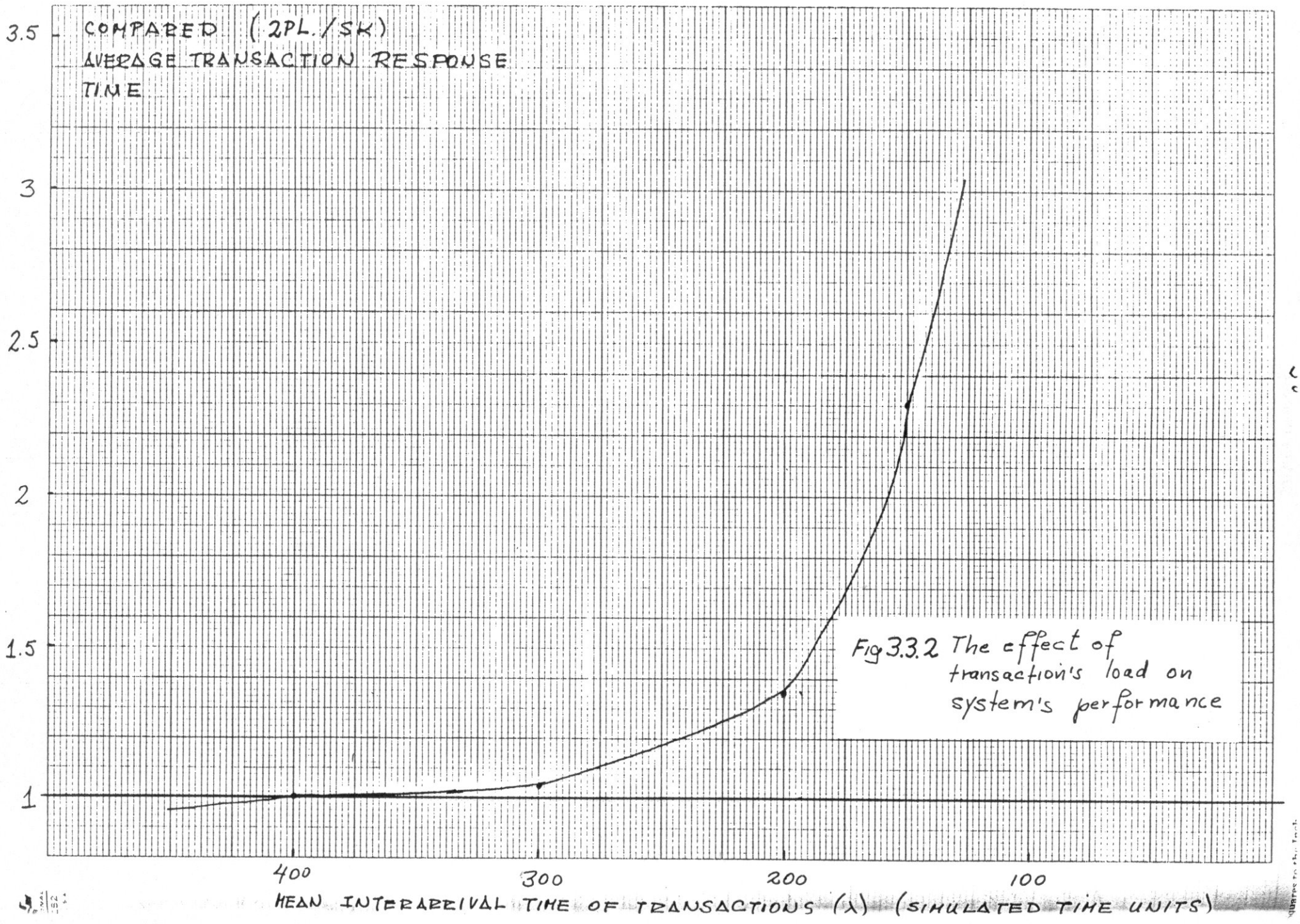

Fig 3.3.2 The effect of transaction's load on system's performance

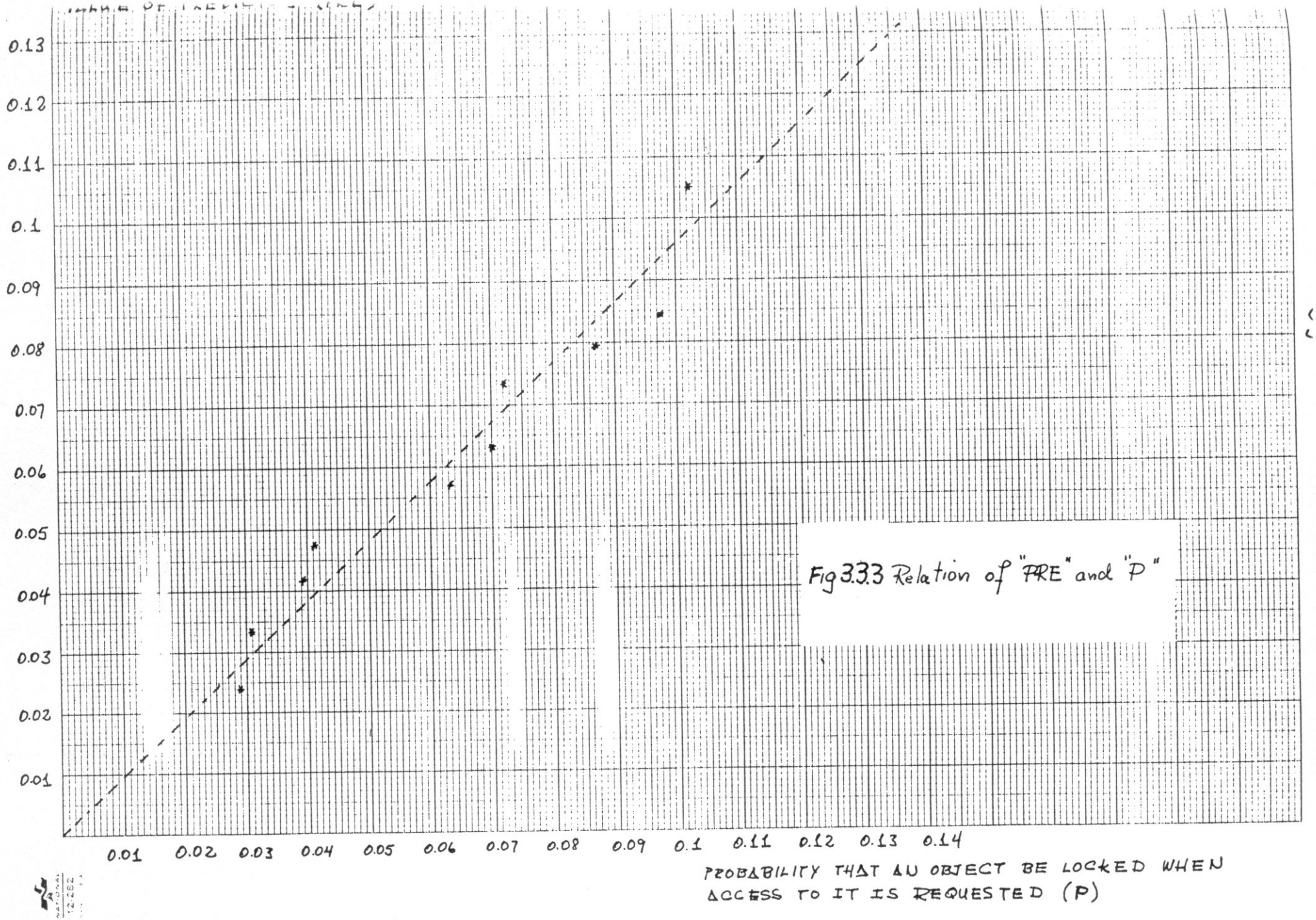

Fig 3.3.3 Relation of "PRE" and "P"

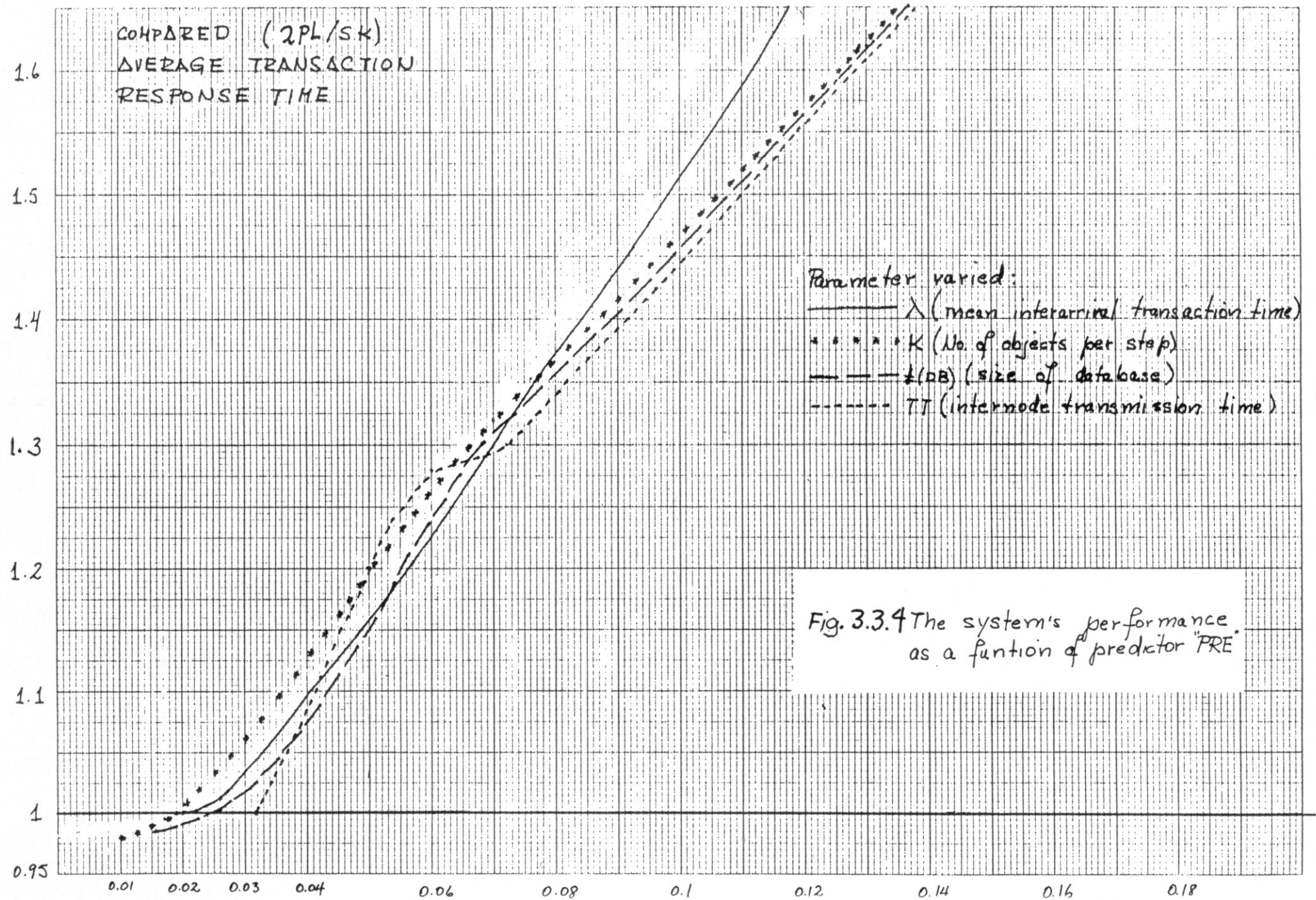

Fig. 3.3.4 The system's performance as a funtion of predictor "PRE"

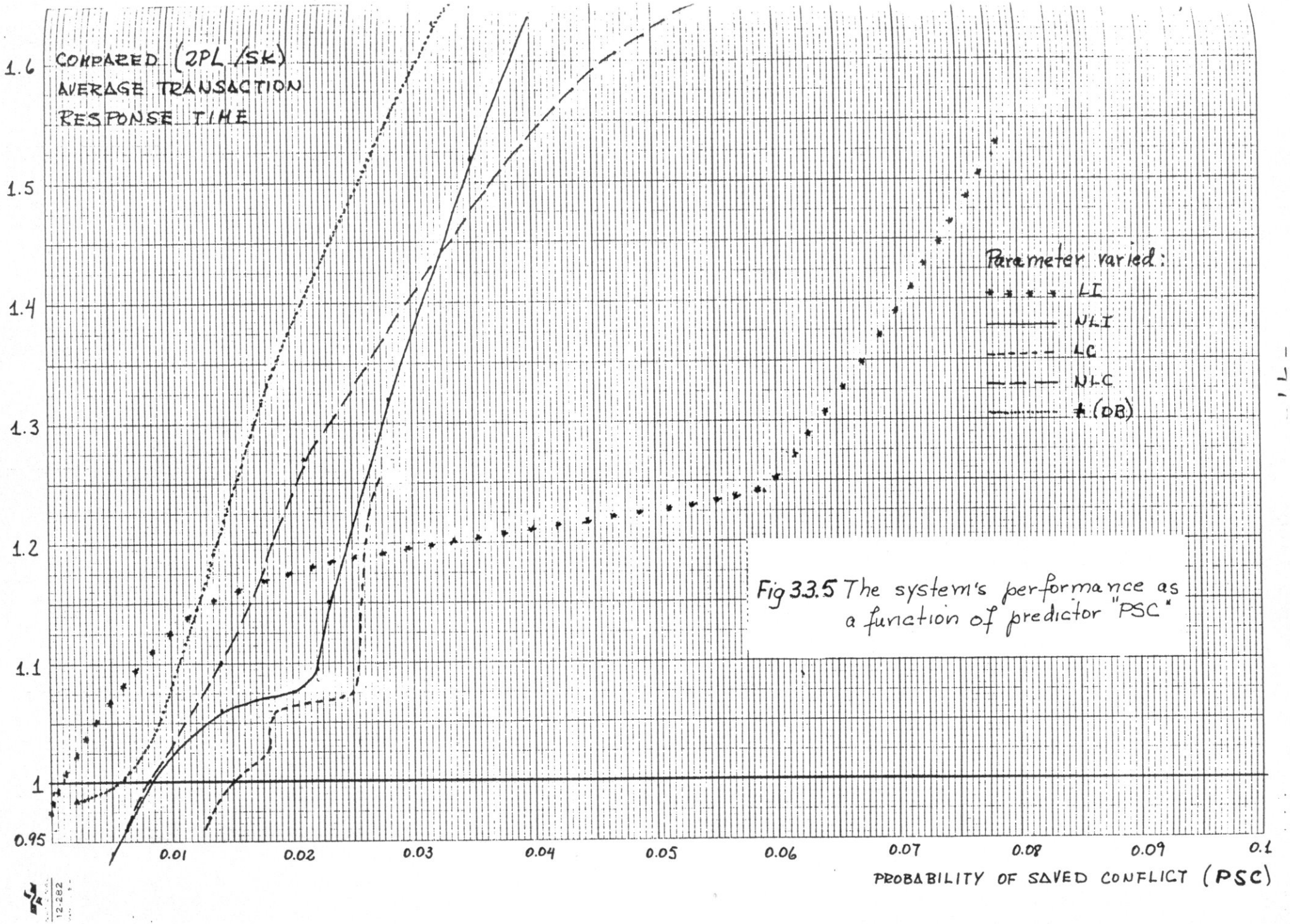

Fig 3.3.5 The system's performance as a function of predictor "PSC"

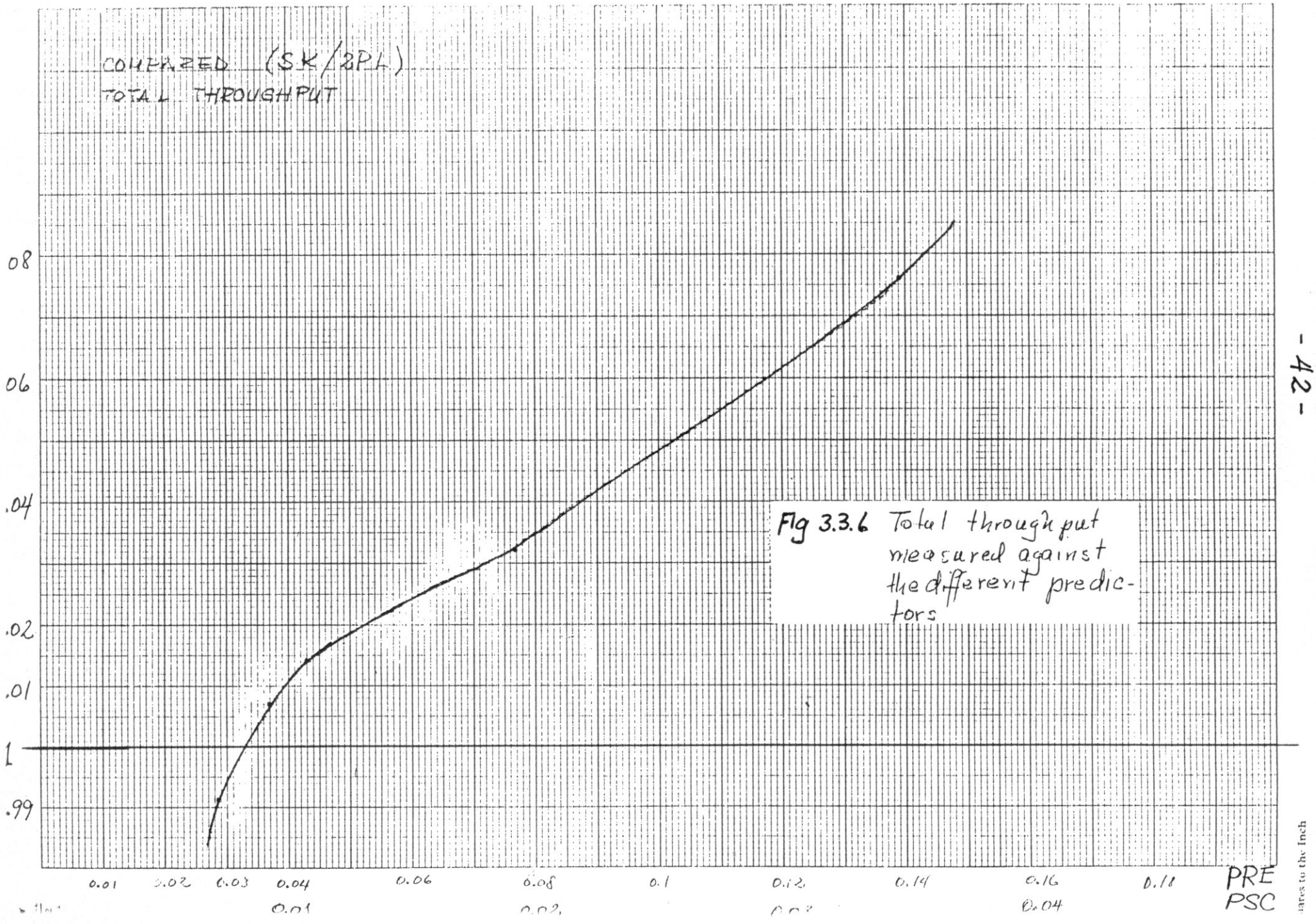

Fig 3.3.6 Total throughput measured against the different predictors

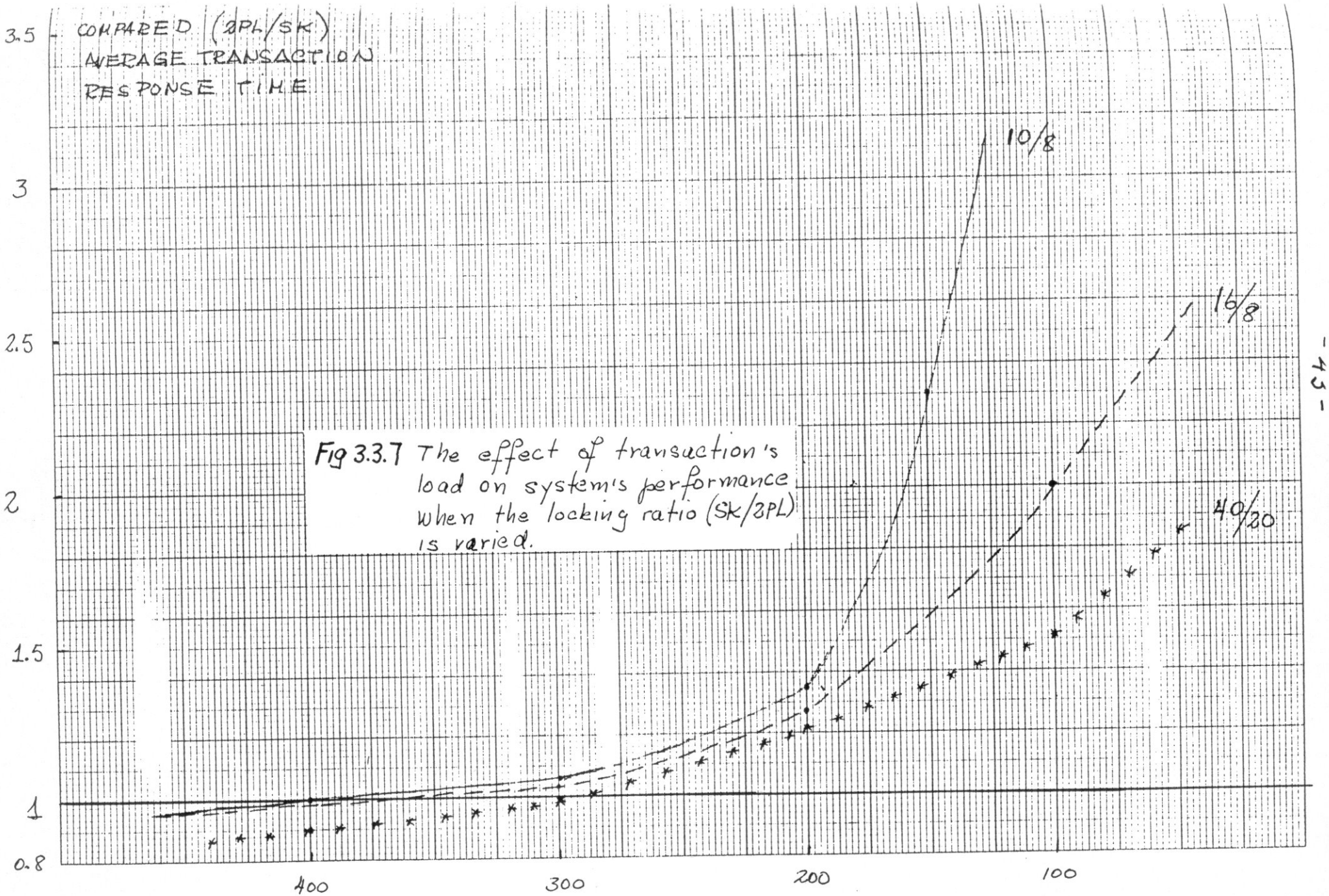

Fig 3.3.7 The effect of transaction's load on system's performance when the locking ratio (SK/2PL) is varied.

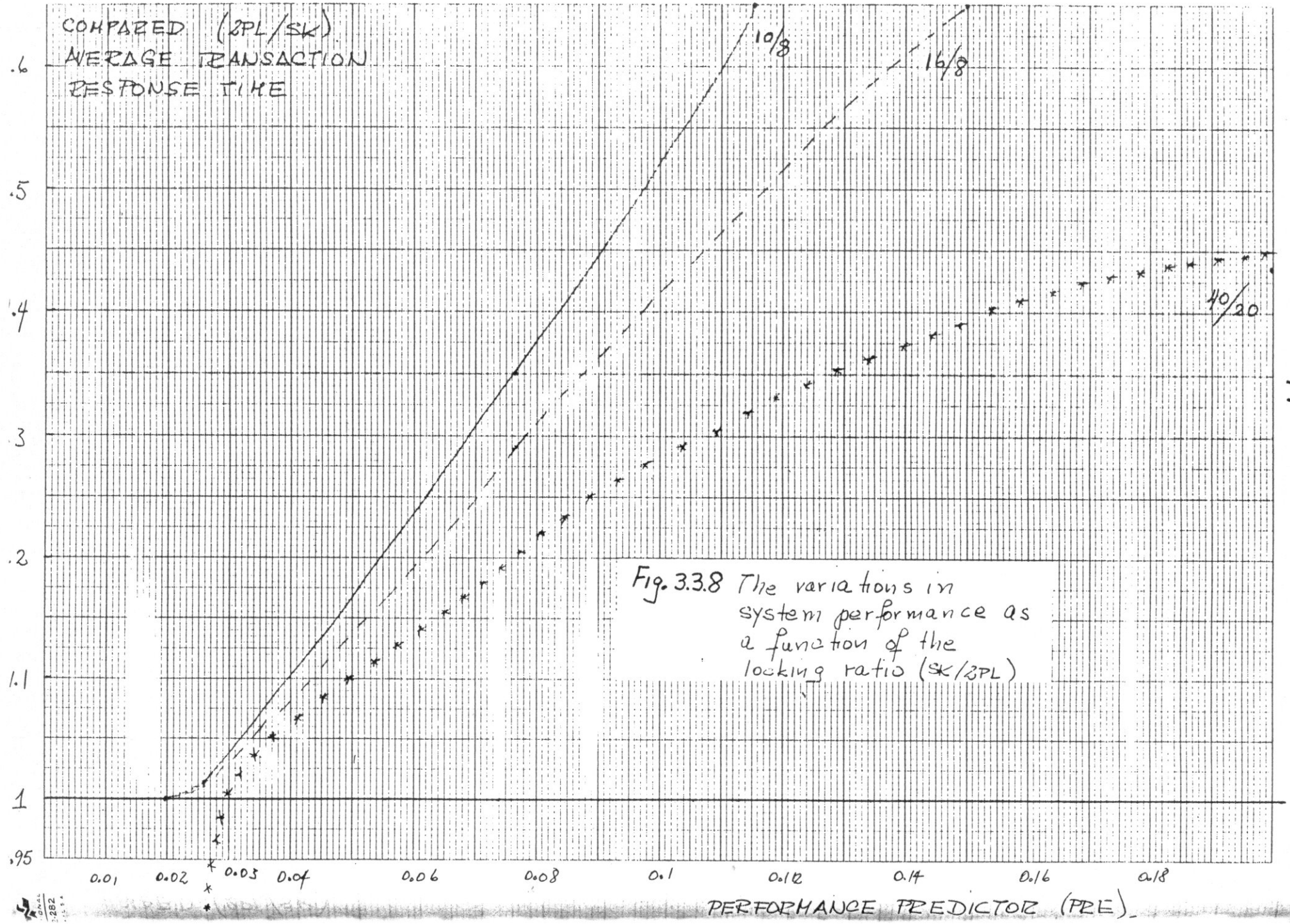

Fig. 3.3.8 The variations in system performance as a function of the locking ratio (SK/2PL)

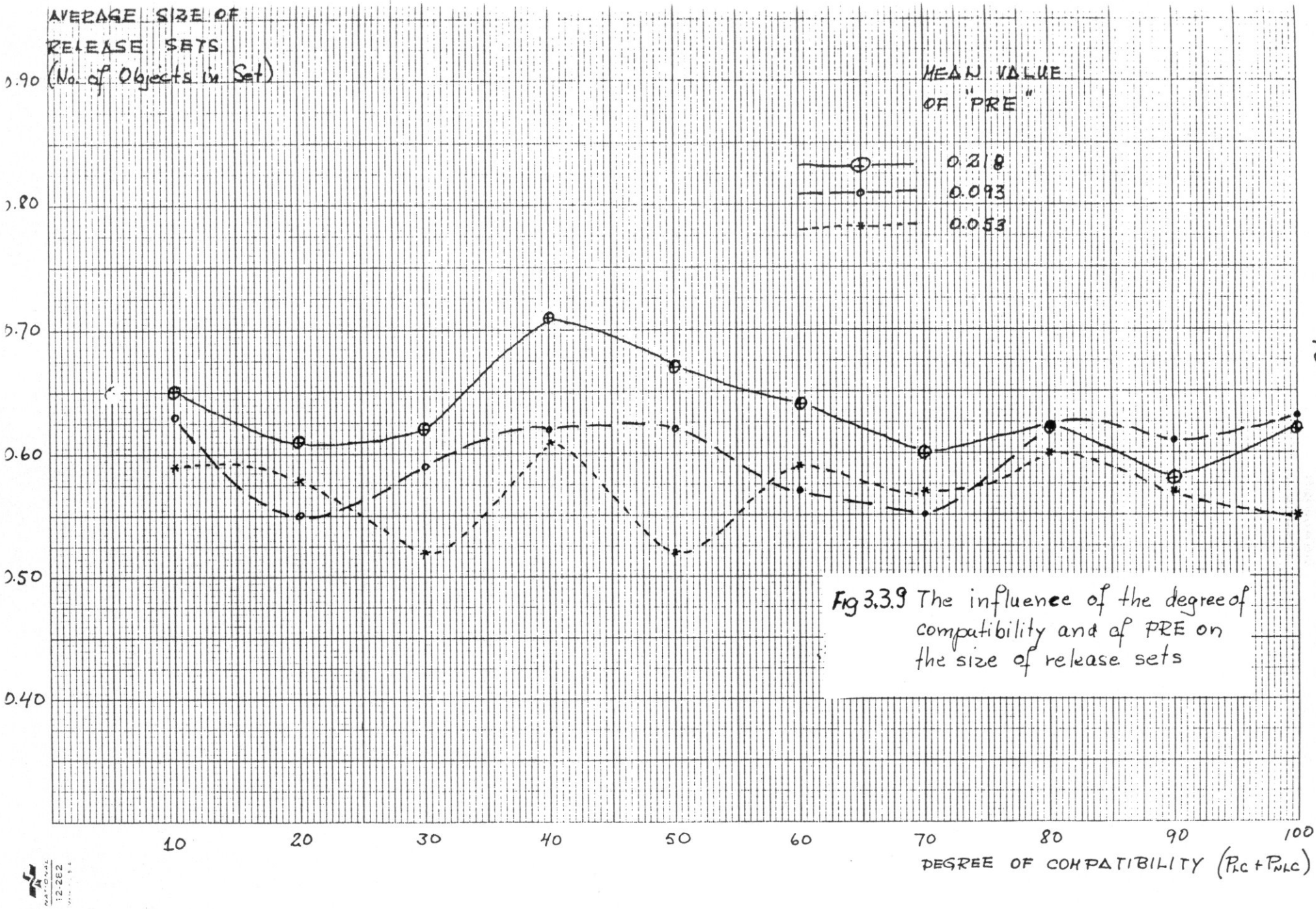

Fig 3.3.9 The influence of the degree of compatibility and of PRE on the size of release sets

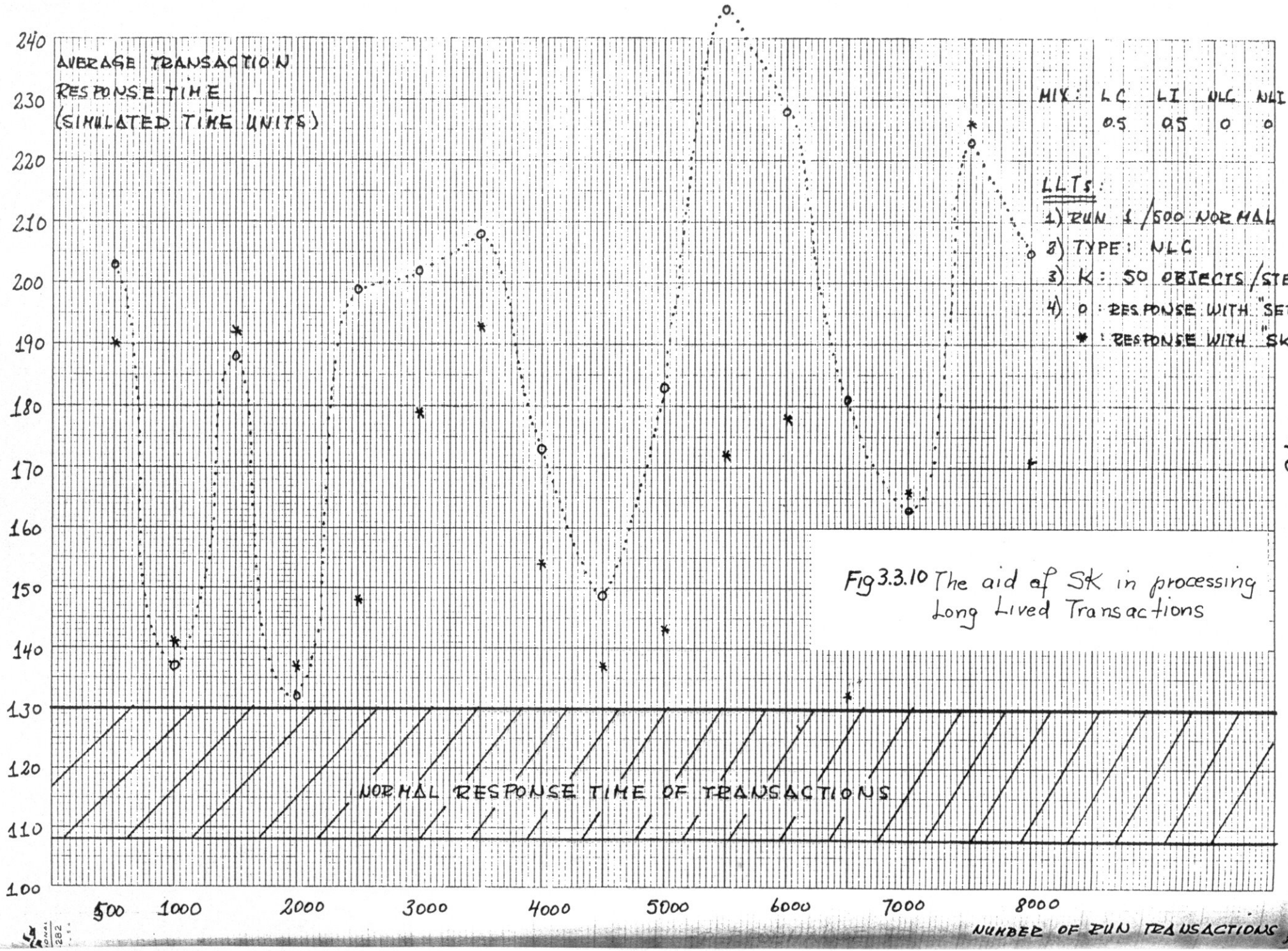

Fig 3.3.10 The aid of SK in processing Long Lived Transactions

Chapter 4

ALGORITHMIC VARIATIONS

4.1. Introduction

Results of the simulation comparing two different concurrency control mechanisms, one enforcing two phase locking (2PL) and the other based on the concept of semantic knowledge (SK), showed that considering the use of SK is advisable just under certain conditions. One of the reasons SK is not always a better algorithm than 2PL resides in its higher CPU overhead. Another reason is the potential it has to make a transaction wait for long periods of time when its type is outside the interleaving descriptor used to lock the object it desires. Such waiting, as we know, is caused by our algorithm, since it has to ensure that only compatible transactions are interleaved. We remind the reader that our algorithm is based on Appendix A of [Garc], a replica of which appears in Appendix A of this thesis.

In this chapter we propose variations on Garcia-Molina's algorithms. Such variations will be based on the knowledge of the problems mentioned above. Section 2 will propose how to process more effectively transactions whose types have an empty compatibility set. This new algorithm will make use of only a subset of the facilities (variables) of the SK algorithm. This property will allow us to combine both algorithms into a new and general concurrency control algorithm, that will take advantage of the semantics of the transactions more fully than the current SK algorithm. A brief example will illustrate the improvement that this variation can achieve. In section 3 we will report about another variation of the algorithm. This variation was originally suggested by Garcia-Molina, and is

based on restricting the compatible sets to hold only local transactions. Section 4 will point out some conclusions very briefly.

4.2. First Variation

Several graphs based on the simulation results have shown that for a low degree, percentage, of compatible transactions submitted to the DBMS, our SK based concurrency control mechanism is easily outperformed by one which is based on 2PL. If there is a low degree of compatibility then there is a high probability that many transactions have an empty compatibility set. Now, there is no need for transactions with empty compatibility sets to be processed by our original SK algorithm (see Appendix A) (*Algorithm* 1), as their incompatibility with other transactions does not allow them to take advantage of that algorithm. Transactions in this class, which will henceforth be known as class E, could therefore run under a variation of our concurrency control mechanism, that resembles a 2PL type of mechanism. We are, of course, not suggesting that there will be two different concurrency control mechanisms in our DBMS. Transactions with an empty compatibility set can bypass the burdensome bookkeeping of algorithm 1 whose purpose is to guarantee that only compatible transactions will run together, making it impossible for data inconsistencies to occur. A new algorithm, *Algorithm* 2, considering these factors follows:

Algorithm 2:

0. When transaction T enters the DBMS

If $cs(ty(T)) \neq \emptyset$ **then**

call *Algorithm* 1

Else

call *Algorithm* 1^* (see below)

Algorithm 1^*:

$<<$ This is an algorithm to effectively process transactions with an empty compatibility set. For simplicity we follow here the conventions on variable naming and algorithm's steps numbering, set forth in algorithm 1 $>>$

I. Before a transaction T starts

$LL\_SET(T) \leftarrow \emptyset$

II. Before transaction T starts a step

"This step is not needed in this algorithm since no global locking will be necessary, as is the case for algorithm 1."

III. Before a step of transaction T is allowed to access object o

IF $\neg GL(o)$ AND $\neg LL(o)$ THEN

$LL(o) \leftarrow true$

ELSE "wait and try locking later";

$LL\_SET(T) \leftarrow LL\_SET(T) \bigcup \{o\}$;

IV. When a transaction T completes a step at node X

IV.A IF T is a local transaction

$<<$ T has completed its only step. $>>$

FOR $p \in LL\_SET(T)$ DO

BEGIN $<<$ release locks $>>$

$LL(p) \leftarrow$ false;

END;

$LL\_SET(T) \leftarrow \emptyset$

IV.B IF T is a non local transaction completing a revocable step, or a counterstep

<< In this algorithm there are no revocable or countersteps, therefore this step is not needed >>

IV.C If T is non-local completing a step (except its last step)

"do nothing"

V. If transaction T must be aborted

<< T should not be aborted if it is in the process of releasing locks>>

FOR all $p \in LL\_SET(T)$ in executing node DO

BEGIN

"Restore object p to its original value (e.g., using log)";

$LL(p) \leftarrow$ false

END;

"Send message to other nodes and undo all steps in the same way"

VI. When non-local transaction T completes all its steps

"send to all nodes a completion message indicating that T has finished."

VII. When a node X receives a completion message for T

FOR "all p in this node such that $p \in LL\_SET(T)$" DO

BEGIN

$LL(p) \leftarrow$ 'false'

$LL\_SET(p) \leftarrow LL\_SET(p) - \{p\}$

END;

(End of Algorithm 1^*.)

To get a better understanding of how helpful it will be to use *algorithm* 2, instead of our original *algorithm* 1, let us informally try to estimate its time benefits by assuming the following average times:

Locking time (TL) for each step with algorithm 1 = 40 msec.

Locking time (TL) for each step with algorithm 1^* = 20 msec.

Computing time (TC) for each step with algorithm 1 or 1^* = 100 msec.

Note: These are the values used in our previous simulation. TC has its typical value (see section 3.2). The values for TL in algorithm 1 and algorithm 1^* are the extreme values used in our simulation when comparing the SK based algorithm to a 2PL based one. (Remember that algorithm 1 is our original SK algorithm, and note that algorithm 1^* is a 2PL based algorithm.) The purpose of taking such values for TL here, is to amplify the advantages of using algorithm 2.

Suppose now that half of the transactions submitted to our DBMS are going to have an empty compatibility set, and the remaining half a non empty one, i.e. one half of the transactions are going to be processed by *Algorithm* 1^* and the other by *Algorithm* 1. If such is the case then, not counting delays due to waiting for locked objects to become unlocked, each transaction step with algorithm 1^*

will take 120 msec. and with algorithm 1 it will take 140 msec. In average, then each transaction step will take 130 msec. with our new algorithm 2, which combines the processing advantages of the two above mentioned algorithms. This means an improvement of

$$\frac{140 - 130}{140} * 100\% = \frac{10}{140} * 100\% \approx 7.14\%$$

per step, over algorithm 1, our original algorithm, outlined in Appendix 1 of [Garc].

Estimating from our simulation statistics, locking and computing of steps' objects will take anywhere from a third to a half of all the needed time for the transaction execution. The remaining time will be utilized in waiting for locked objects and transmitting information among nodes of the distributed system. Therefore our 7.14% per step improvement of algorithm 2 over algorithm 1, in an environment where 50% of the transactions have an empty compatibility set, represents a 2.38% to a 3.57% total improvement in transactions' response time.

4.3. Second Variation

During normal processing, the requirement that interleaving of the steps of transactions take place only between those transactions in the same interleaving descriptor set has a significant influence on how long objects remain with global locks. If two transactions, T_1 and T_2, in the same interleaving descriptor, h, run together, and T_2 finishes before T_1 does, then the global locks on objects accessed by T_2 will have to remain set until T_1 finishes. Failure to do so can result in a third transaction, T_3, outside of h, accessing one free object, released by T_2, and therefore eventually causing an inconsistency. A precise scenario for such a case leading to a inconsistency has already been shown in section 2.3. The use of the

release sets (see below) was also explained in that section. The time duration of the global lock on an object directly depends of the size of the release set (REL) of the object: The global lock on an object cannot be released until that set is empty.

The release set of an object, which contains the identifiers of a series of compatible transactions being interleaved, could, on occasions, become very large, and delay a transaction outside the interleaving descriptor set for long period of time. If these events occur with a high frequency, they could eliminate the advantage of processing a transaction with our SK based mechanism. To avoid such scenarios, [Garc] suggests a modification in the rules of the interleaving descriptor sets. Our original SK algorithms are based on assumption 4.4 of [Garc]:

> "Let h be an interleaving descriptor set, and Y a transaction type. If $h \in cs(Y)$ and $h \neq \emptyset$ then $Y \in h$."

This means that if a transaction T can be interleaved with transactions of other types, then it can be interleaved with transactions of its same type. Later, in the same reference, Garcia-Molina suggests that assumption 4.4 be modified to hold only in the case of local transactions, restricting all non local types so that their interleaving descriptor sets will not contain its own transaction's type. Such modification is just a part of his complete suggestion requiring non local transactions to be interleaved just with local ones. The purpose of this idea is to allow the global locks of the objects accessed by interleaving transactions to be released as soon as the the nonlocal transaction being interleaved finishes. (Remember that local transactions have just one step, and steps are assumed to be atomic, therefore the release set of an object only needs to accumulate the identifiers of

nonlocal interleaving transactions.)

To measure the impact that the suggested modification could have on the performance of a DBMS using our SK based concurrency control mechanism we decided to rely again on simulation results. These results showed that restricting the interleaving descriptors in the suggested way does not pay off; enabling compatible transactions not to build large release sets did not compensate for the decrease in compatibility of nonlocal transactions, and therefore the original SK algorithms still performed better in most cases. This happened even in the case where the degree (percentage) of nonlocal transactions submitted to the system was very small. Such results confirm, as a byproduct, that the size of release sets does not hinder the performance of the original SK algorithms. For a full explanation of the results, including some illustrative graphs, the reader is refered to [Cord].

4.4 Conclusions

The algorithms in Appendix A are a first attempt to fully use the concept of SK in processing transactions in a distributed environment. Therefore, it is arguable if such algorithm construction is the best in terms of time and space complexity. The modification in Section 4.2 led to an immediate improvement, but the one in section 4.3 did not. It is still an open question for future research to propose an algorithm that fully uses all the advantages of SK, and that can be proven to have an optimal time and space complexity.

Chapter 5

FINDING COMPATIBLE TRANSACTIONS

5.1. Introduction

The previous simulation, based on a simple, two site distributed computing system, allowed us to compare the performance of two different concurrency control mechanisms: A general purpose (GP) one, based on two phase locking (2PL), and an application dependent (AD) one, based on semantic knowledge (SK) of the transactions. The results recommended, for certain system configurations the use of SK over 2PL.

If compatibility among transactions is going to play an important role in deciding if SK is worth using, then we ought to investigate the conditions under which transactions are compatible, and to be able to suggest aids that will facilitate the design of compatible transactions. To achieve the latter two purposes we assume that the data consistency constraints are known at transaction design time. Of utmost importance will be the cases when the different steps of the transaction are executed at different computing sites, since, as we know, the long transmission time between nodes will force database objects to be locked for extended periods, therefore blocking the access of non compatible transactions.

This chapter will be organized as follows: In section 5.2 we investigate the compatibility among different types of linear transactions. A linear transaction is a transaction whose actions are assignments of linear combinations of DB objects. Section 5.3 will study some properties of transactions executing set operations. Section 5.4 analyzes the compatibility among transactions where the decision on what value is written to the DB is based on the timestamp of the transaction.

Finally section 5.5 will present a short conclusion.

5.2. Constraints, Linear Constraints, and Linear Assignments

Consistency of the database is partly maintained by observing the consistency constraints (rules, restrictions) that are imposed by the database administrator (DBA). I.e., transactions have to make sure that they do not violate these constraints. These constraints are usually expressed in the form of equations or inequalities, and are the integrity restrictions observed by the real life processes that the transactions represent. The next example presents some common types of consistency constraints.

Example 5.2.1:

1) Money in a bank with four branches must be accounted for:

   $Bal_1+Bal_2+Bal_3+Bal_4=Tot$, where Bal_i = balance at branch "i", and Tot = total bank's money.

2) Suppose there is a company with three different working locations, L_i is the list of employees at each location, and L the complete list of employees. If we want every employee to appears in L and no employee to work at two different sites at the same time, then the constraints should be:

   a) $L_1 \bigcup L_2 \bigcup L_3 = L$; and

   b) $L_i \bigcap L_j = \emptyset$ for all $i,j \in \{1,2,3\}$ such that $i \neq j$.

3) A flight (plane's capacity =200) should not be overbooked:

   $\#RS \leq 200$, where #RS = number of reserved seats

4) Mechanical equations of a robot must always be observed.

   E.g.: Measured from its normal, resting state, the robot's arm

should not move more than 90^o in any direction.

5) The age of an employee in a company should not exceed 65 and his salary should be at least 10000 dollars:

$Emp\#.age \leq 65$

$Emp\#.salary \geq 10000$, where $Emp\#$ is the employees' identification number. ○

Definition 5.2.1 (Balancing action(s)):

The balancing action(s) of an action a_i of a transaction T_j are those actions that will balance the disparity in the consistency constraints created by action a_i, thereby enabling T_j to preserve consistency. ○

Example 5.2.2:

Let a,b and c be objects of the database and "$a + b = 2c$" be a consistency constraint. Consider the following transaction, T_1, whose steps are:

$$T_{11}\text{: a} \leftarrow \text{a} + \text{x; x} \in \mathbf{R}$$

$$T_{12}\text{: c} \leftarrow \text{c} + \frac{1}{2}\,\text{x}$$

Action T_{12} is the balancing action of T_{11} since it enables T_1 to balance the disparity of $a + b = c$ caused by action T_{11}. ○

Example 5.2.3:

Let U be the universal set of available elements in a database. Let A,B and C be subsets of U, and "$A \bigcup B = C$" be a consistency constraint. Let x $\in U$. Con-

sider now transaction T_1 with steps numbered on the left side:

1) A ← A - x

2) If x∉B then C ← C - x

Observe again that action T_{12} balances T_{11}. ○

Linear constraints are common in many databases, thus we will characterize them first.

Notation:

The objects of the database model we will be using in the theorems in this section will be denoted by oy_1, oy_2, $oy_{3,...}$, where y could be any letter, but will be usually a c for objects in consistency constraints and a for objects in assignment statements. The coefficient of these objects will be real numbers, unless otherwise specified.

Theorem 5.2.1:

If all constraints are linear combinations of database objects, i.e. the constraints are of the form $d_1oc_1 + d_2oc_2 + ...=0$, and all of the actions of two transactions T_1 and T_2 are of the form $on \leftarrow on + x$, where $x \in \mathbf{R}$, then T_1, T_2 are compatible. (Note: oi, besides being the representation of an object, will also represent its numerical value).

Proof:

Let T_1, T_2 be two transactions. Let $a_{11}, \ldots, a_{1n}$ be the actions of T_1, and $a_{21}, \ldots, a_{2m}$ those of T_2. Let $B_{11}, \ldots, B_{1n}$ and $B_{21}, \ldots, B_{2m}$ be the respec-

tive sets of balancing actions. Suppose now that for many actions those sets are included in different steps. (If an action and its balancing action(s) are in the same step, then, due to the atomicity of the step, no inconsistency problem will be caused by that action.) By the statement of the theorem we know that each action must be of the form $oc_i \leftarrow oc_i + x_i$, where $x_i \in \mathbf{R}$ and therefore the set of balancing actions is $\{oc_{i_k} \leftarrow oc_{i_k} + x_{i_k} |\ k{=}1 \text{ to } n_i\}$, such that

$$d_i x_i + \sum_{k=1}^{n_i} d_{i_k} x_{i_k} = 0 \quad \text{(Eq.1)}.$$

Due to the commutative and associative properties of the group $(\mathbf{R}, +)$ it does not matter what order we add all the $d_{i_k} x_{i_k}$ products in an algebraic expression like equation 1. The result will always be the same; the sums will always add to zero, and therefore $d_1 oc_1 + d_2 oc_2 + \ldots = 0$ will always be true.

We can now conclude that no matter how we interleave the steps of such transactions, the consistency will always hold, which implies that T_1, T_2 are compatible. ○

We shall now expand on the property of the previous theorem. The property allowed us to declare, that two transactions, obeying concurrency constraints of the form mentioned in the theorem are compatible. Prior to carrying out that purpose, let us define a simple concept for a better understanding of the proof.

Definition 5.2.2 (action tail):

Given an action that is a linear assignment, we call the *tail* of the action, that part of the right hand side of the assignment statement that does not include the assigned object. (E.g.: in $oj \leftarrow oj + x + y + z$, the action tail is $x + y + z$). For notational purposes we will call the tail of the i^{th} action of tran-

saction T_k, the *ki_tail.* ○

Theorem 5.2.2:

Let all constraints be linear combinations of database objects (i.e., $d_1 oc_1 + d_2 oc_2 + \ldots = 0$, where $d_1, d_2, \ldots \in \mathbf{R}$). If transactions are a series of assignments of the form $oc_n \leftarrow oc_n + f_1 oa_1 + \ldots + f_l oa_l$ $(f_1, f_2, \ldots \in \mathbf{R})$), where $oa_k \neq oc_j$ for all objects oa_k in assignment statements and oc_j in consistency constraints, then the transactions are compatible.

Proof:

Let

$$d_1 oc_1 + d_2 oc_2 + \ldots = 0 \quad \text{(Equation 1)}$$

be a consistency constraint. Without loss of generality we will assume here that each action has exactly one balancing action. (The extension of this proof for transactions with actions having more than one balancing action is immediate and straightforward.) Let oc_n, oc_m, oc_s be part of the consistency constraint in equation 1. Consider now the following two transactions (Note that we will be treating here only the problems provoked by object oc_n. Problems of other objects, due to the nature of the transactions are resolved in the same way):

T_1:

.

.

.

$v) oc_n \leftarrow oc_n + f_1 oa_1 + \ldots + f_k oa_k$

.

.

.

$w) oc_m \leftarrow oc_m - (d_n/d_m)\Big[f_1 oa_1 + ... + f_k oa_k\Big]$

.

.

.

T_2:

.

.

.

$x) oc_n \leftarrow oc_n + f_{k+1} oa_{k+1} + ...+ f_{k+l} oa_{k+l}$

.

.

.

$y) oc_s \leftarrow oc_s - (d_n/d_s)\Big[f_{k+1} oa_{k+1} + ...+ f_{k+l} oa_{k+l}\Big]$

.

.

.

Note that action (1w) is the balancing action of (1v), and (2y) the balancing action of (2x), and that the oa_i objects need not all be different.

Since (**R**, +) is an abelian group (i.e, "+" is commutative and associative), then no matter in what order we add $d_n(1v\_tail)$, $d_m(1w\_tail)$, $d_n(2x\_tail)$, and $d_s(2y\_tail)$, the sum will always be zero. In this case, since oc_n, oc_m, and oc_s are not multiplied by any coefficient, it is immediate to see that the balance con-

sistency constraint, equation 1, will be satisfied. Therefore we can say that steps containing actions 1v, 1w, 2x, and 2y can be interleaved in any way without violating the consistency constraint, which implies that T_1, T_2 are compatible. ○

The previous characterization of two compatible transactions was somewhat too restrictive, in the sense that only the assigned object in the assignment could be an object of the consistency constraints. We can relax that condition in some ways, if we take the necessary precautions.

Theorem 5.2.3:

Given the same conditions as in theorem 5.2.2, with the difference that an oa_k object can equal some oc_j. This means that an object in the linear assignment aside from the assigned one, can also be part of the linear constraint). If that action and its set of balancing actions are strictly grouped (i.e., just those actions, no others) into the same step then the two transactions T_1 and T_2 are compatible.

Proof:

If an action and its set of balancing actions are strictly grouped together into the same step, then it is obvious that at the end of the step, since steps are atomic units, that it will cause no disparity in the consistency constraint equation. The remaining problems that these kind of transactions can present are the same as the ones in the previously expressed theorem 5.2.2, therefore, T_1 and T_2 are compatible. ○

Example 5.2.4:

Let "$a + b + c = d + e$" be a consistency constraint, and consider the following transactions:

T_1:

1) a ← a + 2c

2) d ← d + 2c

T_2:

1) c ← c + e

2) d ← d + e

Consider now the following schedule, S: T_{11}, T_{21}, T_{22}, T_{12}. After the execution of S we can find the that a + b + c = d - e, which clearly violates the restriction imposed by the above consistency constraint. We should note the constraint violation could have been avoided if we had followed the conditions of the previous theorem and executed actions 1 and 2 of T_1 in one single step. ○

Another way of relaxing the conditions of Theorem 5.2.2 is now apparent, but we shall again have to take the right precautions.

Theorem 5.2.4:

Given the conditions of Theorem 5.2.2, with the only difference that an oa_k object can equal some oc_j (i.e., we can have an object, aside from the assigned one, take part in the linear constraint). Suppose then that one of the actions in T_1 is $oc_n \leftarrow oc_n + \cdots + oc_j + \cdots$, where oc_j is part of a consistency constraint, and that T_2 has an action of the form $oc_j \leftarrow oc_j +$ ..... If in every such case, the oc_j - action does not come in between the on - action and its set of

balancing actions, then both transactions T_1 and T_2 are compatible.

Proof:

Since no actions of T_2 will come between the action $oc_n \leftarrow oc_n + ... + d_j oc_j + \cdots$ and its set of balancing actions, there exists no possibility that the oc_n - action can cause any disparity in the consistency constraint. We have then reduced the problems of this formulation to those of Theorem 5.2.2, and therefore, T_1 and T_2 are compatible. ○

Observation 5.2.1:

The provision needed for compatibility of T_1 and T_2 in theorem 5.2.4 can easily be implemented by setting an *indicator*, a special type of temporary lock, for the problem object (oc_j in the formulation of theorem 5.2.4). This indicator will force any action of T_2 causing a new assignment to oc_j, to delay execution until the indicator has been unset. This unsetting will occur at the end of the last balancing action of *on*.

Even though this indicator setting implies some sort of locking, we should point out that some differences do exist. Normal locking is usually done by the concurrency control manager, whereas the setting and unsetting of the indicators is done by the transaction itself, which therefore relieves the system of having to take care of such locks. This is a real advantage, since not all transactions need that mechanism. On the other hand, this type of lock remains only for a specified time (from the beginning of an action to the end of its last balancing action), thereby permitting free access to the object by compatible transactions at all other times. A system managed lock would have forced the object to remain locked until the end of the transaction, as is the case in two phase locking. ○

Observation 5.2.2:

Note in example 5.2.4 that if no action of T_2 had come between action $a \leftarrow a + 2c$, and its balancing action $d \leftarrow d + 2c$, assigning a new value to object c (part of the consistency constraint), as theorem 5.2.4 suggests, then no inconsistency would have been caused. ○

Corollary 5.2.1:

Given the same conditions of theorem 5.2.2, allow T_1 to have actions of the form $oc_n \leftarrow r(oc_n) + b_1 oa_1 + b_2 oa_2 + \cdots$, where $r \neq 0$. (I.e., we are for the first time permitting the assigned object to get a new value, that includes a multiple of its own value.) If this action and its set of balancing actions are strictly grouped together, or if no action of the second transaction, T_2, come in between such an action and its set of balancing actions, then both transactions are compatible.

Proof:

Since we can write $oc_n \leftarrow r(oc_n) + \cdots$ as $oc_n \leftarrow oc_n + (r-1)oc_n + \ldots.$, it is easy to see that applying the principles of theorem 5.2.3 or of theorem 5.2.4 respectively, will show the compatibility of T_1 and T_2. ○

Example 5.2.5:

Let "$a + b = c$" be a consistency constraint, and consider the following two transactions:

T_1:

1) a ← 3a + x

2) $c \leftarrow c + 2a + x$

T_2:

1) $a \leftarrow a + y$

2) $c \leftarrow c + y$

Suppose now that we run the following schedule: T_{11}, T_{21}, T_{12}, T_{22}, then we will end up with $a + 2y + b = c$. If $y \neq 0$ consistency will not hold. If we had followed the conditions in the previous corollary 5.2.1 this would not have happened. Thus, we have shown that disobeying corollary 5.2.1 will not guarantee compatibility of transactions like those formulated in the corollary. ○

5.3. Set Operations and Set Constraints

Database applications where transactions perform set operations and where constraints are set based abound in real applications. This section will present a simple way of checking if two such transactions are compatible, and will give suggestions on how to make them compatible if they are not. The method for checking compatibility is based on the propositional calculus and it is readily implemented in a computer, although in worst cases it can be exponentially time complex.

We know that every set equality or inequality can uniquely be represented by a boolean expression. The validity of such equality or inequality can be easily checked by evaluating the corresponding boolean expression.

Notation and conventions:

1) Given a set A such that $A = \{x \in U \mid a(x)\}$, we will call U the universal set, i.e., the set from where the elements of our database are drawn. a is called the predicate or condition defining the set, and $a(x)$ is true iff $x \in A$. Generally sets will be denoted with upper case letters and the predicate or condition defining the set will be denoted with the respective lower case letter.

2) "Transaction T access a consistency constraint C" means that at least one of the variables in C will be modified by T.

Example 5.3.1:

Let U be the universal set and $L_i \subset U$ for all i=1...4, where $L_i = \{x \in U \mid l_i(x) is\ true\}$. I.e., set L_i comprises all those elements of our universal set U such that the condition l_i applied to x is true. E.g.: If U={persons on earth} then $l_i(x)$ could mean x is a woman, or x lives in the US, or x is older than 30 years,....

Let us consider the following set equality: $L_1 \bigcap L_2 = L_3 - L_4$. There will only be a finite number of instances of membership of an element in the different sets that make this equality hold. To find such instances one can evaluate the respective boolean expression c: $l_1 \wedge l_2 \Leftrightarrow l_3 \wedge \neg l_4$, and check which combinations of true(1) (element in set), false(0) (element not in set) values satisfy c. In our special case such combinations are:

l_1	l_2	l_3	l_4
1	1	1	0
1	0	1	1
1	0	0	1
1	0	0	0
0	1	1	1
0	1	0	1
0	1	0	0
0	0	1	1
0	0	0	1
0	0	0	0

With this table in hand we can see, for example (row 1 in table), that when an element is in L_1, L_2, and L_3, but not in L_4, then our original equality is fulfilled. The other rows show all the other valid cases of the equation. ○

In the same way as the previous example, if we have some set based consistency constraints, these can be represented with boolean expressions. At the end of the execution of a schedule we can test the validity of the boolean expressions representing the consistency constraints related to the transactions involved in the schedule. These boolean expressions have to be evaluated for each member of a set, which was accessed during the course of the schedule. If all evaluations of every boolean expression be true, we can say that no consistency constraint was violated, and therefore that the consistency was preserved.

Following the ideas of the previous paragraphs, we now find it easy to test if two transactions T_1 and T_2, doing set operations, are compatible. After the execution of a schedule that interleaves the steps of the two transactions, one or more elements (members) of our universal set will have been accessed (inserted, deleted, transferred, etc.). Since transactions are consistency preserving then we will focus our attention on those elements that were accessed by both transactions. (If an element, say x, was accessed just by one transaction, then due to the consistency preserving property of the of the transaction, we know that x will surely not disrupt the database consistency.) For each element accessed by both transactions, a true(1) or false(0) value can be assigned to the predicate (condition) defining each set that is part of at least one consistency constraint, depending if the element is or is not in the set. The consistency constraints accessed by both transactions could or could not have been violated after the execution of the schedule that interleaved the steps of T_1 and T_2. To test that simply plug the respective 0,1 values of each "potential trouble" element into each boolean expression, representing a consistency constraint, and evaluate them. If these tests turn out to be true, for every permitted interleaving of T_1 and T_2, i.e., that satisfy each boolean expression representing a consistency constraint, then we can say that the two transactions are compatible.

Example 5.3.2:

Let U be our universal set from which the elements of our database are drawn, and L_1,...., $L_6 \subset U$, such that $L_i = \{x \in U \mid l_i(x)\}$ for all i=1...6. (Note that in database operations $l_i(x)$ true if and only if $x \in L_i$.) Let C_1 and C_2 be consistency constraints, and c_1 and c_2 be the respective boolean expressions:

$$C_1\text{: } L_1 \bigcap L_2 = L_3$$

$$C_2\text{: } L_4 \bigcup L_5 = L_6$$

$$c_1\text{: } l_1 \bigwedge l_2 \Leftrightarrow l_3$$

$$c_2\text{: } l_4 \bigvee l_5 \Leftrightarrow l_6$$

Consider the following transactions (numbers refer to steps) T_1, T_2 and T_3, and schedules S_1, and S_2 ($x \in U$):

$T_1(x)$:

1) $L_1 \leftarrow L_1 - \{x\}$

2) $L_3 \leftarrow L_3 - \{x\}$

$T_2(x)$:

1) $L_1 \leftarrow L_1 \bigcup \{x\}$

2) $L_2 \leftarrow L_2 \bigcup \{x\}$

3) $L_3 \leftarrow L_3 \bigcup \{x\}$

$T_3(x)$

1) $L_4 \leftarrow L_4 \bigcup \{x\}$

2) $L_6 \leftarrow L_6 \bigcup \{x\}$

S_1: T_{11}, T_{31}, T_{12}, T_{32}

S_2: T_{21}, T_{11}, T_{22}, T_{12}, T_{23}

Suppose now that initially element x is in every set, i.e., $l_i(x)=1$ for all $i=1...6$. We will consider here, for simplicity, just this one initial database state.

Fortunately, as we will learn later, for transactions of the type in this example, it suffices to use any initial consistent database state to prove compatibility of two transactions. Let us then see what happens after running the two different schedules.

1) After S_1 is run the condition of each set evaluates to: $l_1(x) = l_3(x) = 0$ and $l_2(x) = l_4(x) = l_5(x) = l_6(x) = 1$. This combination of 1,0 assignments satisfies both c_1 and c_2, which in turn implies that constraints C_1 and C_2 were not violated. As a matter of fact, since T_1 and T_3 access completely different sets then we can see that no matter how we interleave the two transactions, we are always going to obtain the same result, i.e., consistency will always be maintained and therefore we can conclude that T_1 and T_3 are compatible.

2) Given the same initial setting of our database, we will analyze now the result of running S_2. At its end we will have $l_1(x)=0$, and $l_j(x)=1$ for all other conditions. Since $l_1(x)=0$ and $l_2(x) = l_3(x) = 1$ do not satisfy c_1, we know that consistency constraint C_1 has been violated. In other words, T_1 and T_2 should not be interleaved as in S_2 if we want to maintain database consistency. We then conclude that T_1 and T_2 are not compatible. ○

Observation 5.3.1:

The past two examples were composed solely of transactions whose steps were of the form $L \leftarrow L \bigcup \{x\}$ or $L \leftarrow L - \{x\}$. We would expect insertions and deletions to be very common set operations in a real DBMS. There are, of course, plenty of cases when a step is comprised of operations that involve more than one element on both operands. For the moment let us consider the analysis

of only insertions and deletions. ○

In example 5.3.2 we talked about an initial setting of the database. In that special case $x \in L_i$ for all i=1...6, i.e., for all i, $l_i(x) = 1$. Fortunately, we will see later in this section, that testing the compatibility of transactions of the type in example 5.3.2 is independent of the initial 0,1 values of the set's conditions, as long as these initial values satisfy the given consistency constraints. There are, however, cases where that is not true, as the following example will show. Therefore, it is important to take into account, when designing a general algorithm to test compatibility of transactions, the different initial values of the set's conditions.

Example 5.3.3:

Let $L_i = \{x \in U \mid l_i(x)\}$ for i=1,2, where U is our universal set of elements of the database. Let "$L_1 = L_2$" be our sole consistency constraint. The respective boolean expression is "$l_1(x) = l_2(x)$", and therefore it will be satisfied only if both set conditions have the same value. Let us consider the following two transactions, where the numbers on the left side correspond to steps:

$T_1(x)$:

1) $Temp \leftarrow (L_1 \bigcup \{x\}) - L_1$

$L_1 \leftarrow L_1 \bigcup Temp$

$L_2 \leftarrow L_2 \bigcup Temp$

$T_2(x)$:

1) $L_2 \leftarrow L_2 - \{x\}$

2) $L_1 \leftarrow L_1 - \{x\}$

The only possible way to interleave the two transactions in a non serializable fashion is by running the following schedule: T_{21}, T_{11}, T_{22}. Let us see then what happens when we run this schedule starting with the two different valid initial conditions.

a) $l_1(x) = l_2(x) = 0$. Since in this case $x \notin L_1$ then $Temp = \{x\}$ and transaction T_1 will add x to both sets. We will end up with $l_1(x) = 0$, but $l_2(x) = 1$, therefore violating the database consistency.

b) $l_1(x) = l_2(x) = 1$. Here $x \in L_1$ implies that $Temp = \emptyset$ and therefore x is not added to any set. Our end result is $l_1(x) = l_2(x) = 0$, which holds the database consistency.

We can now see that the consistency of our data at the end of the same schedule depended on what initial valid combination of 0,1 values we started with. As this was the only possible way of interleaving the two transactions we can then conclude, that in this case, determining the compatibility of the transactions is dependent on the initial values of the set conditions. ○

The intuition behind our idea, on how to prove that two transactions doing set operations are compatible, should by now be clear, and therefore we think it is proper now to outline our desired algorithm. Before doing that, we want the reader to take note of the following two observations.

Observation 5.3.2:

Often transactions include conditional statements. The value of the condition inside the statement will determine the execution path of the transaction. Since the actions to be performed by a transaction depend on the execution path, it will be necessary in our next algorithm to consider all the possible execution paths that both transactions could take. ○

Observation 5.3.3:

It is interesting and important to notice that by using boolean expressions to test preservation of set based consistency constraints, we are not introducing a different type of structure. We are just introducing a new representation of set equalities or inequalities that is easier to operate and evaluate than the equalities themselves. This statements applies to computer implementations as well. ○

Algorithm 5.3.1:

Given: a) Two transactions T_1 and T_2; and

b) N set based consistency constraints, $C_1, \cdots, C_N$, such that T_1 and T_2 access at least one set in each of the constraints. ‡

Result: Determine if T_1 and T_2 are compatible.

1) For each of the N consistency constraints let the respective boolean expressions be $c_1, \cdots, c_N$, and let $c^* = c_1 \wedge \ldots \wedge c_N$. (Note that satisfying c^* is equivalent to satisfying all of the c_i's)

‡Note that if there is no common consistency constraint being accessed by both transactions, i.e., if $N=0$, then T_1 and T_2 are automatically compatible.

2) Let PI_{12}={all possible interleavings of the steps of T_1 and T_2}.

3) Let VALID={initial 0,1 combinations that satisfy c^*}

4) **Begin**(*Test*)

For each element x_i accessed by both transactions **do**

For each initial combination in VALID **do**

For each schedule of T_1 and T_2 in PI_{12} **do**

For each different execution path of T_1 and T_2 **do**

Evaluate $c^*(x)$ (* are all constraints satisfied? *)

If c^* is not satisfied **then** T_1, T_2 *are not compatible*

STOP

Endif

Endfor

Endfor

Endfor

Endfor

T_1, T_2 *are compatible* (* c^*'s value was always true *)

End(*Test*) ○

Observation 5.3.4:

It is important to understand the significance of the statement of the first **For** loop in the previous algorithm:

For each element x_i accessed by both transactions **do**.

The input to a transaction will often be an n-tuple (n $>$ 1) of elements belonging to our universal set U, and not just one as in examples 5.3.1 and 5.3.2. In these cases any element in the n-tuple input to T_1 could match any element in the n-tuple input to T_2 and therefore it is necessary when running algorithm 5.3.1 to consider all those possibilities. (E.g.: If we have $T_1(x_1,x_2)$ and $T_2(y_1,y_2,y_3)$, then x_1

could be the same element as y_1, y_2 or y_3. The same goes for x_2. We have then here six different possibilities to consider.) ○

It is desirable to establish the complexity of any algorithm. It is particularly desirable for algorithm 5.3.1, since we propose to show later that for certain types of transactions its complexity can be reduced.

Theorem 5.3.1:

The time complexity of algorithm 5.3.1 is exponential in the size of the transactions, the number of different sets involved in the constraints accessed by T_1 and T_2, and the number of conditional statements of the transactions.

Proof:

The complexity of algorithm 5.3.1 depends on the maximum number of times that the inner **For** loop statement must be executed. (The time for the statement itself can easily be bounded by a real number, say M.) Such a number will be the product of the number of repetitions of each of the four different **For** loops. Assuming that r elements will be accessed by both transactions, then exactly r repetitions will be performed be the outermost loop. (Usually r will be assumed to be very small, and therefore negligible.) The second loop will be executed for as many different valid 0,1 combinations as there are in set VALID. Let *NDS* be the number of different sets involved in the consistency constraints, then there will be at the most 2^{NDS} different combinations. As for the third loop, its number of repetitions depends on the number of permitted interleavings that there are in PI_{12}. In the worst case, when the transactions are compatible, the number of iterations of the third loop will be equal to the size of PI_{12}. Let us

then calculate the cardinality of PI_{12}.

Let n_1 be the number of steps in T_1, and n_2 the number in T_2. It is not too difficult to see that the number of different permitted interleavings is equal to choosing n_1 (or n_2) steps out of $n_1 + n_2$. Since steps have to be ordered, then such a number is

$$\left(\begin{matrix} n_1 + n_2 \\ n_1 \end{matrix} \right) = \left(\begin{matrix} n_1 + n_2 \\ n_2 \end{matrix} \right).$$

This expression is maximal when

$$n_1 = \frac{1}{2} \lceil n_1 + n_2 \rceil,$$

i.e., when $n_1 = n_2 = n$. This being the case, and using Stirling's approximation ($n! \approx \sqrt{2\pi n} \left(\frac{n}{e} \right)^n$), the cardinality of PI_{12} is

$$\#(PI_{12}) = \left(\begin{matrix} n_1 + n_2 \\ n_1 \end{matrix} \right) = \left(\begin{matrix} 2n \\ n \end{matrix} \right)$$

$$= \frac{(2n)!}{n! * (2n - n)!} = \frac{(2n)!}{n! * n!}$$

$$\approx \frac{\left(\frac{2n}{e} \right)^{2n}}{\left(\frac{n}{e} \right)^n * \left(\frac{n}{e} \right)^n}$$

$$= 2^{2n} = 2^{(n_1 + n_2)}.$$

We are now missing only the number of paths that can be taken by each transaction. This will give us the total number of repetitions by the innermost loop. Each conditional statement allows two different directions for the

transaction's execution flow. Assuming that T_1 and T_2 have t_1 and t_2 conditional statements respectively, then each transaction will have 2^{t_1} and 2^{t_2} different execution paths. This brings the total number of repetitions of the innermost loop to

$$2^{t_1} * 2^{t_2} = 2^{t_1 + t_2}.$$

The complexity of the algorithm is then, in the worst case,

$$O(r * 2^{NDS} * 2^{(n_1 + n_2)} * 2^{(t_1 + t_2)}) = O(r * 2^{NDS + (n_1 + n_2) + (t_1 + t_2)}) \qquad \bigcirc$$

The reader should observe, that although algorithm 5.3.1 is easy to understand, it may not be practical to implement because of its exponential time complexity. However, taking the following two aspects into account, its implementation might be worth considering.

1) This is the first attempt at giving the DBMS the tools for the self determination of two transactions compatibility, relieving therefore the DBA of this burden.

2) This algorithm will have to be executed for any two transactions just once, and if the time complexity poses a problem in the implementation of the algorithm, we can drastically cut its complexity by restricting the type of the transactions that it will take as input. Two examples of such types of transactions, along with proofs of their reduced complexity follow.

Definition 5.3.1 (NC Transactions):

A transaction of type NC (Not Conditional) is a transaction where all the actions are simple insert or delete assignment statements, and the value that an action assigns is always independent of the database state. (E.g.: The transactions in examples 5.3.1 and 5.3.2 are of this type; transaction T_1 in example 5.3.3

is not.) ○

Theorem 5.3.2:

Testing the compatibility of two transactions T_1 and T_2 of type NC, in the previous algorithm 5.3.1, is independent of the initial assignments of 1's and 0's satisfying $c^* = c_1 \wedge .... \wedge c_N$, where c_i is the boolean expression representing the consistency constraint C_i for i=1,...,N.

Proof:

Suppose by contradiction that such a dependency exists. If so, there must exist two different initial conditions (1,0 assignments) IC_1 and IC_2, such that, given a valid interleaving (schedule) of transactions T_1 and T_2, say I, then executing it with IC_1 as initial condition does not violate consistency, but executing it with IC_2 initially does violate consistency.

Given the scenario of the previous paragraph, there must exist at least one c_i, without loss of generality say exactly one, such that its initial value determines the outcome of c^* when running schedule I with IC_1 and IC_2 as initial assignments of 1's and 0's satisfying c^*. Let c_j, representing constraint C_j, be the problematic boolean expression. Then the initial value of one of the predicates in c_j, must determine our result. Let this predicate be c_{jk}, representing the set L_{jk}. This implies that the membership of an element, say x, in set L_{jk} has an influence in the violation or non-violation of consistency constraint C_j. (E.g.: If at the beginning of the schedule execution $x \in L_{jk}$ then we have initial condition IC_1 and consistency will hold at the end of I's execution. Else we have IC_2 and consistency will not be maintained). We can infer from the last statement that the

status of the membership of x in L_{jk} is not altered when running I. Considering that T_1 and T_2 are NC transactions, i.e., their execution does not depend on the database state, and therefore neither does I's execution, we can conclude that if schedule I is to maintain consistency at the end of its execution, then it is missing an action, a simple delete or insert (remember this characteristic of NC transactions), that properly alters the status of the membership of x in L_{jk}. This implies that a balancing action, with the responsibility for this set is missing. If this is the case then either T_1 and/or T_2 is missing such a balancing action, and therefore one or both of the transactions are missing such balancing action, i.e, one of the transactions, or both, is (are) inconsistent. But this last is certainly a contradiction, since we were assuming that T_1 and T_2 are transactions in the full sense of the transaction definition, i.e., they are consistency preserving units.

Due to the contradiction arrived in the previous paragraph, we can conclude that the assumed dependency on initial conditions does not exist. *q.e.d.* ○

Corollary 5.3.1:

Algorithm 5.3.1 can be modified for NC transactions and its time complexity reduced to $O(r * 2^{(n_1 + n_2)})$ (variables are as defined in theorem 5.3.2).

Proof:

The independence of NC transactions on the initial conditions when testing for compatibility allows us to remove from algorithm 5.3.1 the second and fourth **For** loops. As we know, these loops would have been executed, in the worst case, $O(2^{NDS})$ and $O(2^{(t_1 + t_2)})$ times. Removing these loops leaves us with a $O(r * 2^{(n_1 + n_2)})$ time complexity for the algorithm. ○

In DBMSs where transactions performing set operations play an important role, transactions of the type NC can account for a large percentage of the daily load. The previous reduction in time complexity of algorithm 5.3.1 for NC transactions is an important result and we think that for such systems its implementation will pay off.

Example 5.3.4:

Let U be the universal set from which the elements of our database are drawn. Let L_1, L_2, $L_3 \subset U$ such that for all i=1,2,3 $L_i=\{x \in U \mid l_i(x)$ *is true*$\}$. The following consistency constraint, C_1, relates these sets "$L_1 \bigcap L_2=L_3$". Suppose now that we define the following transaction types (the numbers denote the steps):

$T_1(x)$: Add element x to L_3.

1) $L_1 \leftarrow L_1 \bigcup \{x\}$

2) $L_2 \leftarrow L_2 \bigcup \{x\}$

3) $L_3 \leftarrow L_3 \bigcup \{x\}$

$T_2(x)$: Delete element x from L_1.

1) $L_1 \leftarrow L_1 - \{x\}$

2) $L_3 \leftarrow L_3 - \{x\}$

$T_3(x)$: Delete element x from L_2.

1) $L_2 \leftarrow L_2 - \{x\}$

2) $L_3 \leftarrow L_3 - \{x\}$

We want to know now which transaction types are compatible. Following the algorithm we have to first produce the respective boolean expression for consistency constraint C_1. This is $c_1 = (l_1 \wedge l_2) \Leftrightarrow l_3$. Therefore $c^* = c_1$. The true (1), false (0) values that satisfy c^* are the following:

l_1	l_2	l_3
0	0	0
0	1	0
1	0	0
1	1	1

We are first going to test if T_1 and T_2 are compatible types. Since if $x \neq y$ and running $T_1(x)$ and $T_2(y)$ interleaved in any way will always produce consistent results as they do not interfere in their actions, then we will concentrate in both transactions on the same element, say x. (Note that this is always the case in transactions that deal with set operations.)

For simplicity we will agree from now on that in the context of schedules the number xy denotes the action T_{xy}, i.e., the the y^{th} action of transaction T_x. Our first schedule is S_1: 21, 11, 22, 12, 13. This produces $l_1(x) = l_2(x) = l_3(x) = 1$, that satisfies c^*. We have to continue to our next schedule S_2: 11, 21, 22, 12, 13. This last schedule produces $l_1(x) = 0$ and $l_2(x) = l_3(x) = 1$. This assignment certainly does not satisfy c^* and therefore T_1 and T_2 are not compatible.

In a similar way we can show that T_1 and T_3 are not compatible.

For the case of T_2 and T_3, every permitted interleaving of the two transactions will produce a 0 value for each $l_i(x)$ (i=1,2,3). Such an assignment satisfies

c^* and therefore these two types of transactions are compatible.

With equal ease it can be proved that a transaction of type T_1 is compatible with itself. The same case occurs for T_2 transactions and T_3 transactions.

Our conclusion is then that, $cs(T_1)=\{\{T_1\}\}$, and $cs(T_2)=cs(T_3)=\{\{T_2, T_3\}\}$. ○

Definition 5.3.2 (IF_1 transactions):

An IF_1 transaction is a transaction that includes at least one statement of the form

If $x \in A$ **then**

S_1

Else

S_2

Endif ;

where set A is part of a consistency constraint, and S_1 and S_2 are assignment statements of the sort that appear in NC transactions. ○

Note: We have decided to call the previous type of transactions IF_1, since there are other types of conditional transactions. Those other types will not be considered in this thesis.

Notation:

Let $L_1, \ldots, L_n$ be subsets of our universal set U such that $L_i = \{x \in U \,/\, l_i(x)\}$. Let $V = \{(l_1(x),..., l_n(x))\}$ be a set of possible initial conditions, and $M = \{(l_{j_1}(x),..., l_{j_r}(x)) \,/\, r < n\}$. We will say that a member

$m = (l_{j_1}(x),..., l_{j_r}(x))$ of M is "*included*" in a member $v = (l_1(x),..., l_n(x))$ of V iff m is the projection of v on coordinates $j_1, \ldots, j_r$. E.g.: Suppose we have sets A, B, C and $V = \{(a(x), b(x), c(x))\} = \{(0,0,0), (0,1,1)\}$, and $M = \{(a(x), b(x))\} = \{(0,0), (0,1), (1,1)\}$ then only (0,0) and (0,1) are *included* in members of V. Note that there is no 3-tuple of V that can *include* the 2-tuple (1,1) of M. ○

Corollary 5.3.2:

1) Let T_1 and T_2 be two transactions of type IF_1

2) Let $X=\{x_1, \ldots, x_r\}$ be the set of common elements accessed by T_1 and T_2, and $\{x_1, \ldots, x_m\}$ $(m \leq r)$ be the set of elements that participate in "If" statement's conditions (If $x_i \in A$ then ....).

3) Let x_j $(1 \leq j \leq m)$ be the element that participates the most in "If" statement's conditions, and let α be the number of times it participates.

Algorithm 5.3.1 can be modified to test compatibility of IF_1 transactions, and its complexity reduced to

$$O(r * 2^{(n_1 + n_2) + \alpha})$$

Proof:

IF_1 transactions are NC transactions with the only difference that the insertion of the "If" statements present alternatives in the flow of the transaction's processing, depending on the value of the conditions to evaluate. Such values are based on whether an x_i, $(1 \leq i \leq m)$, is a member of a set or not. Since independent of the processing flow, an IF_1 transaction, as well as any other transaction, will maintain consistency, then testing compatibility of two IF_1 transactions is

equal to testing compatibility of two *NC* transactions with the only difference that we now have to consider all the possible flow routes of each transaction. Each flow route can be thought of as a combination of 0,1 values representing the predicates of the different sets involved in the "If" statements. (E.g.: Suppose we have the statement "**If** $x \in A$ **then**....". If $x \in A$ then $a(x)$=1 and the flow takes the **then** path; if $x \notin A$ then $a(x)$=0 and flow is directed to the **else** path, if there is such an alternative.)

Let $R(x_i)$ $(1 \leq i \leq m)$ be the set of the 0,1 combinations giving the different flow routes determined by element x_i. (E.g.: If x_i participates in 2 "If" statements with sets A and B, then $R(x_i) = \{(0,0), (0,1), (1,0), (1,1)\}$.) The previous considerations amount to having to test for compatibility not just for any one valid initial condition, as theorem 5.3.2 proved for NC transactions, but for for all the different initial conditions that represent the different permissible flow routes as determined by the participations of x_i in "If" statements. What flow routes are permissible depends on the initial valid combinations of zeros and ones in set VALID. Therefore the valid flow routes for x_i will be the combinations of zeros and ones that can be included in at least one combination of VALID. Other flow routes should not be permissible, since then we would be testing compatibility with inconsistent initial conditions. Such considerations about flow routes forces us to use a modified set VALID in algorithm 5.3.1, that we will call MOD_VALID(x_i). For each x_i, it will be the smallest set containing the valid initial combinations of zeros and ones that can include each of the flow routes as given by set $R(x_i)$. In the case that $i > m$, i.e., when x_i does not participate in any "If" statement condition, as stated in hypothesis # 2, then the flow of the transactions will be fixed, i.e., transactions T_1 and T_2 will behave on input x_i as if

they were NC transactions, and MOD_VALID(x_i) will contain just any one valid initial combination of zeros and ones. Algorithm 5.3.1 can now be modified:

change VALID for MOD_VALID(x_i) in the second **For** loop.

Since x_j, the element that participates the most in "If" statement conditions, does so for a number α of times, then obviously $|R(x_i)| \leq 2^{\alpha}$ for all $x_i \in \{x_1, \ldots, x_r\}$. Therefore the second loop, whose number of repetitions depends on the cardinality of set MOD_VALID(x_i) (calculated with the help of sets VALID and $R(x_i)$ for all i), will be executed at the most 2^{α} times. Clearly $2^{\alpha} \leq 2^{NDS}$ since $\alpha \leq NDS$. The time complexity has now been reduced to

$$O(2^{\alpha} * r * 2^{(n_1 + n_2) + (t_1 + t_2)}) \ .$$

Considering that no conditional statements, but just the class mentioned in definition 5.3.2 are allowed to take part in IF_1 transactions, then the complexity reduces further by the factor $2^{(t_1 + t_2)}$, to

$$O(2^{\alpha} * r * 2^{(n_1 + n_2)})$$

$$= O(r * 2^{(n_1 + n_2) + \alpha}) \qquad \bigcirc$$

The next example allows the reader to get more acquainted with the mechanism of the algorithm, and allows us to show and discuss how some special type of aids can help in achieving more compatibility among transactions.

Example 5.3.5:

Suppose that a company has factories at two different locations B_1 and B_2. The list of employees in site B_i is L_i (i=1,2). A separate list, L, of the employees at both factories is kept at the company headquarters, and therefore our first consistency constraint, C_1, is $L = L_1 \bigcup L_2$. The company's directors do not want an

employee to work at the two different places concurrently, and here we have our second consistency constraint (C_2), $L_1 \bigcap L_2 = \emptyset$

Since the company is normally in the process of firing old employees, of hiring new ones, and of transferring one employee from one working site to another, three main transaction types will be defined to operate on the company's distributed database (numbers correspond to steps):

a) Del(L_i, E): Delete employee E from list L_i. Steps are:

1) $L_1 \leftarrow L_1 - \{E\}$

2) $L_2 \leftarrow L_2 - \{E\}$

3) $L \leftarrow L - \{E\}$

b) Add(L_i, E): Add employee E to list L_i. Steps are:

1) $L_i \leftarrow L_i \bigcup \{E\}$

2) $L_j \leftarrow L_j - \{E\}$ (j=1 if i=2, else j=2)

3) $L \leftarrow L \bigcup \{E\}$

b) Tran(L_i, L_j E): Transfer employee E from L_i to L_j. Steps are:

1) $L_i \leftarrow L_i - \{E\}$

2) $L_j \leftarrow L_j \bigcup \{E\}$ (j=1 if i=2, else j=2)

3) $L \leftarrow L \bigcup \{E\}$

Please note that some steps in the definition of the transactions may seem awkward, but they are needed to guarantee that transactions can maintain consistency if they are run to completion without interference. These provisions

were taken due to our desire to make all transactions of type *NC*, so that we can show later on how to increase compatibility of transactions with the introduction of conditional statements.

Our goal now is to find out which transactions are compatible. We will not go into detail through the steps of the algorithm, because then we would have to enumerate too many trivial instances. Instead, we are just going to point out the instances of interest.

Recall that the consistency constraints are:

C_1: $L_1 \bigcup L_2 = L$

C_2: $L_1 \bigcap L_2 = \emptyset$

Assume that $L_i = \{E \in U \mid l_i(E)$ *is true*$\}$, where $l_i(E)$ = employee E works at site i (i=1,2). This implies that

c_1: $(l_1 \vee l_2) \Leftrightarrow l$; and

c_2: $(l_1 \wedge l_2) \Leftrightarrow 0$,

therefore $c^* = c_1 \wedge c_2$ is $[(l_1 \vee l_2) \Leftrightarrow l] \wedge [(l_1 \wedge l_2) \Leftrightarrow 0]$, and the combinations of values satisfying c^* are:

l_1	l_2	l
1	0	1
0	1	1
0	0	0

Are *Del* and *Add* types compatible?

Let $T_1 = Del(L_1, E)$ and $T_2 = Add(L_1, E)$ and run the following schedule:

11, 12, 21, 22, 23, 15.3. The final values will be $l_1(E)=1$, $l_2(E)=l(E)=0$ which do not satisfy c^*. This makes *Del* and *Add* incompatible types.

Are *Del* and *Tran* types compatible?

Let $T_1 = Del(L_1, E)$ and $T_2 = Tran(L_1, L_2, E)$ and run the following schedule: 11, 12, 21, 22, 23, 15.3. The final combination of values $l_1(E)=0$, $l_2(E)=1$, $l(E)=0$ does not satisfy c^*. Therefore *Del* and *Tran* are not compatible types.

Are *Add* and *Tran* types compatible?

Let $T_1 = Add(L_1, E)$ and $T_2 = Tran(L_1, L_2, E)$ and run the following schedule: 11, 21, 22, 12, 23, 13, that produces the values $l_1(E)=0$, $l_2(E)=0$, $l(E)=1$ which do not satisfy c^*. Therefore *Add* and *Tran* are not compatible types.

In the same way it is possible to show that *Add* transactions are not compatible among themselves. The same goes for *Tran* transactions. *Del* transactions can be shown to be compatible among themselves. ○

If we want more compatibility among the three different transactions in the previous example, it will be necessary that we reformulate their codes. Since we want to avoid unwanted (problem causing) interleavings and this happens most of the time when the deletion of an element from a set is attempted and it is really not in the set, or inserted into one that he is already in, then it is right to take the proper precautions. Let us therefore reformulate the transactions as follows:

a) *Del*(L_i, E). Delete E from L_i. Steps are:

1) **If** $E \in L_i$ **then** $L_i \leftarrow L_i - \{E\}$

   Else ABORT

2) $L \leftarrow L - \{E\}$

b) *Add*(L_i, E). Add E to L_i. Steps are:

1) **If** $E \notin L$ **then** $L_i \leftarrow L_i \bigcup \{E\}$

   Else ABORT

2) $L \leftarrow L \bigcup \{E\}$

c) *Tran*(L_i, L_j, E). Transfer E from L_i to L_j. Steps are:

1) **If** $E \in L_i$ **then** $L_i \leftarrow L_i - \{E\}$

   Else ABORT

2) $L_j \leftarrow L_j \bigcup \{E\}$ (j=2 if i=1; else j=1)

Our transactions have been modified and are now of type IF_1. To test their compatibility we can use the modified version of algorithm 5.3.1 for IF_1 transactions as explained in corollary 5.3.2.

To test if *Add* and *Tran* transactions are compatible we will first construct the set $R(E)$, since E is the only common element that is accessed by both transactions, and with the help of VALID construct later the set MOD_VALID(E). The membership of E in a set is questioned just for the sets L (in *Add*), and L_i (in *Tran*), so we have

$$R(E) = \{ \textit{all pairs } (l_i(E), l(E)) \mid l_i(E),\ l(E) \in \{0, 1\}\}$$
$$= \{(0,0),\ (0,1),\ (1,0),\ (1,1)\}.$$

Since

$$VALID=\{(l_i(E),l_j(E),l(E)) \ \textit{consistent initial combinations of } 0,1\}$$
$$=\{(1,0,1),\ (0,1,1),\ (0,0,0)\},$$

then

$$MOD\_VALID=\{(l_i(E),l_j(E),l(E) \ \textit{consistent initial combinations of } 0,1 \ \textit{that include a pair } R(E)\}$$
$$=\{(1,0,1),\ (0,1,1),\ (0,0,0)\}$$

Let $A=Add$ and $T=Tran$ then

$$PI_{12}=\{A_1,\ T_1,\ A_2,\ T_2;\quad T_1,\ A_1,\ T_2,\ A_2;\quad A_1,\ T_1,\ T_2,\ A_2;\quad T_1,\ A_1,\ A_2,\ T_2\}.$$

Since E is the only element accessed by both transactions and #(MOD_VALID(E))=3 and $\#(PI_{12})=4$, then 12 different cases would have to be considered to test compatibility, but in $Add(L_i, E)$, the set L_i is an input variable, and therefore our algorithm considers in its analysis the two cases: $Add(L_i, E)$ and $Add(L_j, E)$ since *Tran* is a function of L_i and L_j. This brings the total number of cases to consider to 24. To analyze each of the 24 different cases is a straightforward exercise that we will not perform here. The result of such analysis will demonstrate that *Add* and *Tran* are compatible types.

In an analogous way we can analyze the remaining cases and conclude that

cs(Add)={{Add, Tran}},

cs(Del)={{Del, Tran}},

cs(Tran)={{Tran, Del}, {Tran, Add}}.

We should observe at this point that the same compatibility would have held if we had n sets, such that $\bigcup_{i=1}^{n} = L$ and $L_i \bigcap L_j = \emptyset$ for all $i \neq j$, $i,j=1,...,n$. This happens thanks to testing always at the beginning of a transaction if the

adequate condition is fulfilled.

In our particular example, the inclusion of the "If...then...else abort" statements as tests for a desired condition, forces the abortion of the transaction if not fullfilled. This probably adds some complexity to the transaction processing, but we definitely believe that it allows for an improvement in performance, by making the compatibility sets of the different types larger. The added complexity, due to the "If..." statements is minor, and will always insure proper management of the desired elements. Note that this advantage is enhanced as the number of sets participating in the constraints (see previous paragraph) grows larger. By using this technique, based on avoiding interleavings leading to possible data inconsistencies, we have shown a possible way of increasing the number of compatible transaction types.

5.4. Latest Value Overwrite Transactions

In the two previous sections we have talked about linear assignments' transactions and transactions that perform set operations. In this section we shall talk about transactions that are frequent in databases where the new values of the objects are not derived from other values in the database, but are set (imposed) by decisions of the persons who use the DBMS (e.g.: Salary of employee 324 will be 45000 dollars). These types of transactions are very common at the creation or deletion of part or all of the database. Also, most of the time, in this type of transaction, the modifications to a database object, performed by the latest transaction are those that must be recorded. In this section we will briefly study some of the properties of this type of transactions.

Definition 5.4.1 (Transaction's timestamp):

The timestamp of a transaction is a unique number set by the system to a transaction at the moment it is submitted. Timestamp numbers are totally ordered.

Definition 5.4.2 (Object's timestamp):

The timestamp of an object is the number corresponding to the last transaction that modified it.

Definition 5.4.3 (LVO transaction):

An LVO (Latest Value Overwrites) transaction is one that overwrites a database object if its timestamp is greater than or equal to the object's timestamp.

Proposition 5.4.1:

An LVO transaction is not guaranteed to be compatible with a non LVO transaction.

Proof:

Let "$a + b = c$" be a consistency constraint. Let T_1, T_2 be two transactions such that,

T_1: 1) $a \leftarrow a + x$

2) $c \leftarrow c + x$

T_2: 1) $b \leftarrow b^*$

2) $c \leftarrow c^*$; (such that $b^* - b = c^* - c$)

Let $ts(T)$ denote the timestamp of the transaction T. Assume that

$ts(T_1) > ts(T_2)$. Consider now schedule S = 11, 21, 22, 12. After executing schedule S the equation of the consistency constraint will look as follows: $(a + x) + b^* = c^*$. If $x \neq 0$, then obviously this last equation does not hold, and therefore consistency is not maintained. ○

Theorem 5.4.1:

Let T_1, T_2 be two transactions of type LVO.

Let C_1,...., C_n be the database (DB) consistency constraints.

Let $obs(T_j, C_i) = \{o \in DB \mid o \textit{ is part of } C_i \text{ and } \textit{is accessed by } T_j\}$

If for all i=1,...n

$$obs(T_1, C_i) \bigcap obs(T_2, C_i) = \emptyset$$

or

$$obs(T_1, C_i) = obs(T_2, C_i)$$

then T_1 and T_2 are compatible.

Proof:

If $obs(T_1, C_i) \bigcap obs(T_2, C_i) = \emptyset$ for an i, then obviously no problem can arise: the transaction accessing C_i will see that consistency of this constraint is maintained. If $obs(T_1, C_i) = obs(T_2, C_i)$, then, since the transaction with the largest timestamp, say T_2, overwrites, at the end of a schedule, the values of the objects in C_i will be the ones set by T_2. Since T_2 preserves consistency, by transaction definition, then C_i's consistency will not have been violated. To conclude, consistency will always be maintained at the end of T_1, T_2's interleaving, and therefore the two transactions are compatible. ○

Observation 5.4.1:

If the previous theorem's conditions do not hold, then there is no guarantee that T_1 and T_2 are compatible. Suppose that "$a + b = c$" is a consistency constraint, say C, and consider the following two transactions:

T_1: 1) $a \leftarrow a^*$

2) $c \leftarrow c^*$; (such that $a^* - a = c^* - c$)

T_2: 1) $b \leftarrow b^+$

2) $c \leftarrow c^+$; (such that $b^+ - b = c^+ - c$)

Assume that $ts(T_2) > ts(T_1)$, and that we will run the schedule S= 11, 21, 22, 12. These assumptions will produce for the variables a, b, c the new values a^*, b^+, c^+, respectively. But do these new values make the consistency constraint C true? I.e., it is true that $a^* + b^+ = c^+$? Since $b^+ - b = c^+ - c$ then there exist a number x, such that $b^+ = b + x$ and $c^+ = c + x$. By an analogous reasoning, there must exist a number y, such that $a^* = a + y$ and $c^* = c + y$. Our consistency constraint will hold just if $(a + y) + (b + x) = (c + x)$. This equation is mathematically reducible to $a + y + b = c$. If $y \neq 0$ then C is not fulfilled and therefore the consistency is immediately violated. We should observe that the reason for the consistency violation is that $obs(T_1, C) \bigcap obs(T_2, C) = c \neq \emptyset$ and $obs(T_1, C) \neq obs(T_2, C)$. I.e., the conditions on the *obs* sets, as expressed in the theorem, were not met, and therefore the consistency can not be guaranteed. ○

5.5 Conclusions

In this chapter we have studied compatibility properties among three different types of transactions: 1) Linear transactions; 2) Transactions executing

set operations; and 3) Latest Value Overwrite Transactions. The conclusions about these different types are the followings:

1) The results obtained with the different types of linear transactions can be helpful in achieving higher concurrency in applications like banking operations, accounting of companys, etc., if an SK mechanism is used. With more restrictions, and aids, such as *indicators*, these results could be expanded. The expansion would make the use of SK very beneficial in cases such as the above mentioned examples.

2) Transactions executing set operations are easy to analyze if we use a boolean algebra based algorithm, as described in section 5.3. This algorithm has the possibility of being included as a part of the DBMS, and letting the computer do the work of determining the compatibility of two transactions. More work needs to be done in this area, especially in the way of specifying the transactions so that the computer can understand the semantics.

3) The results in the area of Latest Value Overwrite Transactions are restricted, but can be useful in some applications, e.g., on a database containing information about the employees of a company.

Chapter 6

COPING WITH LONG LIVED TRANSACTIONS

6.1. Introduction

As the name indicates, a *long lived transaction* is a transaction whose execution, even without interference from other transactions, takes a substantial amount of time, possibly on the order of hours or days. A long lived transaction, or LLT, has a long duration compared to the majority of other transactions either because it accesses many database objects, or because it pauses for inputs from the users, or a combination of these factors. Examples of LLTs are transactions which produce the monthly account statements at a bank, transactions which process claims at an insurance company, and transactions which collect statistics over an entire database [Gray2].

In most cases, LLTs present serious performance problems. Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Gray2]. To make a transaction atomic, the system usually locks the database objects accessed by the transaction until it commits, and this typically occurs at the end of the transaction. As a consequence, other transactions wishing to access the LLT's objects are delayed for a substantial amount of time.

Furthermore, the transaction rate of abortion can also be increased by LLTs. As discussed in [Gray1], the frequency of deadlock is very sensitive to the "size" of transactions, that is, to how many objects transactions access. (In the analysis of [Gray1], the deadlock frequency grows in proportion to the fourth power of the transaction size.) Hence, since LLTs access many objects, they may cause many

deadlocks, and correspondingly, many abortions. From the point of view of system crashes, LLTs have a higher probability of encountering a failure, and are thus more likely to encounter yet more delays and more likely to be aborted themselves.

In general, there is no solution that eliminates the problems of LLTs. Even if we use a mechanism different from locking to ensure atomicity of the LLTs, the long delays and/or the high abort rate will remain. No matter how the mechanism operates, a transaction that needs to access the objects that were accessed by a LLT cannot commit until the LLT commits.

However, *for specific applications*, it may be possible to alleviate the problems by relaxing the requirement that an LLT be executed as an atomic action. In other words, without sacrificing the consistency of the database, it may be possible for certain LLTs to release their resources before they complete, thus permitting other, waiting transactions to proceed.

To illustrate this idea, consider an airline reservation application. The database (or actually a collection of databases from different airlines) contains reservations for flights, and a transaction T wishes to make a relatively large number of reservations (for a round-the-world tour, say). For this discussion, let us assume that T is a LLT. In this application it may not be necessary for T to hold on to all its resources until it completes. For instance, after T reserves a seat on flight F_1, it could immediately allow other transactions to reserve seats on the same flight. In other words, we can view T as a collection of "subtransactions" T_1, T_2, ..., T_n that reserve the individual seats.

However, we do *not* wish to submit T to the system simply as a collection of independent transactions because we still want T to be a unit that is either

successfully completed or not done at all. That is, we would *not* be satisfied with a system that would allow T to reserve three seats and then (due to a crash) do nothing more. On the other hand, we would be satisfied with a system that guaranteed that T would make all of its reservations, or would cancel any reservations made if T had to be suspended.

This example shows that a control mechanism that is less rigid than the conventional atomic-transaction ones but still offers some guarantees regarding the execution of the components of a LLT would be useful. In this chapter we will present suggestions for such a mechanism.

In the airline example, it seems natural to execute each seat reservation as an atomic unit and to allow arbitrary interleavings of such sub-transactions. In other cases, though, it may only be possible to interleave the sub-transactions of a LLT with other transactions if certain conditions are satisfied. For example, consider an insurance claim transaction L that is a LLT. Again, L can be broken into sub-transactions to enter the original claim information, check the history of the customer, verify the claim with the police, estimate the replacement value of the property, and so on. It may very well be possible to interleave these sub-transactions with those of other concurrent claims, but it may be dangerous to interleave a claim transaction with a transaction that is changing the table with which property values are estimated. (If this were allowed, L could record in the database one value for the settlement and then write a check for a different value to the customer.)

Thus, in this case we would like to have a system that could take as input the rules for interleaving LLTs and enforce them. Such a system would let L observe the partial results of other insurance claims (speeding up execution

times), but not of the transaction that modifies the property value tables. Similarly, this last transaction could not observe the results of L until L finished.

Two ingredients are necessary to make the ideas we have presented feasible: system support and properly designed LLTs.

(a) **System support.** On a conventional database system there are only two choices for a LLT. If we submit it as a single transaction, consistency and atomicity are guaranteed, but at a high cost. If we break up the LLT into a collection of sub-transactions, performance improves but the system makes no guarantees about consistency and atomicity. For many LLTs, neither choice is acceptable. † Therefore, additional facilities within the system, or on top of an existing system, are needed to define how LLTs can be broken into components and how these components can be executed.

(b) **Properly Designed LLTs.** The LLTs (and the database itself) must be designed in such a way that they can be broken up into smaller units that can be interleaved. For this, the programmers (and/or users) must understand the application well, and this is not always easy.

In this chapter we will concentrate in discussing the issues in the first category, and leave the issues in the second one for for future research and discussion. In section 6.2 we will introduce the reader to the importance of restricting the interleavings of LLTs, and in section 6.3 a mechanism to serve that purpose will be presented. Such a mechanism is an adaptation to LLTs of the ideas of Semantic Knowledge (SK) presented in section 2.3. In section 6.4 we will

---

† For some LLTs, like one that collects statistics, no guarantee of consistency may be acceptable.

describe some of the relevant components needed to manage LLTs in a DBMS that incorporates the ideas of SK.

6.2. Restricting Concurrency

In the introduction we argued that some LLTs are composed of sub-transactions that are not real transactions, i.e., they are not consistency preserving units and hence need a special execution mechanism. In the example we used, an insurance claim LLT could be interleaved with other claims that access the same data, but not with a transaction that modified critical price tables. In other words, if LLTs L_1, consisting of sub-transactions $T_{1,1}$, $T_{1,2}$, $T_{1,3}$, and L_2, consisting of sub-transactions $T_{2,1}$, $T_{2,2}$, $T_{2,3}$, are claims, and T_x modifies the critical tables, then the execution schedules

$$\cdots \quad T_{1,1} \quad T_x \quad T_{1,2} \quad T_{1,3} \quad T_{2,1} \quad T_{2,2} \quad T_{2,3} \quad \cdots \tag{1}$$

and

$$\cdots \quad T_{1,1} \quad T_{2,1} \quad T_x \quad T_{1,2} \quad T_{2,2} \quad T_{2,3} \quad T_{1,3} \quad \cdots \tag{2}$$

should not be allowed because they may violate consistency. However, the schedules

$$\cdots \quad T_{1,1} \quad T_{1,2} \quad T_{1,3} \quad T_x \quad T_{2,1} \quad T_{2,2} \quad T_{2,3} \quad \cdots \tag{3}$$

and

$$\cdots \quad T_{1,1} \quad T_{2,1} \quad T_{1,2} \quad T_{2,2} \quad T_{2,3} \quad T_{1,3} \quad T_x \quad \cdots \ , \tag{4}$$

for this application, are acceptable.

To improve performance, the system needs as much flexibility as possible in scheduling transactions and sub-transactions. Specifically, it needs more flexibility than a conventional atomic transaction mechanism which would allow schedules like (3) but not ones like (4). (Note that in schedule (4), L_2 can be

started before L_1 completes.)

How can the system be informed of the allowable interleavings, so it can use them to improve performance? There are many answers to this question. We will here describe a strategy that uses *compatibility sets* to specify the valid interleavings. This strategy is an adaptation to LLTs of one presented in [Garc]. Here, we will only summarize the main concepts and refer the reader to the mentioned reference for details.

6.3 Compatibility Sets.

The basic idea here is to break up LLTs into sub-transactions. However, the sub-transactions in this case are not true transactions, as already mentioned previously. Therefore, executing the sub-transactions as atomic actions is not sufficient.

To describe the valid interleavings of sub-transactions (e.g., like the ones in schedules (3) and (4)), LLTs and regular transactions are classified, say by the system administrator, into *semantic types* (e.g., L_1 above may be of type "insurance claim"). For each LLT type, a *compatibility set* is defined. If LLT L_1 is of type Y_1, and this type has compatibility set $CS(Y_1) = \{Z_1,...,Z_q\}$, then the sub-transactions of L_1 can be interleaved with the sub-transactions of transactions of types Z_1 through Z_q. In other words, after a sub-transaction of L_1 completes, the data it accessed may be accessed by the specified sub-transactions (or transactions), even if L_1 as a whole has not completed. However, each sub-transaction must still be executed as an atomic action. (Note that regular transactions do not have compatibility sets since they have no sub-transactions.)

For the insurance claim example, let us call the type of claim LLTs L_1 and

L_2, *IC*; the type of the table update transaction T_x, *TU*; and let type *CH* represent transactions that simply check the history of a given customer. We can then define the compatibility set of *IC* to be

$$CS(\ IC\ ) = \{\ IC,\ CH\ \}$$

This would make schedules like (3) and (4) valid, and in addition would allow a transaction T_c of type *CH* to be executed at any point in these schedules. Schedules like (1) and (2) would not be allowed because *TU* is not in *CS*(*IC*).

We will say that transaction (either normal or LLT) T is ***compatible*** with LLT L if the type of T is in the compatibility set corresponding to L. In our example, L_1 and L_2 are compatible with each other (since *IC* is in *CS*(*IC*)), but in general, the compatible relationship is not reflexive. That is, we could have a LLT that could not tolerate sub-transactions of another LLT of its same type. Also note that if no transactions are compatible with some LLT, then that LLT must be performed as a conventional transaction (i.e., as an atomic unit). At the other extreme, if all transactions and sub-transactions are compatible with all LLTs, then no special provisions need to be taken, and transactions and LLTs can run concurrently with no risk of violating consistency.

In our mechanism each sub-transaction of an LLT is provided with a compensating sub-transaction which reverses the actions performed by it (but does not necessarily return the database to its original state). If the LLT must be aborted, the compensating sub-transactions are executed by the system in the reverse order in which the sub-transactions of the failed LLT were committed.

The definition of compatibility sets must be considered in light of the compensating sub-transactions. That is, when we state that T is compatible with LLT L, it must be possible to interleave T with the sub-transactions *or with the*

compensating sub-transactions of L. In our insurance claim example where LLTs of type IC are compatible, the schedule

$$\cdots \quad T_{1,1} \quad T_{2,1} \quad T_{1,2} \quad T_{2,2} \quad T_{2,2}^{-1} \quad T_{1,3} \quad T_{2,1}^{-1} \quad T_x \quad \cdots , \tag{5}$$

where $T_{i,j}^{-1}$ is the compensating sub-transaction of $T_{i,j}$, must also be valid. (Contrast this schedule with (4).)

6.4. **Managing Long Lived Transactions Using SK. A First Approach.**

In this section we will describe with little detail some of the relevant components needed to manage LLTs in a DBMS that incorporates the adaptation of the SK ideas set forth in the previous section. We are going to assume here a centralized system where no two LLTs can be compatible. Considering that most of the actual DBMSs are centralized and that in the present time most applications process very few LLTs we think that studying this restricted case is important. We think too that given that in the future, a distributed environment and an environment that handles LLTs with unconstrained compatible sets should be studied, then the knowledge that we can gain from this restricted case will also be very useful. We will not deal with the latter case here.

For the purposes of this section we will assume a database divided in blocks stored in secondary memory, usually a disk unit(s), each of them holding one or more records. Records, the units of locking, will consist of one or more fields. They will be identified by a specific number, and each time a transaction requires access to them, the block holding the record will be read into a page in main memory. To that end a page table will be kept in core to establish the correspondence between pages and blocks. A list of the free blocks will also be in core, to be used whenever new records are created and a free block is needed. Every time

a block is freed in secondary memory its identifier will be added to that list.

6.4.1 The transaction manager (TM)

For every transaction submitted to the DBMS, the TM typically: 1) creates a process that executes the actions of the transaction; 2) calls the crash recovery manager to take provisions for recovery in the case of a failure, so that transactions may be executed atomically; and 3) uses the services of our concurrency control mechanism to enforce that access to the DB objects by the transaction is only done in accordance to the rules of SK. (In the algorithms that appear in Appendix A we have to think of our normal transactions as being their *local* transactions, and our LLTs, their *nonlocal* ones. Please see the reference for more details.) In the case of a LLT the TM will execute the subtransactions one by one in accordance with the information received from the LLT executor (see below).

6.4.2 The LLT executor (LEXEC).

The LEXEC calls upon the transaction manager (TM) to execute the different subtransactions of the LLT. When a LLT is submitted to the system it will be recognized as such by the information in its header, and passed immediately to the LEXEC for control. The "executor" then gathers from the code of the LLT the types and order of execution of the different subtransactions, along with some or all of their corresponding input data, and safely stores all this information in a special relation of the database called the "LLT Agenda". Since subtransactions sometimes depend for their input values on previous subtransaction(s), not all of them will be known at LLT's submission time, but will be placed in the LLT Agenda, by the LEXEC, as soon as they are produced.

Together with each subtransaction's type will also be stored the type of its corresponding compensating subtransaction, but not all its input data, as this will be known completely only when the subtransaction finishes execution.

At the scheduled LLT's processing initiation time the LEXEC will pass the type and input data of the first subtransaction to the TM. When this subtransaction successfully finishes execution the TM will notify the LEXEC, which in turn will pass the information concerning the next subtransaction to the TM,.... and so on, until the last subtransaction has been processed successfully. At this point in time the LEXEC will signal the commit of the LLT and mark its information on the LLT agenda as 'old', so that this space can be used for subsequent LLTs.

To conclude our brief introduction to LEXEC we have to say that the information written onto the *LLT Agenda* could be immediately stored in secondary memory to have it safe and ready after a system crash. But considering that the *Agenda* is a relation that is part of the database, we will write the actions of the transaction that wrote the information onto the *Agenda*, to the undo/redo log of our crash recovery manager (see section 6.4.4).

6.4.3 Table of Compatibility (TOC):

This table gives information about transaction types and their compatibility sets. We can think of it as being organized in the following way: The first column will have the names of transaction types and the second the compatibility set of the transaction type in the first column. A third column will have the identifiers, and submission time timestamps, of in-progress transactions of the given type. The creation and maintenance of the information in the first two columns will be the responsibility of the database administrator (DBA). The tran-

saction identifiers on the third column will be written by the LEXEC and the TM respectively at LLT's or normal transaction's submission time, and erased after the transactions have successfully finished execution. It will be very useful to have this table always ready in core, so that when a transaction T starts processing, i.e. when its starts executing step 1 of our concurrency control algorithm, which appears in Appendix A, the value of $ID(T)$ can immediately be assigned. In order for this always to be possible we will have to load this table from secondary memory at every system start or restart. The latter, as we will see in the description of our crash recovery mechanism, will be necessary after restart from a failure, in order for the mentioned mechanism to properly recuperate all the needed information for immediate continuation of normal processing.

6.4.4 The crash recovery manager (CRM)

This is the software component in charge of taking enough provisions for recovery in the case of a computer crash, and of executing the recovery itself. Such provisions will insure that all modifications done by committed transactions or subtransactions (of a LLT) will survive a failure, and that those made by uncommitted ones will not be seen after the recovery stage, i.e., that they can be undone in the case that they have already migrated to the physical database.

The objective of this subsection is to explain how to modify and use a conventional *undo/redo* log based CRM [Gray3], to assure reliability of a DBMS of the type mentioned in the introduction to section 6.4. We will start by explaining in subsection 6.4.4.1 the generalities of such a CRM, including the added mechanisms needed to handle LLTs and their respective subtransactions. In subsection 6.4.4.2 we turn to discuss the special features needed to support the use of a SK

based concurrency control mechanism as outlined in Appendix A of [Garc].

6.4.4.1 Undo/Redo Logging for transactions and LLTs

Our CRM, as already mentioned, will use an undo/redo log. The log or audit trail [Kohl, Verho], as it is also known, is a region of n contiguous blocks of secondary memory (n to be determined by the DBA) to be used in a circular buffer fashion. For the sake of reliability in the case of a media failure (accident/crash causing loss of information stored on disk), we will duplicate the log onto a second disk. The two disks will be assumed to have independent failure modes, i.e., that within a short period of time, usually the time needed to copy the whole information in one log from one disk to the other, both disks will not fail [Lamp].

To insure full reliability of transaction or subtransaction processing, as described at the beginning of this subsection, in the case of system crash (partial or total loss of main memory information) the CRM will use a special protocol called the Write Ahead Log Protocol (WAL). This protocol implies that before a database object (DB-object) is modified in the physical DB an entry in the next available position in the log will be made. Such entry will contain the number of the transaction that accessed the object, its *id* (identification), its *old* value, and its *new* value. Besides that information, each transaction will log a *Begin Transaction* entry at processing start, and an *End Transaction* or *Commit* entry together with the transaction number, when its execution has successfully completed. (Note: For the purposes of our CRM, a transaction or subtransaction writes its *End (Sub)transaction* record onto the log just when it is ready to commit, and therefore we will invariably call that record the *Commit* record also.) In

the case of a subtransaction the CRM will also log at completion time, before the *End Subtransaction* entry, the type of the compensating subtransaction along with the necessary input data. If this is done, the subtransaction's modifications can be compensated for in the case of abortion of the owner LLT, during normal processing (not failure recovery), due, for example, to deadlock or timeout reasons. It is necessary to point out here that the numbers identifying a subtransaction will have different characteristics than the ones identifying a transaction. (Eg.: A transaction could easily be identified with a unique natural number, for example 123. A subtransaction will have a number like $L456.7$, where $L456$ is the number of the LLT (the L before the digits recognizes it as a LLT), say T, and the 7 refers to the seventh subtransaction of T). It is important that the CRM knows about it so that in the case of an abortion the proper information can be handed to the TM, and also because apart from the information concerning compensating transactions, some more information, to be discussed later on, might be necessary to log for subtransactions.

To recover from a system crash is easy. Knowing the address of the last record written to the log, the CRM just needs to read the log backwards, and as it goes, reinstate the *old* values of records modified by transactions or subtransactions still in progress at the time of the crash, and accumulate in a set the identifiers of transactions or subtransactions for which a *Commit* record is found (i.e., of transactions or subtransactions that already successfully finished execution). The committed transactions and subtransactions will then be redone by reading the log forward from the beginning to the end, and for every object that appears in the log as having been modified by a transaction or subtransaction in the mentioned set, reinstate the *new* value in the database.

Reading the complete log backwards, and then forwards, often might be unnecessary and time consuming. Therefore periodic checkpoints, complete flushes of main memory information to disk, will be made, and a record of such action written to the log. Now at system failure's recovery time the same, already explained process of *undo*, and collection of the identifiers of committed transactions and subtransactions, will take place all the way from the end of the log backwards to the checkpoint entry. Thereafter just the old values of records accessed by uncommitted transactions at crash time, and in progress at the time of the checkpoint (UIP), will have to be put back in their respective places. And more, the log will be read backwards just until the *Begin Transaction* entry of the oldest UIP transaction is read. The recovery will continue now, as expected, redoing the committed transactions from the checkpoint onwards. The recovery will finalize by taking a checkpoint, so that we can be sure of having our recovered data safely in disk. Please note that the *undo* process can in some cases go backwards not just past the last checkpoint entry, but past previous checkpoints until the earliest submitted UIP transaction has been completely undone.

The log could of course grow very large and therefore a mechanism to move the beginning of the log forward, will be utilized from time to time. This mechanism is sometimes called *firewalling* [Kent]. We suggest the interested reader refer to the mentioned paper for details.

To backout a LLT will also be easy. Upon decision to do so, by the user or by the system (see sections 6.4.5 and 6.4.6 on deadlocks and timeouts), the following two steps will be taken:

1) The CRM reads the log backwards until encountering the *Begin Subtransaction* record for the subtransaction in progress, if any, and as it does this, it replaces the values of records modified by the subtransaction by the *old* values that appear in the log.

2) The CRM continues reading the log backwards until hitting the *Begin LLT* record and as it does this, it passes one by one to the TM, the encountered information of the compensating subtransactions. Such compensating subtransactions will be executed in the reverse order that the subtransactions were executed. But this is no problem, since as we already know, that is exactly the order in which they will be found in the log when reading it backwards. Note that these compensating subtransactions will be executed by the TM as if they are normal transactions (but not consistency preserving). The CRM will therefore write onto the log the proper information (old/new values), so that the LLT backout's modifications will not be lost in the case of a system crash.

6.4.4.2 Adding SK to the CRM

Are the features mentioned in the previous subsection enough for our CRM to handle the intricacies of SK applied to the management of LLTs? No, our CRM as it stands now is not complete, there are still small important additions to be made. To better understand the specifics of such additions it will be necessary to review some facts about SK and to go through some explanations. The important matter will be that our CRM closely cooperates with the concurrency control manager, in order for transactions and LLTs to obey DB consistency. Its responsibility will therefore consist of making available after a crash all the neces-

sary information needed by the concurrency control manager to carry out its duties with no problem.

As we know, even though subtransactions are not consistency preserving units, they are handled as transactions by the TM, and their modifications, as we have already seen, are recovered after a system crash in the same way that transaction's modifications are recovered. This implies that LLTs could not be finished at the time of a system crash, and will not be undone completely at recovery. Therefore they will be restarted at the start of their subtransaction in progress at the crash (note that this subtransaction was already undone by the CRM), or at the start of the subtransaction which is next to be processed, if there was no subtransaction running when the computer crashed. This gives us the advantage of not losing the already processed parts of the LLT, but in order to continue processing the LLT after the crash, the SK based concurrency control mechanism will have to be supplied, as already said, with the SK accounting information. A normal concurrency control mechanism, such as a two phase locking (2PL) one, will not need this information.

From our knowledge about the SK concurrency control mechanisms, we already know what the needed accounting information from LLTs will be, when normal processing restarts after a crash: For every object o accessed by a LLT L in progress at the time of the system failure, the values of $GL(o)$, $SW(o)$, $PRE(o)$ and $Rel(o)$; and for every such LLT L, $Wait(L)$ and $T\_Wait(L)$. Note that we will not have to worry about in progress normal transactions. At crash recuperation all uncommitted transactions at failure time are completely undone, so that when normal processing restarts there will be no such transaction that is halfway through its processing and needs to be terminated. We will need, however, some

information about objects accessed by committed, normal transactions compatible with a non terminated LLT L, and that participated in an interleaving with L. We will come back later to discuss this.

To see why it is necessary to have all that LLT's information handy at normal processing restart let us view the following scenario. Suppose that the database is checkpointed at time t_0 and that LLT L_1, composed of subtransactions $S_1 \cdots S_8$ (we say $L_1 = \{S_1 \cdots S_8\}$), is submitted to the system at time t_1 ($t_1 > t_0$). At time t_2 ($t_2 > t_1$) subtransaction S_3 accesses object o, and successfully finishes computation before the next system crash, that occurs while subtransaction S_5 is being executed. Since S_3 had already committed, then at recovery all its actions will be redone, but with our original CRM, its accessed objects, including o, will be left with no locks. After the system restarts normal processing, transaction T_2, incompatible with L_1, is submitted. T_2 accesses object o and successfully finishes execution before the last subtransaction of L_1, S_8, starts processing. Finally S_8 also accesses o and finishes successfully, thereby committing L_1. Object o was then accessed first by L_1, then by T_2, and finally again by L_1, forming the cycle $L_1 \rightarrow T_2 \rightarrow L_1$, and cycles in a dependency graph imply that database consistency could have been be violated [Eswar]

From the scenario posed in the previous paragraph we can conclude that we absolutely need the values of $GL(o)$ and of $SW(o)$ when normal processing restarts. They will restrict the access to objects accessed by LLTs just to transactions that are compatible with them. But, what about $Pre(o)$, $Rel(o)$, $Wait(L)$ and $T\_Wait(L)$, for all objects o accessed by LLT L? The support given by the release (Rel) sets is undoubtly necessary for normal execution of our SK concurrency control mechanism. Without the information about the release sets of

objects accessed by L we are risking a problem of the following nature: T_2, a normal transaction, compatible with L, accesses after a system restart the object o already accessed before the crash, by a committed subtransaction S, of L. S sets $Rel(o)=\{L\}$ before the crash, but after the crash this information could erroneously be $Rel(o)=\emptyset$. If this occurs, then when T_2 finishes it could well release the global lock of o. Such an event now opens the possibility that a third transaction T_3, incompatible with L and T_2, accesses o, and when T_3 finishes, that o gets accessed by another subtransaction of L, say S_x. This schedule produced the dependency cycle

$$L \rightarrow T_2 \rightarrow T_3 \rightarrow L \ ,$$

where T_3 is not compatible with L and T_2, and therefore the database consistency could be violated. If $Rel(o)$ is necessary for all objects o accessed by the uncommitted LLT L at system's restart, then the values of $Wait(L)$ and of $T\_Wait(L)$ will be equally necessary, since they closely contribute to the formation of the release sets.

In the case of $Pre(o)$, we have to remember that the necessary contribution of those sets to our concurrency control algorithms in maintaining consistency is in cases like the following: Transaction T_1 accesses o, and before it successfully terminates execution o is accessed by a compatible transaction T_2. If T_1 terminates execution before T_2 does, then in the absence of $Pre(o)$ (that if used would now be $Pre(o)=\{T_2\}$ after the deletion of T_1), the global lock on o could be immediately released, giving then a chance for a third transaction, T_3, incompatible with T_1 and T_2, to access o, opening the possibility of violating consistency. Cases like the one just presented will never show up when restarting our system after a crash, since all uncommitted transactions and subtransactions (of

a LLT) will have been undone before restarting normal processing. It is not difficult to see that in other cases of potential problem, the presence of $Rel(o)$ will be enough so that the global locks will not be released before it is safe to do so. Since $Rel(o)$ will always be made available at system's restart, then we could forget about $Pre(o)$ for all o accessed by L, a LLT in progress at the time of the crash, and assume at restart that $Pre(o)=\emptyset$ for all objects o in the database.

We have now established that $GL(o)$, $SW(o)$, $Rel(o)$, $Wait(L)$ and $T\_Wait(L)$ will be necessary at restart, but it will not be necessary to log all that information for every LLT L. Actually we are fortunate to know that recuperating the values of those variables will be very simple in our restricted environment. If when reading the log backwards after a system crash the CRM encounters the *End Subtransaction* record of a subtransaction S of an uncommitted LLT L, i.e., of a LLT whose *End LLT* record has not appeared in the log already, then besides the normal "redo" processing of actions, just the steps outlined below will be needed to take. Note that these steps, at recovery time, are just in addition to the ones already mentioned in the previous paragraphs, and will be executed together, as the CRM reads the log backwards.

1) For every object o, modified by L, set $GL(o)$='true', since the LLT L has not yet finished.

2) For every object o, modified by L, set $Rel(o)=\{L\}$. Remember that since L was the only LLT allowed in an interleaving of compatible transactions, then L will be the sole member of $Rel(o)$, and it has to be there forcefully since the subtransaction S (of L) has already finished.

3) For every object o, modified by L, set $SW(o)=CS(L)$. The identifier of the

subtransaction S, will immediately tell the CRM the number of LLT L owning it. CRM will search now in the TOC to find out the compatibility set for such a LLT.

After having done the recovery for every outstanding subtransaction of L, the CRM will either hit the "Begin LLT" record, or hit the record of the last checkpoint. At this moment it will do the following:

4) Set $Wait(L)=\emptyset$, since all committed subtransactions are already redone, and the only, if any, subtransaction at the time of the crash has already been undone. (For a better understanding of this explanation and the one in (5) we refer the interested reader to Appendix A of [Garc].)

5) Set $T\_Wait(L)=\{L\}$, since L was the only LLT in an interleaving, if any. $T\_Wait(L)$ could very well be empty if there was no interleaving with other transactions or if certain conditions of the interleaving did not hold. However, setting $T\_Wait(L)=\{L\}$ will do no harm and will add no extra processing time, so that we can safely do so. ○

To conclude our discussion on CRM we will turn our attention to normal transactions, and the information from them, that will be needed at system restart. For our purposes, normal transactions can be grouped in three classes:

1) Transactions not compatible with any LLT in progress at crash time.

2) Transactions compatible with a LLT, say L, in progress at crash time, but that did not access any object previously accessed by L.

3) Transactions compatible with a LLT, say L, in progress at crash time, that

did access at least one object previously accessed by L.

Transactions in the first class present no problem at all: They are not compatible with any LLT in progress at crash time, which implies that they were not able to participate in interleavings with transactions not terminated when the failure occurred. And since at system restart all outstanding normal transactions will have been either redone or undone, then none of the information related to its identifier or accessed objects will be needed.

The transactions of (2) are compatible with an in progress LLT, but since they did not access any objects previously accessed by the LLT, they can be considered on the same conditions as the transactions of class (1), and therefore no information related to them will be needed at restart.

Transactions in class (3) do present a problem if they committed before the crash. In this case some of their related SK information will be needed after a crash to properly recuperate the database, and allow for a problem-free continuation of normal processing. If a transaction in this class was in progress at the moment of a crash, then we know that it will be competely undone, and its modifications will not show up in the database after system restart, which implies that none of its related information will be needed when normal processing begins. Failure to recognize the SK information needs after system restart, from committed transactions in class (3) can result in an eventual consistency violation as the example in the next paragraph shows. A similar example has already been given. The reader should, however, notice that the scenario in the next example is different. In the former case we were making a point on the necessity of logging SK information related to a LLT. In the next example we will show that it is also necessary to log some information concerning committed transactions in class (3),

and therefore the example has some different features.

Consider LLT $L = \{S_1,....S_7\}$ and assume that L is compatible with normal transaction T_1. Suppose that subtransaction S_2 accesses object o_1. After S_2 finishes, transaction T_1 is submitted to the system. It accesses objects o_1 and o_2, and commits before the next system crash, leaving both objects with global locks. When the failure occurs, L is not finished and is forced to continue execution at restart of normal processing with subtransaction S_4, previously undone while recuperating from the failure. After S_4 finishes, a normal transaction, T_2, not compatible with L and T_1 is submitted and asks permission to access o_2. If none of the SK information related to the objects accessed by T_1 is available after the crash then o_2 could be found to be free and the permission granted. This opens the possibility for a consistency violation to occur: Suppose that T_2 finishes before S_7, and that S_7 also accesses o_2. If this last subtransaction finishes successfully, committing thereby L, our transactions dependency graph will show the following cycle:

$$L \rightarrow T_1 \rightarrow T_2 \rightarrow L \ ,$$

and consistency can therefore not be guaranteed. If the CRM would have made available to our concurrency control mechanism at system restart the values of $GL(o)$ and $SW(o)$, for every object o accessed by T_1, then a graph dependency cycle of non compatible transactions would have never occurred. It will suffice here to say that if $GL(o)$ is needed, then $Rel(o)$ is also needed, else we would not have a way of knowing when to safely release the global lock on o.

What about *Wait* and *T_Wait* sets information for (normal) transactions in class (3) that committed before a system crash? It can be seen in Appendix A

that none of the information in those sets will be needed for continuation of normal processing, once the transaction finished execution. Thus our CRM will not care about them when reconstructing the database after a crash.

To provide the concurrency control manager with the information about $GL(o)$, $SW(o)$ and $Rel(o)$ for every object o accessed by a transaction in class (3), say T, that committed before the crash, our CRM will take the following step: Together with the commit record, it will write the value of $ID(T)$ onto the log. As we know (see Appendix A), when a normal transaction T is submitted ot the DBMS $ID(T) = \emptyset$, and it remains like that until it is forced to chose an interleaving descriptor (ID), in our LLT adaptation, a set of compatibility. This occurs when it first tries to access an object globally locked by a LLT, say L, whose compatibility set (CS) contains the type of T. At this moment $ID(T)$ becomes $ID(T) = CS(L)$, and for all objects o already accessed, or to be accessed by T, $SW(o) = CS(L)$. If T never tries to access an object previously accessed by a compatible LLT, then it will run until termination with $ID(T) = \emptyset$.

Given the previously mentioned facts, we conclude that if the value written next to the commit record of a normal transaction is "$\emptyset$", then when the CRM reads the log backwards after a crash, it will not worry about this transaction SK information, but only undo it in the previously mentioned way, in subsection 6.4.4.1. If such value is not equal to $\emptyset$, the CRM will know that T participated in an interleaving with a LLT, say L, and that the written value refers to $CS(L)$. To find out what the identification number (id) of L is, the CRM will go to the TOC and look in the column of compatible sets until it finds the one that equates the value written in the log. From the description of the TOC we know that next to that column, the CRM will find the identifiers of transactions of the given type.

Since in our restricted environment no two LLTs will be allowed to participate in interleavings, then the *id*'s column of the TOC will have, at the most, one *id* of a LLT, in our case the one of L. If the timestamp associated to L's *id* is smaller than the timestamp of T, the CRM can be sure that T was participating in an interleaving with L. (If the timestamp is greater, then T participated in an interleaving with a previous LLT, say R, of the same type as L. Due to the restrictions of our system, R must then have finished successfully before L started, and therefore no special provisions at recovery will be taken for T's accessed objects.)

To windup the discussion of the previous two paragraphs, we can say the following: When reading the log backwards at system restart, if the CRM finds next to the commit record of a normal transaction T, a set value different from $\emptyset$ (remember this is the value of $ID(T)$), then it will have to find out, in the previously described way, the *id* of the unfinished LLT, say L, with the one it participated with in an interleaving (assuming T participated with an unfinished LLT). Then, as it continues to read the log backwards, it will execute, together with all other steps of our CRM, for all objects o modified by this transaction T the following steps:

1) Set $GL(o) = '\ true'$;

2) Set $Rel(o) = \{L\}$; and

3) Set $SW(o) = CS(L)$.

6.4.5 Deadlocks

As has already been pointed out in the introduction, the probability of running into deadlock situations is very sensitive to the "size" of the transaction (number of objects that the transaction will access, time needed to carry out

computation).A LLT as its name indicates, will have a large size, and therefore have a high probability of running into conflicts. We ought then to consider what is the most suitable strategy to solve those conflicts. We could actually take several different approaches to solve the problem, although we think that the best solution can come just out of a close knowledge of the type of workload submitted daily to the DBMS. But whatever we decide to do, the mechanisms should have flexible parameters, so that the DBA can tailor it to its own convenience. These ideas are, however, only one general approach to solving the problem.

Our plan here will be to present a way of solving a deadlock situation once it has been detected. We will not talk about how to detect a deadlock, since as everyone knows that can be done easily and straightforwardly [Agra]. The main focus here will be on how to break a cycle in a transaction's "wait for" graph, i.e., which transaction should be killed, backed out, to break the cycle, and allow for normal processing to continue. And if we decide to back out a LLT involved in the cycle, should we back it out completely, or will it suffice to just undo the running subtransaction and compensate for a few, not all, of the outstanding committed subtransactions? This aspect of our DBMS owes its importance not just to the existence of LLTs, but to the use of SK as well.

In order to be as fair as possible each transaction will be assigned a priority according to its stage of computation, and type: normal or long lived. The former is done to base our decision on which transaction to kill in a deadlock case, on the knowledge of how advanced the transaction is in relation to its whole computational needs. The latter will serve to give the LLTs some sort of advantage (see below) over normal transactions. LLTs have bigger probabilities of getting involved in deadlocks at an advanced stage of life, and it would be unfair to

choose them for back out based on the same factors we will be using for normal ones. We think that LLTs should have some amount of leverage.

At transaction's submission, an approximate number of the objects that the transaction needs to process will be known. Let's call this number α_i for each transaction T_i, and let's call l_i the total number of objects that have already been acquired with local locks. We could assign, barring a small future consideration, for each normal transaction T_i at any moment during its life the priority:

$$p_i = \frac{l_i}{\alpha_i} \ .$$

For LLTs the priority will be different, based on our above reasoning. Here the number of subtransactions composing the LLT will also be known at submission time, and we consider that the successful termination of each of them marks the definite advancement of one more step in its computation. Therefore, if we call s_i the total number of subtransactions and r_i the number of transactions that have already executed, then for each LLT L_i its priority at any moment in time, again barring a small future consideration, will be

$$p_i = \frac{l_i}{\alpha_i} + \frac{r_i}{s_i} \ .$$

Continuing with our desire of being as fair as possible, we will define that each transaction starts processing with priority $p_i = 0$, but if any transaction involved in a deadlock is chosen as the one to kill then when it is submitted again for processing it will start with its old priority in order to give such transactions some advantage over newly submitted ones. This is the just mentioned consideration we needed to take into account, so that now we can rephrase in a final way our recently expressed formulas. Let us first assume that when a transaction is

backed out due to deadlock situations it executes the the operation $old\_p_i \leftarrow p_i$, and that the values of old_p_i will always be null before a transaction is first submitted to the DBMS. Our new formulas are then, respectively for normal and LLT transactions:

$$p_i = old\_p_i + \frac{l_i}{\alpha_i} \quad \text{, and}$$

$$p_i = old\_p_i + \frac{l_i}{\alpha_i} + \frac{r_i}{s_i} \; .$$

Every time that a transaction asks for a lock the TM calls a special routine, called the DD (Deadlock detector) routine. If the lock is granted then normal processing will continue, otherwise an edge will be added to a "wait_for" graph, and the TM notified to delay processing of the current transaction until the required database object becomes free again, and this transaction is the next one in line to access the object. To this end our concurrency control mechanism will call the DD routine every time an object is unlocked, so that the DD can delete the proper edge from the "wait_for" graph and advise the TM of the next transaction to be permitted to access the object. The TM will then reschedule a continuation of the transaction's processing as soon as the CPU can accommodate it. If the last edge added to the graph closes a cycle we know that a deadlock has been detected. In such a case the DD will signal it to the TM together with the identifier of the transaction chosen (see below) to be backed out. The TM will then, with help of our CRM, back out the unlucky chosen transaction.

Up to this point nothing has been said explicitly about how to choose the transaction to kill in the case of a deadlock. The rules will be the following (some comments are included):

Note: For an easier understanding of the rules we will assume the existence, and make references to the following cycle:

$$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow ... \rightarrow T_0$$

1) Choose the transaction with the lowest priority, among the ones involved in the cycle, say T_1.

2) Assume that T_0 and T_1 are not compatible. Let o_1 be an object locked by T_1, and suppose that this is the object for which T_0 is waiting to be released. Suppose that T_1 participated in an interleaving with a LLT L, then $Rel(o_1) = \{L\}$. (To know if this is the situation just check that $Rel(o_1) \neq \emptyset$ and that $T_1 \notin Rel(o_1)$.) In this case two things can happen:

   a) L is one of the transactions already in the cycle. Killing T_1 will not do any good: a smaller cycle will be formed, where T_0 will be waiting for L, since the global lock on o_1 can not be released until $Rel(o_1) = \emptyset$, which will not occur until L finishes execution. T_0 will have to wait for L since T_0 is not compatible with L, as it was not compatible with T_1.

   b) L is not a transaction in the cycle. This an easier situation to deal with, but anyway, T_0 will have to wait for L to finish, to be able to access o_1. This can take long , and it is therefore not advisable to kill T_1.

   We conclude that whatever situation shows up, (a) or (b), the result of killing T_1 is not desirable. The advise is then that we completely or partially backout the transaction that does not have the same problem as T_1. However, since other problems can arise, the choice of the transaction with lowest priority (to backout) will be made according to rules # 3, 4 and 5 below.

3) If T_1 is a normal transaction abort it.

4) Assume T_0 and T_1 are not compatible. If T_1, the transaction chosen to back out, is a LLT, say $T_1 = \{S_1, S_2, ... S_{10}\}$, then we will be able to choose to completely back it out, or just to undo the currently processing subtransaction, say S_7, and compensate for some of the subtransactions in the reverse order of submission until the object that T_0 is waiting for, say o_1, has been freed. E.g.: If the first subtransaction to have accessed the database object o_1 is S_4 then undo S_7 and execute the compensating subtransactions S_6^{-1}, S_5^{-1} and S_4^{-1}, in order to undo as little already done work as possible. (This will not always be feasible. See next paragraph.) The decision on what to do with the LLT, if a partial backout is feasible could be based on the following fact: If many (a number to be specified by the DBA) transactions are waiting for objects accessed by our LLT, T_1, prior to the access of o_1, then it will be useful to back it out to take advantage of the call that is being made to the CRM.

Is it always possible by just partially backing out the LLT, in our case T_1, that o_1 will become free for T_1 to access it? Unfortunately this is not always possible. To see why, suppose that between subtransactions S_1 and S_7 a normal transaction, T_y, compatible with T_1 executed. If after running compensating subtransaction S_4^{-1} we release the global lock of o_4, and allow therefore, the non compatible transaction T_0 to access it, this could easily lead to the violation of a consistency constraint. What should we then do? Three solutions are possible:

a) Back out T_1 completely. This will allow to release the global lock on o_1 without causing any problem in the future.

b) Find out if there was a compatible transaction, like T_y, that executed between S_1 and S_7. If the answer to this question is affirmative then only a full backout will be possible. If the answer is negative, then by executing subtransactions S_6^{-1}, S_5^{-1}, and S_4^{-1}, we will leave the database exactly as it was before S_4 started execution. † In this case it will be possible to do a "partial backout".

c) If no transaction executed between S_1 and S_7, that accessed o_4 or any other object involved in a consistency constraint that includes o_4, then executing S_6^{-1}, S_5^{-1}, and S_4^{-1}, will leave the value of object o_4, with a very high probability, exactly as it was before S_4 started. In this case T_0 could be allowed to access o_1 after S_4^{-1} finishes, with a high probability that it will not violate the database consistency. Under this circumstances a partial backout of the LLT T_1 will be allowed. However, since it might be very time- and space-consuming to take provisions to check easily if o_1 and all objects related to o_1 in consistency constraints were accessed by another transactions between S_1 and S_7, then this solution is not considered a practical one.

Before going on to rule # 5 let us direct our attention to the following problem: What happens if we decide to (partially) backout the LLT T_1, and one of the compensating subtransactions, say S_6^{-1} runs into a deadlock situation? We could apply the rules given here to break the cycle, but we think that there is an easier solution: Kill the transaction that S_6^{-1} will be waiting for. Call this transaction T_x. Why is this solution easier? T_1 has not finished

† In some strange cases, a compensating subtransaction, even without the LLT having had any type of previous interleaving interference from other compatible transaction, will not return the state of the DB to its old state, but just to a consistent one.

execution yet, therefore it has not released any global locks, which implies that S_6^{-1} is waiting for a transaction compatible with T_1 to release the local lock it holds on the "problem object". Due to our environment's restrictions, this compatible transaction T_x has to be a normal transaction, and killing a normal transaction is not a difficult task.

5) If T_1 is a LLT and the transaction preceding it in the "wait-for" graph (T_0) is compatible with T_1 then just undo the currently executing subtransaction of T_1, say S_7. This will immediately break the cycle since both transactions being compatible implies that T_0 was waiting for a local lock to be released, and never for a global lock in a previously executed subtransaction. Note that this rule could be considered as a special case of rule 4.

6.4.6 Timing Transactions

A very useful idea when dealing with LLTs is a mechanism that will time a transaction's waiting period for a locked object. When a transaction asks for an already locked database object then the DD will put it on a waiting queue for the object, and a timer will be set for the transaction. At the DBA's discretion a fixed time could be set, so that when such a time has gone by and the waiting transaction is still on the queue, then it will be assumed to have a large potential for causing a deadlock, and will therefore be aborted. As an alternative, different times could be chosen according to the transaction's size and/or the computational stage of the transaction, so that larger transactions and/or transactions that have already executed a large portion of its work will not be easily backed out. Note the usefulness of such strategy in the case of a LLT.

There are other good uses of timing transactions. Considering that the average transaction's response time (or transaction's turnaround time) is very important, we could use timing to improve such response. Contemplate the case of a LLT $L = \{S_1 \ldots S_{30}\}$. Suppose now that while it executes, several other normal transactions, non compatible with L, are put to wait for objects held by L, that were accessed in L's early stages. In this case all normal transactions will have to wait until L finishes, to continue with their own executions. (Note that other transactions could well be waiting for those transactions to finish too.) The turnaround time for transactions in the system will then be very high. For a better understanding let us vaguely approximate the turnaround time following two different strategies. Suppose that each subtransaction of L would normally, without problems, run in 1 Min., i.e., L will take, if it does not run into trouble, a little bit more than 30 Min. (t_L). Suppose too that each of the other 10 normal transactions takes 1 Min to execute (t_T) and that all of them are stopped by L already when S_{10} finishes. If we wait for L to finish then the turnaround time will be greater than:

$$\frac{t_L + 10 \times (t_T + 20)}{11}$$

$$= \frac{30 + 10 \times (1 + 20)}{11}$$

$$= \frac{30 + 210}{11}$$

$$= \frac{240}{11} \approx 21.8 \; Min.$$

Had we aborted L and resubmitted it, say 10 Min. later, then the turnaround time would be approximately

$$\frac{10 + t_L + \textit{abort time of } L + 10 \times (t_T + \textit{abort time of } L)}{11}$$

$$\approx \frac{10 + 30 + 10 + 10 \times (1 + 10)}{11}$$

$$= \frac{50 + 110}{11} = \frac{160}{11} \approx 14.5 \textit{ Min.}$$

To approach this problem what we could do is to provide the system with an algorithm that takes into consideration the time the LLT L needs to finish, the sum of the different times that the normal transactions have been waiting for L, the number of such transactions, and their time still needed for them to finish. This algorithm will then, taking into consideration the normal daily load of our DBMS, signal the TM whether L should be aborted or not. In case of abortion, the TM will request the CRM to back out the LLT. Such an algorithm could run on behalf of the TM and as a part of our known DD routine everytime a normal transaction is put to wait for an object held by a LLT. To make it simpler we could run it at time intervals set by the DBA. The exact appearance of this algorithm, we think, is beyond the scope of our analysis and is left as an open, very interesting question for future research.

Chapter 7

SUMMARY AND CONCLUSIONS

In this thesis we studied several aspects of a concurrency control mechanism based on semantic knowledge. The motivation behind this study was to take advantage of the fact that in application dependent (AD) databases, the utilization of semantic information can help the DBMS to achieve a better performance, than if it uses a general purpose (GP) scheduler. A GP mechanism, like the widely known 2PL, was shown in chapter 2 to be sufficient, but not necessary to produce consistent schedule executions. On the other hand, we showed, in the same chapter, that our SK based concurrency control mechanism can produce consistent data values that can never be produced by a 2PL mechanism.

Motivated by the potential advantages of SK, we compared a SK concurrency control mechanism with a 2PL one. The scenario for the comparison was a simulation of a two site distributed DB. Chapter 3 presented the simulation model in detail, together with some of the results that showed the performance behavior of both mechanisms in different circumstances. One result is of special interest: The behavior of the mechanism depended mainly on two predictors: The probability of conflict (PRE) and the probability of saved conflicts (PSC). For the environment simulated we showed that for values of PRE $\geq$ 0.035 and/or PSC $\geq$ 0.02, it is worth considering the utilization of SK. The availability of such values is an important result, since they will permit us to consider the use of SK without having to rely on time consuming experiments. It remains an open question to verify in a real life environment the accuracy of those values for our particular model. Another important open question is the following: Do these values, or similar ones, apply to different environments?, i.e., given the

same values for a different system configuration, will SK still be worth considering?

The algorithms in appendix A of [Garc] have to be considered only as a first attempt in the construction of a concurrency control mechanism that utilizes SK. This fact was proven in chapter 4 by showing that shortcuts can be implemented in the algorithms. Improved versions of these algorithms can reduce the average time complexity, and as discussed in the same chapter, for some classes of transaction types it can be a real advantage. How far can we go in the reduction of time complexity for a reliable scheduler that takes advantage of semantic information? At this moment we do not have an answer for this question, but we think that it should be investigated and answered in the future. If we can design an algorithm that improves the performance of 2PL in any (or most) DBMSs environment(s), then we will have made a substantial improvement in concurrency control mechanisms.

In this thesis we were also concerned with providing assistance to the DBA and/or DB programmer about how to determine if two transactions are compatible. Chapter 5 was devoted to making a first approach in that direction. In that chapter we studied properties and relations among compatible transactions. We also gave some suggestions on how to make two transactions compatible if they were not already compatible. The results were encouraging, especially those pertaining to transactions executing set operations. The latter results demonstrate the possibility of incorporating these aids in the DBMS. This would relieve the DBA of the burden of having to determine, by himself, if two transactions are compatible. However, we are still very far from achieving our next goal in this area: "Given any two transaction types (not forcefully different), determine easily

and immediately if they are compatible". The final goal, already considered in the section on set operations, is to be able to equip the DBMS with the proper tools to determine by itself when two transactions are compatible. A large amount of research is still needed to achieve these goals.

Our final subject of SK research was in the area of LLTs. In chapter 3 we showed simulation results where only one transaction, out of every 500 transactions submitted to the system, was long lived. The results indicated that if a LLT is compatible with at least 50% of the remaining normal transactions submitted to the system, a SK mechanism will process the LLT considerably faster then a 2PL mechanism. Encouraged by this result, we studied the possibility of implementing a DBMS that uses SK to process LLTs in an efficient way. In chapter 6 we discussed some of the issues involved.

From the study in chapter 6 we can conclude the following:

1) A DBMS, where no two LLTs are allowed to be compatible, and that uses an SK concurrency control mechanism, can be implemented on top of an already existing DBMS by making small modifications, and additions to the latter.

2) The mechanisms for resolving deadlocks, presented in section 6.3.5, and the ideas in section 6.3.6. concerning timing transactions should be investigated further, in order to achieve applicable results. For the latter ideas we could set as a goal the determination of exactly when a LLT should be aborted, given the scenarios posed in the respective section.

Our final conclusion in this thesis is that the idea of SK, as an aid to efficiently process concurrently executing transactions is still at a very early stage

of development. The SK mechanism has a big potential, as some of the results here have shown, but more research will be needed to fully discover all of its advantages.

REFERENCES

[Agra] Agraval, R., Carey, M. J., Dewitt, D. J. *Deadlock Detection is Cheap* ACM-SIGMOD Records 13, 2 (June, 1983) pp. 19-34.

[Bern] Bernstein, P. A., Goodman, N. *Concurrency Control in Distributed Database Systems.* ACM Computing Surveys 13, 2 (June 1981) pp. 185-221.

[Clark] Oral presentation at the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks (Emeryville, CA., Feb. 3-5), Lawrence Berkeley Lab. of the University of California at Berkeley, Berkeley, CA., 1981.

[Cord] Cordon, R., Garcia-Molina, H. *Semantic Knowledge based Concurrency Control: Variations and Performance.* Technical Report 344, EECS Dept., Princeton University, May, 1985.

[Eswar] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. *The notions of consistency and predicate locks in a database system.* Commun. ACM 19, 11 (Nov. 1976), pp. 624-633.

[Fisch] Fischer, M.J., Griffeth, N.D., and Lynch, N. A. *Global States of a Distributed System.* IEEE Transactions on Software Engineering, Vol. SE-8, No. 3 (May, 1982), pp. 198-202

[Garc] Garcia-Molina, H. *Using Semantic Knowledge for Transaction Processing in a Distributed Database.* ACM Transactions on Database Systems, Vol. 8, No. 2, June 1983, pp. 186-213

[Gray1] Gray, J., Homan, P., Korth, H., Obermarck, R. *A Straw Man Analysis of the Probability of Waiting and Deadlock.* IBM Research Lab., San

Jose, CA.

[Gray2] Gray, J. N. *The transaction concept: virtues and limitations.* In Proc. Seventh Int. Conf. Very Large Data Bases (Cannes, France, Sept. 9-11), ACM, New York, 1981, pp.144-154

[Gray3] Gray, J. N. *Notes on database operating systems.* Advanced Course on Operating Systems Principles, Tech. Univ. Munich, July 1977; also in *Operating Systems: An Advanced Course,* R. Bayer, R. M. Graham, and G. Segemuller, Eds. Springer Verlag, 1979, pp. 393-481.

[Kent] Kent, J., *Performance and Implementation Issues in Crash Recovery* Ph.D. Thesis, Princeton University (June 1985).

[Klein] Kleinrock, L., *Queueing systems.* Wiley Interscience Publication, New York, 1975, Vol. 1

[Kohl] Kohler, W. H., *A survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems.* ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.

[Kung] Kung, H.T., Papadimitriou, C.H., *An optimality theory of concurrency control for databases.* Proc. ACM-SIGMOD Int. Conf. Management of Data (Boston, Mass., May 30-June 1), ACM, New York, 1982, pp.57-65.

[Lamp] Lampson, B. W. & Sturgis, H. E., *Crash recovery in a distributed storage system.* Comm. ACM, to appear.

[Lehma] Lehman, L. P., Bing Yao, S., *Efficient Locking for concurrent Operations on B-Trees.* ACM Transactions on Database Systems, Vol. 6, No. 4, Dec., 1981.

[Lynch] Lynch, N. A., *Multilevel Atomicity.* Proc. of ACM Symposium on Principles of Database Systems (Los Angeles, CA., Mar. 29-31), pp. 63-69

[Moss] Moss, J. E., *Nested Transactions: An Introduction.* U.S. Army War College, Carslile Barracks, PA.

[Silbe] Silberschatz, A., Kedem, Z., *Consistency in hierarchical database systems* J. ACM 27:1, pp. 72-80

[Stron] Stron, B.I., *Consistency of redundant databases in a weakly coupled distributed computer conferencing system.* Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer Networks (Emeryville, CA., Feb. 3-5), Lawrence Berkeley Lab. of the University of California at Berkeley, Berkeley, CA., 1981, pp. 143-153.

[Verho] Verhofstadt, J. S. M., *Recovery Technics for Database Systems* ACM Computing Surveys 10, 2 (June 1978), pp. 167-195.

[Wiede] Wiederhold, G., *Databases.* IEEE COMPUTER, Vol. 17, No. 10, Oct., 1984

Appendix A

This is the only appendix in this thesis. It is a replica from Appendix A in [Garc]. We describe here the management of locks by the SK transaction processing mechanism.

Associated with each object o in the database, we have:

$LL(o) =$ A boolean variable which indicates if the object has a local lock.

$GL(o) =$ A boolean variable indicating if the object has a global lock.

$SW(o) =$ the share-with set of a global lock. It is an interleaving descriptor giving the types of transactions that may share the global lock. (Only defined if $GL(o) =$ true.)

$REL(o) =$ the release set of a global lock. The global lock on o can only be released when all the transactions in this set have completed. (Only defined if $GL(o) =$ true.)

$PRE(o) =$ the set of transactions that have obtained a global lock on o but have not yet accessed it. The global lock cannot be released if this set is not empty. (Again, $PRE(o)$ is only defined if $GL(o) =$ true.)

Associated with each transaction T are the following sets:

$LL\_SET(T) =$ the set of objects on which T currently holds a local lock.

$GL\_SET(T) =$ the set of globally locked objects that have T in their $PRE(o)$ set.

$ID(T) =$ the interleaving descriptor being used by T to set global locks. If T is non-local, $ID(T)$ must be T's only descriptor; otherwise, $ID(T)$ can be any one of T's descriptors, or the empty set.

$WAIT(T)$ = the wait set of T. Transaction T accumulates in $WAIT(T)$ the $REL(o)$ sets of all the globally accessed objects it has accessed in the current step. The global locks obtained by T can only be released when all the transactions in $WAIT(T)$ reach their termination point.

$T\_WAIT(T)$ = the total wait set of T. This set is used to accumulate the $WAIT(T)$ sets obtained at the end of each step.

To simplify the presentation, we assume that when each non-local transaction T completes, it broadcasts a message to all nodes indicating this and containing $T\_WAIT(T)$. We also assume that each node X keeps a list, $DONE(X)$, of all the non-local transactions that have completed. Each node also keeps the $T\_WAIT(T)$ sets for all the non-local transactions in $DONE(X)$. We wish to emphasize that these assumptions are made in order to simplify the presentation only. In a system where broadcasts are not expensive this might actually be the best way for the mechanism to operate, but other alternatives are possible for other systems. We do not discuss these additional strategies in this paper.

We now outline the steps that must be followed by each transaction in locking and unlocking. (There are many variations or improvements which we do not discuss here due to space limitations.) Transactions that are locking or unlocking objects should not interfere with each other, so there must be a mutual exclusion mechanism to avoid this. For simplicity we do not discuss this mechanism here.

Comments are enclosed in double brackets " << >> ".

Step 1. Before a transaction T starts

$LL\_SET(T) \leftarrow \emptyset$; $GL\_SET(T) \leftarrow \emptyset$;

$WAIT(T) \leftarrow \emptyset$; $T\_WAIT(T) \leftarrow \emptyset$;

IF $ty(T) \in LOCAL$ THEN $ID(T) \leftarrow \emptyset$ ELSE $ID(T) \leftarrow id(ty(T))$;

Step 2. Before transaction T starts a step

"for as many objects o which we know in advance will be referenced by this step; and for each object o which may be referenced by its counter-step (if any) but not by the step, do:"

IF $\neg GL(o)$ THEN

BEGIN $GL(o) \leftarrow$ true; $SW(o) \leftarrow ID(T)$;

$PRE(o) \leftarrow \{T\}$; $REL(o) \leftarrow \emptyset$ END

ELSE IF $ID(T) = SW(o)$ AND $SW(o) \neq \emptyset$ THEN

$PRE(o) \leftarrow PRE(o) \bigcup \{T\}$

ELSE IF $ty(T) \in LOCAL$ AND $ID(T) = \emptyset$ AND $ty(T) \in SW(o)$ THEN

BEGIN << its time to select a descriptor for locking >>

$PRE(o) \leftarrow PRE(o) \bigcup \{T\}$;

$ID(T) \leftarrow SW(o)$;

FOR $p \in GL\_SET(T)$ DO $SW(p) \leftarrow SW(o)$

END

ELSE "wait and try global locking later";

$GL\_SET(T) \leftarrow GL\_SET(T) \bigcup \{o\}$;

Step 3. Before a step of transaction T is allowed to access object o

IF $\neg GL(o)$ OR $T \notin PRE(o)$ THEN

"perform global locking as in Step 2 above";

IF $\neg LL(o)$ THEN $LL(o) \leftarrow$ true

ELSE "wait and try local locking later";

$LL\_SET(T) \leftarrow LL\_SET(T) \bigcup \{o\}$;

$WAIT(T) \leftarrow WAIT(T) \bigcup REL(o)$;

Step 4. When a transaction T completes a step at node X

A. IF T is a local transaction

$<<$ T has completed its only step. At this point $LL\_SET(T)$ must equal $GL\_SET(T)$. $>>$

$T\_WAIT(T) \leftarrow WAIT(T)$;

WHILE "there is an $R \in T\_WAIT(T)$ not yet processed in this loop such that $R \in DONE(X)$" DO

$T\_WAIT(T) \leftarrow (T\_WAIT(T) - \{R\}) \bigcup T\_WAIT(R)$;

FOR $p \in LL\_SET(T)$ DO

BEGIN $<<$ release locks $>>$

$REL(p) \leftarrow REL(p) \bigcup T\_WAIT(T)$; $LL(p) \leftarrow$ false;

$PRE(p) \leftarrow PRE(p) - \{T\}$;

IF $PRE(p) = \emptyset$ AND $REL(p) = \emptyset$ THEN $GL(p) \leftarrow$ false;

END;

B. If T is non-local completing a revocable step,

or a counter step, or its last step

$<<$ At this point $LL\_SET(T)$ should be a subset of $GL\_SET(T)$ $>>$

FOR $p \in LL\_SET(T)$ DO

BEGIN

$REL(p) \leftarrow REL(p) \bigcup \{T\}$; $LL(p) \leftarrow$ false;

END;

$LL\_SET(T) \leftarrow \emptyset$; $GL\_SET(T) \leftarrow \emptyset$;

$T\_WAIT(T) \leftarrow T\_WAIT(T) \bigcup WAIT(T); \quad WAIT(T) \leftarrow \emptyset;$

C. If T is non-local completing a non-revocable step

(except its last step)

"do nothing"

Step 5. If transaction T must be aborted

<< T should not be aborted if it is in the process of releasing locks>>

<< Undo all irrevocable steps (if any) and the currently executing step or counter-step>>

FOR $p \in LL\_SET(T)$ DO

BEGIN

"Restore object p to its original value (e.g., using log)";

$LL(p) \leftarrow$ false

END;

$LL\_SET(T) \leftarrow \emptyset; \quad WAIT(T) \leftarrow \emptyset;$

FOR $p \in GL\_SET(T)$ DO

BEGIN

$PRE(p) \leftarrow PRE(p) - \{T\};$

IF $PRE(p) = \emptyset$ AND $REL(p) = \emptyset$ THEN $GL(p) \leftarrow$ false

END;

$GL\_SET(T) \leftarrow \emptyset;$

"For all outstanding revocable steps, execute their counter-step (in order). Each of these counter-steps is executed just like a regular step, except that no pre-locking is necessary (i.e., skip Step 2)."

Step 6. When non-local transaction T completes all its steps

or all of its counter-steps

<< Step 4.B must have been executed at this point >>

"send to all nodes a completion message indicating that T has finished and containing $T\_WAIT(T)$.

Step 7. When a node X receives a completion message for T

(including T_WAIT(T))

$DONE(X) \leftarrow DONE(X) \bigcup \{T\}$;

WHILE "there is an $R \in T\_WAIT(T)$ not yet processed in this loop

such that $R \in DONE(X)$" DO

$T\_WAIT(T) \leftarrow (T\_WAIT(T) - \{R\}) \bigcup T\_WAIT(R)$;

FOR "each object p such that $GL(p)$ is true" DO

BEGIN

$PRE(p) \leftarrow PRE(p) - \{T\}$;

IF $T \in REL(p)$ THEN

$REL(p) \leftarrow (REL(p) - \{T\}) \bigcup T\_WAIT(T)$;

IF $REL(p) = \emptyset$ AND $PRE(p) = \emptyset$ THEN $GL(p) \leftarrow$ false;

END;

(End of Appendix A.)